ISSN 0190-3918
IEEE Computer Society Order Number 724
Library of Congress Number 79-640377
IEEE Catalog Number 86CH2355-6
ISBN 0-8186-0724-6

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

# PROCEEDINGS

OF THE

# 1986 INTERNATIONAL CONFERENCE

ON

# PARALLEL PROCESSING

August 19–22, 1986

Editors
Kai Hwang
Steven M. Jacobs
Earl E. Swartzlander

Co-Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the

IEEE Computer Society
In Cooperation with the

Association for Computing Machinery

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE
COMPUTER
SOCIETY
PRESS

1986 INTERNATIONAL CONFERENCE
ON PARALLEL PROCESSING
Hwang • Jacobs • Swartzlander

ISSN 0190-3918
IEEE Computer Society Order Number 724
Library of Congress Number 79-640377
IEEE Catalog Number 86CH2355-6
ISBN 0-8186-0724-6

IEEE
COMPUTER
SOCIETY
PRESS

**ISSN 0190-3918**
**IEEE Computer Society Order Number 724**
**Library of Congress Number 79-640377**
**IEEE Catalog Number 86CH2355-6**
**ISBN 0-8186-0724-6**

# PROCEEDINGS

OF THE

# 1986 INTERNATIONAL CONFERENCE

ON

# PARALLEL PROCESSING

August 19–22, 1986

Editors
Kai Hwang
Steven M. Jacobs
Earl E. Swartzlander

Co-Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the

IEEE Computer Society
In Cooperation with the

Association for Computing Machinery

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL
AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE
COMPUTER
SOCIETY
PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

ISSN 0190-3918
IEEE Computer Society Order Number 724
Library of Congress Number 79-640377
IEEE Catalog Number 86CH2355-6
ISBN 0-8186-0724-6 (paper)
ISBN 0-8186-4724-8 (microfiche)
ISBN 0-8186-8724-X (case)

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

# Preface

In this decade, the field of parallel processing has exploded! This conference provides clear proof: in the early 1980's on the order of 100 to 125 papers were submitted annually; in contrast, over 400 papers were submitted this year. Such growth is both gratifying and challenging. On the one hand, it indicates a high and growing level of interest in parallel processing techniques and technologies. The resulting parallel systems are sorely needed to provide computing resources for ever more demanding applications in science, medicine, commerce, and industry.

On the other hand, such growth also challenges the logistics implicit in organizing the conference. Consider, for example, the paper selection process: (1) papers are submitted to the program co-chairs, (2) the co-chairs reassign some papers to other program areas (based on the co-chairs' mutually agreed definitions of the subject areas), (3) at least three reviews are solicited for each paper, (4) as the program selection date approaches, additional reviews are solicited (on a "crash" basis) for those papers with fewer than three reviews, (5) each program co-chair makes a tentative selection of papers and organizes them into potential sessions, (6) the three co-chairs merge their tentative programs, coalescing overlapping sessions, deleting orphan papers (those good papers that don't quite fit with others to form cohesive sessions), and continue to raise the acceptance standards to meet the conference size constraints, and finally (7) acceptance and rejection letters are sent to the authors. The difficulty of this process increases in proportion to the number of submissions, and in inverse proportion to the paper acceptance rate.

For many conferences, a typical paper acceptance rate is 60% to 70%, in our case, we were limited by the conference facilities and by the desire that the proceedings be portable to a 45% acceptance rate. While such a low rate indicates that the papers that comprise this proceedings are of extremely high quality, it also indicates that many good papers had to be rejected. The impact is probably more profound on new entrants to the field and foreign (i.e., non-native English speaking) authors than upon experienced authors, which may give the appearance that the field has become "in-bred". This year's program committee has tried to avoid any such prejudice, but given our limitations, the program can only be viewed as a best approximation to the rapidly evolving state of the art of parallel processing in 1986.

It has been our privilege to have received support from a large and dedicated assembly of reviewers as listed on the next page. Without their rapid assistance, the conference could not occur. It is also a pleasure to identify the secretaries that have handled the various filing, copying, correspondence, and countless other chores required to develop the program. Our sincere thanks go to: Jenine Abarbanel, Vicki Adame, Lauren Hall, Alice Harris, Shirin Mistry, and Elaine Smiles. The continuous support and encouragement of the Conference General Chairman, Professor Tse-yun Feng has been most gratifying. Most of all, we thank the authors for taking the time and effort to share their work with the parallel processing community.

Program Co-Chairs:
Kai Hwang
Steve Jacobs
Earl Swartzlander

# Reviewers

Abeynayake, K.
Abraham, J. A.
Abu-Sufah, W.
Adams, G. B.
Agrawal, D. P.
Agrawala, A.
Agre, J. R.
Agusa, K.
Ahmad, O.
Akers, S. B.
Allen, F.
Allen, S. J.
Alnuweiri, H. M.
Alonso, R.
Amano, H.
Antony, R.
Arden, B. W.
Arvind
Ashcroft, E. A.
Atwood, J. W.
Axelrod, T.
Baer, J-L.
Bagherzadeh, N.
Batcher, K. E.
Bequillard, A. L.
Berman, F.
Berra, B.
Berry, M.
Best, E.
Bhargave, B.
Bhuyan, L. N.
Bic, L.
Bojanczyk, A.
Bouldin, D.
Bounds, P.
Breuer, M.
Briggs, F. A.
Bronson, E.
Brower, J.
Brown, G.
Bruner, J. D.
Bucher, L.
Bui, T. D.
Burger, J. R.
Burkowski, F.
Butner, S. E.
Buzbee, W.
Calahan, D. A.
Calhoun, J. R.
Canas, D. A.
Cappello, P.
Chatterjee, B.
Chen, C-Y.
Chen, M. C.
Chen, S-S.
Chen, T. C.
Cheng, H-D.
Chern, M-Y.
Chiang, Y. P.
Chiarulli, D. M.
Chin, C-Y.
Chin, F.
Chiou, I.
Chou, C-R.

Chow, Dr.
Chow, Y. C.
Chowkanyun, R.
Chu, P-Y. M.
Chuang, C. Y.
Chuang, H. Y. H.
Cimet, I. A.
Culik, K.
Culler, D.
Cuny, J.
Cytron, R.
Davidson, E. S.
Davis, N. J.
Davis, T. A.
Decker, W. F.
Dekel, E.
De Mori, R.
Demurjian, S. A.
Desai, B. C.
Deshpande, S. R.
Dhar, S.
Dhodi, M. K.
Dietz, H.
Dimopoulod, N.
Dongarra, J.
Doshi, K.
Drogin, E.
Du, D.
Dubois, M.
Dyer, C.
Easley, D. C.
Eggers, S. J.
Eisenstadter, Y.
Ellis, J. A.
Elmargaruid, A.
Eltgroth, P.
Encson, L.
Ercal, F.
Ercegovac, M.
Ericson, L.
Eshayhian, M. M.
Faigle, U.
Fam, A. T.
Feng, T-Y.
Finkel, R.
Flanders, P. M.
Flynn, M.
Foo, Y-P.
De Forcrand, P.
Ford, R.
Forrest, S.
Fortes, J. A. B.
Foster, M. J.
Fox, G. C.
Franklin, M. A.
Freeman, H.
Fucik, G.
Fung, L. W.
Gaboury, P.
Gajski, D.
Gallopoulos, E.
Gannon, D.
Gao, G. R.
Garcia, O.

Gasser, L.
Gaudiot, J. L.
Gazit, T.
Gendreau, T. B.
Ghosh, J.
Ghozati, S.
Glidden, R. M.
Goldwasser, S.
Goltz, W.
Goodman, J. R.
Gopalakrishnan, P. S.
Grogono, P.
Gu, J.
Guerra, C.
Gustafson, J. L.
Guzman, A.
Hall, N. R.
Hamacher, V. C.
Hankin, C. L.
Harrod, W. J.
Hawkinson, S.
Hayes, J. P.
Helmbold, D.
Hennessy, J. L.
Hightower, C. H.
Hiner III, F. P.
Hiranpruk, R.
Hirayama, M.
Hockeney, R. W.
Hoeflinger, J. P.
Hollaar, L.
Hong, Y-C.
Hope, G. S.
Horiguchi, S.
Hotzel, E.
Houstis, C.
Hsiao, D.
Hsiung, C.
Hsu, F. H.
Hsu, W. T.
Hudak, P.
Husman, H. E.
Hwang, K.
Ibarra, O. H.
Ipgen, S.
Ipsen, I.
Irani, K. B.
Irwin, M. J.
Jacobs, S. M.
Jacobson, D. W.
Jain, B.
Jalby, W.
Jamieson, L. H.
Janakiram, V. K.
Jenevein, R. M.
Jeng, M.
Jenkins, K.
Jin, L.
Johnson, A. M.
Johnsson, L.
Jordan, H. F.
Jump, R.
Kak, S. C.
Kale, L. V.

Kao, M-L.
Kaudel, F.
Keller, R.
Khatibi, F.
Kim, D.
Kime, C.
Kiyoki, Y.
Kogge, P. M.
Koren, I.
Krishnamurthy, B.
Krishnamurti, R.
Krothapalli, V. P.
Kruskal, C. P.
Kuck, D.
Kuehn, J. T.
Kumar, M.
Kumar, P. R. S.
Kumar, V. K. P.
Kung, S. Y.
Kuo, H-C.
Kwok, A. Y.
Lai, S.
Lakshmivaraham, S.
Lam, M.
Lang, T.
Larson, J.
Lauwereins, R.
Lawrie, D. H.
Lee, C.
Lee, C. S. G.
Lee, I.
Lee, J.
Lee, K.
Lee, M.
Lee, R. L.
Lee, S-Y.
Lee, W. W.
Leighton, F. T.
Leiserson, C. E.
Leu, J-S.
Li, G-J.
Li, Y.
Li, Z.
Liano, K.
van de Liefvoort, A.
Lilien, L.
Lin, F.
Lin, H. C.
Lin, S-C.
Lin, T-C.
Lin, T-T.
Lipovski, G. J.
Liu, A. C.
Lo, S-C.
Louri, A.
Lu, D. C. S.
Lu, M.
Ma, E.
Madefl, S.
Madison, D. E.
Mahgoub, I. O.
Malek, M.
Malony, A. D.
Mazumder, P.

McAulay, A. D.
McGee, K. J.
McGurk, F.
McMillen, R. J.
McNiven, G.
Mendoza-Grado, V.
Meyer, D. G.
Meyer, G. G. L.
Mierendorff, H.
Miller, G.
Minden, G. J.
Miranker, W. L.
Mitra, D.
Moldovan, D. I.
Mossaad, K.
Mudge, T. N.
Mulder, H. M.
Murata, T.
Naik, V. K.
Najjar, W. S.
Narahari, B.
Nash, J. G.
Nelson, P. A.
Ni, L. M.
Nuth, P.
Obaidat, M. S.
Opsahl, T.
Ortega, J. M.
Oruc, A. Y.
Oxley, D. W.
Parberry, I.
Patterson, D. A.
Payne, T. H.
Peir, J-K.
Petrie, C. J.
Pham, S.
Phister, G. F.
Polychronopoulos, C.
Poplawski, D. A.
Potter, J. L.
Pradhan, D. K.
Price, C.
Przytola, K. W.
Quinn, M. J.
Raghavendra, C. S.
Rajopadhye, S. V.
Ramakrishnan, I. V.
Ramamoorthy, C. V.
Reddy, S. M.
Reeves, A. P.
Reisis, D.
Rice, J. R.
Ruhman, S.
Sadayappan, P.
Sahni, S.
Saltz, J. H.
Sang, A.
Sawchuk, A.
Scherson, I.
Scheurich, C.
Schultz, W. L.
Schwartz, D. A.
Schwartz, J. T.
Schwederski, T.

Schwetman, H. D.
Scott, K.
Seitz, C. L.
Sen, S.
Sethi, A. S.
Shaffer, P. L.
Shapiro, E.
Sharp, E.
Shaw, D.
Shoja, G. C.
Shu, J. C.
Shu, K-S.
Siegel, H. J.
Siewiorek, D. P.
Simon, H. D.
Sips, H. J.
Smitley, D.
Staley, C. A.
Stankovic, J.
Sterling, T.
Stolfo, S. J.
Stout, Q. F.
Sugla, B.
Swartzlander, E. E.
Szymanski, T. H.
Taha, M. S.
Takefuji, Dr.
Tang, J.
Tang, P.
Tang, W. P.
Tanimoto, S. L.
Taylor, F. J.
Thanawastien, S.
Thomas, R. H.
Tian, S.
Tokoro, M.
Tokuta, A. O.
Torng, H. C.
Tseng, P. S.
Tsin, Y. H.
Tu, S-C.
Tzeng, N-F.
Utka, S.
Uyar, M. U.
Varadarajan, R.
Varma, A.
Varman, P.
Venkataraman, K. N.
Vishnubhotla, P.
Wagner, R. A.
Wah, B.
Waid, P. B.
Wang, Y-X.
Ward, C.
Wei, Y-H.
Wells, D.
Wetzel, G. F.
Wing, O.
Wu, C. L.
Xu, Z.
Yau, K.
Yew, P-C.
Yoder, M.
Zak, S. H.

# AUTHOR INDEX

# TABLE OF CONTENTS

**Session Ia: Interconnection Networks I**

**Session 1b: Memory Management I**

**Session 1c: Mathematical Analysis and Computation**

**Session 2a: Interconnection Networks II**

**Session 2b: Memory Management II**

**Session 2c: Parallel Processing Languages**

**Session 3a: Interconnection Networks III**

**Session 3b: Operating Systems**

**Session 3c: Parallel Algorithms I**

**Session 4a: Array Processors**

**Session 4b: Fault Tolerant Designs**

**Session 4c: Parallel Algorithms II**

**Session 5a: VLSI and Optical Architectures**

**Session 5b: Performance Analysis and Fault Tolerence**

**Session 5c: Parallel Algorithms III**

**Session 6a: Multiprocessors I**

**Session 6b: Pipeline/Vector Processors**

**Session 6c: Parallel Computation I**

**Session 7a: Multiprocessors II**

**Session 7b: Dataflow Computers**

**Session 7c: Parallel Computation II**

**Session 8a: Hypercube Architecture**

**Session 8b: Systolic Architecture I**

**Session 8c: Artificial Intelligence**

**Session 9a: Supercomputing Hardware**

**Session 9b: Systolic Architecture II**

**Session 9c: Simulation and Modeling**

**Session 10a: Multigrid/Mesh Applications**

**Session 10b: Vectorization/Data Flow Processing**

**Session l0c: Graph Theory**

**Session 11a: Parallel Processing Techniques**

**Session 11b: Data Structures I**

**Session 11c: Distributed Processing I**

**Session 12a: Logic Programming**

**Session 12b: Data Structures II**

**Session 12c: Distributed Processing II**

# AN AUTOMORPHISM OF A CLASS OF INTERCONNECTION NETWORKS

WOEI LIN

*Department of Electrical Engineering*
*The Pennsylvania State University*
*University Park, PA 16802*

## ABSTRACT

A class of interconnection schemes, commonly referred to as multistage interconnection networks, is proposed as a means, providing simultaneous connections among processing elements in multiprocessor systems. In this paper, we present an automorphism, with which the logical structure of the interconnection networks can be changed. We show that the resulting logical structure of a network is isomorphic to its physical structure. For the purpose of demonstration, three popular networks are examined: baseline network, omega network and indirect n-cube network.

## I. INTRODUCTION

A class of multistage interconnection networks have been proposed, and many of them have been implemented for providing simultaneous, reconfigurable connections among processing elements in multiprocessor systems. These networks include baseline network[1]-[3], flip network[4], omega network[5], and many others[6]-[8]. Advantages of these networks include cost effectiveness, logarithmic communication delay, modular expansibility, and partitionabilit. The primary goal of this work is to show that for each network of this class, there exist a multiplicity of logical network structures, which are isomorphic to its physical network structure. More specifically, we will present a one-to-one correspondence, which turns out to be an automorphism, changing logical structures of these networks. To illustrate this idea, three popular multistage interconnection networks are chosen for demonstration: baseline network, indirect n-cube network and omega network.

The rest of this work is organized as follows. In Section II, we will describe the integration of the networks, including network components and general network configuration. In Section III, we will present the definition of a one-to-one correspondence. Section IV contains the proofs, showing that this correspondence is an automorphism with respect to the three networks.

## II. PRELIMINARIES

### 2.1 Configuration of Networks

The multistage interconnection networks to be examined are characterized by three attributes: (1) switching element, (2) arrangement of switching elements, and (3) permutation pattern between stages of switching elements.

(1)Switching element. These networks employ a 2x2 crossbar switch as a buliding block, as shown in Figure 1. This 2x2 switching element has two inputs and two output, denoted $X_1$, $X_2$, $Y_1$, and $Y_2$. It has the capability of connecting the input $X_1$ to either the output $Y_1$ or the output $Y_2$, depending on the value of some routing bit of the input $X_1$. If the routing bit is 0, the input is connected to the output $Y_1$, and if the routing bit is 1, the connection is made to the output $Y_2$. Input $X_2$ of the switch behaves similarly with a routing bit.

(2)Arrangement of switching element. These networks is composed of a logarithmic number of stages of switching elements; a network of size $N=2^n$ comprises n stages of switching elements. Furthermore, each stage comprises N/2 switching elements, as described above. Consequently, an interconnection network of such a configuration has *N network inputs* and *N network outputs* and contains $(N/2)\log_2 N$ switching elements.

(3)Permutation pattern between stages of switching elements. In a network, two adjacent stages of switching elements are cascaded by a set of N communication links, from the outputs of switching elements of the preceding stage to the inputs of switching elements of the succeeding stage. In a network of size N, there are $(\log_2 N)+1$ levels of communication links. Each level is associated with a specific permutation pattern. It should be noted that these networks, however, are different in the permutation patterns between stages of switching elements. In Section IV the permutation patterns of each individual network will be described.

### 2.2 Addressing of networks

To facilitate describing the configurations of these interconnection networks, a uniform addressing scheme is presented. As shown in Figure 2, in a network of size $N=2^n$, the network inputs are addressed in a sequence from 0 to N-1, from top to bottom. In a similar way, the network outputs are addressed from 0 to N-1. Recall that an interconnection network of size N, as described in the previous subsection, contains $n=\log_2 N$ stages of switching elements and n+1 levels of links. The addressing schemes of stages and switching elements are depicted as follows. These n stages are labeled in a sequence from 0 to n-1 with 0 for the leftmost stage and n-1 for the rightmost stage. Similarly, the levels of links are labeled in a sequence from 0 to n. In each stage, the N/2 switching elements are addressed from 0 to N/2-1, each of which can also be represented by an (n-1)-bit binary number of the form, $s_{n-1}s_{n-2}...s_1$. In each stage, each switching element is associated with four communication links; for a switching element labeled by $s_{n-1}s_{n-2}...s_1$, the upper link to the input $X_1$ is addressed by an n-bit binary number, $s_{n-1}s_{n-2}...s_1 0$, and the lower one to the input $X_2$ is addressed by $s_{n-1}s_{n-2}...s_1 1$. Similarly, the two links from outputs $Y_1$ and $Y_2$ are addressed by $s_{n-1}s_{n-2}...s_1 0$ and $s_{n-1}s_{n-2}...s_1 1$, respectively.

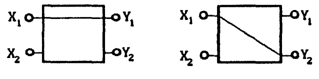
From topological point of view, these interconnection networks are different in their permutation patterns between stages of switching elements. To describe the permutation patterns of an interconnection network, the following notation will be adopted:

$$\gamma_i(Y)=X.$$

For a network of size $N=2^n$, $\gamma_i$ specifies the permutation of communication links of level i (from the outputs of stage i-1 to the inputs of stage i of switching elements). However, in the case of i=0, the input domain of $\gamma$ is the set of network inputs. Similarly, in the case of i=n, the output domain of $\gamma$ is the set of network outputs. For instance, if the permutation pattern of level i is a perfect shuffle permutation[12], then we have

$$\gamma_i(Y)=(2Y+\lfloor 2Y/N \rfloor) \bmod N,$$

for all the outputs of switching elements of stage i-1. That is, the output, Y, of stage i-1 is connected to the input, $((2Y+\lfloor 2Y/N \rfloor) \bmod N)$, of stage i.



Routing bit of $X_1$ is 0 Routing bit of $X_1$ is 1

Figure 1. A 2x2 switching element.

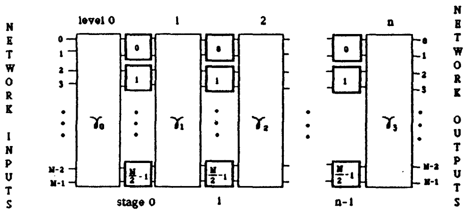


Figure 2. Configuration of a multistage interconnecton network of size $N=2^n$.

0190-3918/86/0000/0001 $01.00 © 1986 IEEE

## III. THE DEFINITION OF $\xi$

This is the purpose of this section to define a one-to-one correspondence, which turns out to be an automorphism, with respect to the multistage interconnection networks. Also, we will present two functions, which are used to rearrange the switching elements and the topology of the five networks.

**DEFINITION.** Let S be a set of n-bit binary numbers from 0 to $2^n-1$, or $S=(0,1,\dots 2^n-1)$. And let C be a constant, $0\leq C\leq 2^n-1$. $\xi_C$ is a mapping from S to S, and $\xi_C(X)=X\oplus C$, for every $X\epsilon S$.

With the definition above, we now present the statement of the problem to be investigated as follows.

**STATEMENT OF THE PROBLEM:** If we relabel the network inputs and outputs of a multistage interconnection network by $\xi_p$ and $\xi_Q$, respectively, then there exists a way of rearranging the switching elements and links such that the topology of the rearranged network is equivalent to that of the original network. That is, both of the two topologies are isomorphic.

To illustrate the statement above, an example is given in Figure 3. Consider a baseline network of size 16, whose network inputs and outputs are relabeled through $\xi_{11}$ and $\xi_6$, respectively. What we intend to show is that the two network topologies are isomorphic. For convenience, hereafter $\xi_{p:Q}$ denotes "apply $\xi_p$ to the network inputs and $\xi_Q$ to the network outputs of a multistage network."

Before presenting the proof that $\xi_{p:Q}$ is an automorphism onto the multistage interconnection network, we further introduce two functions, which will be used to rearrange switching elements and their associated links:

(1)MOVE $\mu_i$: This is a one-to-one correspondence, which is used to rename the switching elements of stage i, according to some mapping rule.

(2)TWIST $\tau_i$: This is used to interchange a pair of inputs or outputs of a switching element of stage i. Associated with $\tau_i$, there are two control bits $c_x$ and $c_y$, which are used to specify the interchange on the input side and the output side of a switching element, respectively. As illustrated in Figure 4, there are four possible operations of $\tau_i$ on a switching element:

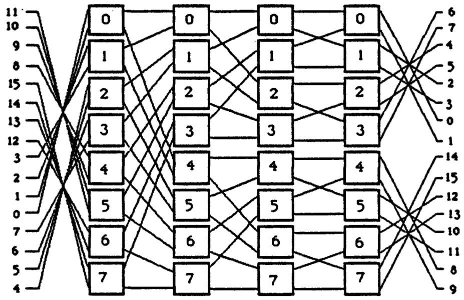


Figure 3. A baseline network whose network inputs and outputs are permuted by $\xi_{11:6}$.

1. *No-twist.* When $c_x=0$ and $c_y=0$, no operation is performed on the switching element;
2. *Left-twist.* When $c_x=1$ and $c_y=0$, the input side of the switching element is twisted 180° counterclockwise;
3. *Right-twist.* When $c_x=0$ and $c_y=0$, the output side of the switching is twisted 180° clockwise;
4. *Vertical-flip.* When $c_x=1$ and $c_y=1$, the switching element is flipped vertically.

## VI. THE AUTOMOPHISM OF NETWORKS

In this section, we prove that $\xi_{p:Q}$ is an automorphism onto the three multistage interconnection networks: baseline network, omega network, and indirect n-cube network. (However, only the proof for the baseline network is presented, and the other two proofs for the omega network and the indirect n-cube network are omitted.) Throughout the discussion, we assume that $X=x_{n-1}x_{n-2}\dots x_0$ is an input and $Y=y_{n-1}y_{n-2}\dots y_0$ is an output of a switchig element. For the reason of uniform notation, here we define that $\tau_{-1}(\mu_{-1}(Y))=\xi_P(Y)$ and $\tau_n(\mu_n(X))=\xi_Q(X)$; that is, network inputs are considered as the outputs of a virtual stage, Stage -1, and similarly network outputs are considered as the inputs of a virtual stage, Stage n.

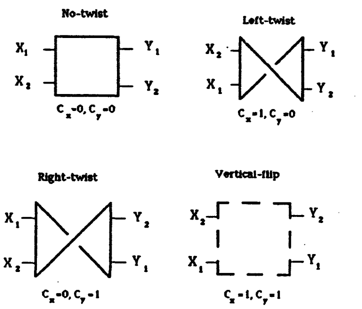


Figure 4. Four combinations of $\mu$ and $\tau$ on a switching element.

### 4.1 Baseline Network

The permutation patterns of a baseline network of size $N=2^n$ are defined by:

$$\gamma_i(Y)=x_{n-1}x_{n-2}\dots x_0 = \begin{cases} y_{n-1}y_{n-2}\dots y_0 & i=0,\\ y_{n-1}\dots y_{n-i+1}y_0y_{n-i}\dots y_1 & 1\leq i\leq n-1,\\ y_{n-1}y_{n-2}\dots y_0 & i=n. \end{cases} \qquad (1)$$

Let P and Q be two n-bit numbers, and $P=p_{n-1}p_{n-2}\dots p_0$, $Q=q_{n-1}q_{n-2}\dots q_0$. The function of $\mu_i$ is to move a switching element $(z_{n-1}z_{n-2}\dots z_1)$ of stage i to a new location

$$\begin{cases} (z_{n-1}z_{n-2}\dots z_1)\oplus(p_{n-1}p_{n-2}\dots p_1) & i=0,\\ (z_{n-1}z_{n-2}\dots z_1)\oplus(q_{n-1}\dots q_{n-i}p_{n-1}\dots p_{i+1}) & 1\leq i\leq n-2,\\ (z_{n-1}z_{n-2}\dots z_1)\oplus(q_{n-1}q_{n-2}\dots q_1) & i=n-1. \end{cases}$$

In terms of binary addresses of the communication links associated with switching elements of stage i, the function of $\mu_i$ can be explicitly described by

$$\mu_i(X)=\begin{cases} (x_{n-1}x_{n-2}\dots x_0)\oplus(p_{n-1}p_{n-2}\dots p_10) & i=0\\ (x_{n-1}x_{n-2}\dots x_0)\oplus(q_{n-1}\dots q_{n-i}p_{n-1}\dots p_{i+1}0) & 1\leq i\leq n-2\\ (x_{n-1}x_{n-2}\dots x_0)\oplus(q_{n-1}q_{n-2}\dots q_10) & i=n-1 \end{cases}$$

and

$$\mu_i(Y)=\begin{cases} (y_{n-1}y_{n-2}\dots y_0)\oplus(p_{n-1}p_{n-2}\dots p_10) & i=0\\ (y_{n-1}y_{n-2}\dots y_0)\oplus(q_{n-1}\dots q_{n-i}p_{n-1}\dots p_{i+1}0) & 1\leq i\leq n-2\\ (y_{n-1}y_{n-2}\dots y_0)\oplus(q_{n-1}q_{n-2}\dots q_10) & i=n-1. \end{cases} \qquad (2)$$

In addition, $\tau_i$ is defined by

$$\tau_i(X)=(x_{n-1}x_{n-2}\dots x_0)\oplus(0\dots 0p_i) \qquad 0\leq i\leq n-1$$

and

$$\tau_i(Y)=(y_{n-1}y_{n-2}\dots y_0)\oplus(0\dots 0q_{n-i-1}) \qquad 0\leq i\leq n-1. \qquad (3)$$

The equivalent function of $\tau_i$, as defined above, is to left-twist all the switching elements at stage i if $p_i=1$ and right-twist the switching elements if $q_{n-i-1}=1$. With $\tau_i$ and $\mu_i$ above, we now prove that there always exists an equivalence relationship between a baseline network and its rearranged network by $\xi_{p:Q}$.

**THEOREM 1.** $\xi_{p:Q}$, $0\leq P,Q\leq 2^n-1$, is an automorphism onto a baseline network of size $2^n$.

**Proof**: Our approach is to show that after the switching elements of stage i-1 are rearranged by $\tau_{i-1}$ and $\mu_{i-1}$, and the switching elements of stages i are rearranged by $\tau_i$ and $\mu_i$, the rules of (1) still hold for all the links of level i. Thus we have

$$\gamma_i(\tau_{i-1}(\mu_{i-1}(Y)))=\begin{cases} (y_{n-1}\oplus p_{n-1})(y_{n-2}\oplus p_{n-2})\dots(y_0\oplus p_0) & i=0,\\ (y_0\oplus q_{n-1})(y_{n-1}\oplus p_{n-1})\dots(y_1\oplus p_1) & i=1,\\ (y_{n-1}\oplus q_{n-1})\dots(y_{n-i+1}\oplus q_{n-i+1})(y_0\oplus q_{n-i}) & \\ \qquad (y_{n-i}\oplus p_{n-1})\dots(y_1\oplus p_i) & 2\leq i\leq n-1,\\ (y_{n-1}\oplus q_{n-1})(y_{n-2}\oplus q_{n-2})\dots(y_0\oplus q_0) & i=n. \end{cases}$$

Furthermore, we have

$$\tau_i(\mu_i(X))=\begin{cases} (x_{n-1}\oplus p_{n-1})(x_{n-2}\oplus p_{n-2})\dots(x_0\oplus p_0) & i=0,\\ (x_{n-1}\oplus q_{n-1})(x_{n-2}\oplus p_{n-1})\dots(x_0\oplus p_1) & i=1,\\ (x_{n-1}\oplus q_{n-1})\dots(x_{n-i+1}\oplus q_{n-i+1})(x_{n-i}\oplus q_{n-i}) & \\ \qquad (x_{n-i-1}\oplus p_{n-1})\dots(x_0\oplus p_i) & 2\leq i\leq n-1,\\ (x_{n-1}\oplus q_{n-1})(x_{n-2}\oplus q_{n-2})\dots(x_0\oplus q_0) & i=n. \end{cases}$$

As a result, we have $\gamma_i(\tau_{i-1}(\mu_{i-1}(Y)))=\tau_i(\mu_i(X))$, for all $0\leq i\leq n$. □

For instance, Figure 5 illustrates a rearranged baseline network of size 16 by $\xi_{11:6}$ and $\tau$ as well as $\mu$ as defined by (2) and (3). Note that the rearranged network is topologically equivalent to the baseline network.

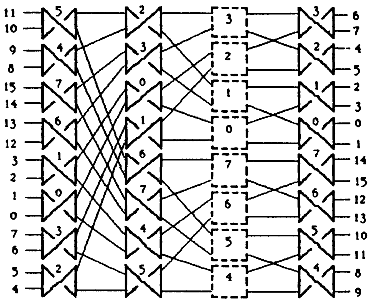


Figure 5. Configuration of a rearranged baseline network.

### 4.2 Omega Network

The permutation patterns of an omega network of size $N=2^n$ are described by:

$$\gamma_i(Y)=x_{n-1}x_{n-2}\cdots x_0 = \begin{cases} y_{n-2}y_{n-3}\cdots y_0\, y_{n-1} & 0\le i\le n-1, \\ y_{n-1}y_{n-2}\cdots y_0 & i=n. \end{cases} \qquad (4)$$

Let P and Q be two n-bit numbers, and $P=p_{n-1}p_{n-2}\cdots p_0$, $Q=q_{n-1}q_{n-2}\cdots q_0$. The function of $\mu_i$ is to move a switching element $(z_{n-1}z_{n-2}\cdots z_1)$ of stage i to a new location

$$\begin{cases} (z_{n-1}z_{n-2}\cdots z_1)\oplus(p_{n-2}p_{n-3}\cdots p_0) & i=0 \\ (z_{n-1}z_{n-2}\cdots z_1)\oplus(p_{n-i-2}\cdots p_0q_{n-1}\cdots q_{n-i}) & 1\le i\le n-2 \\ (z_{n-1}z_{n-2}\cdots z_1)\oplus(q_{n-1}q_{n-2}\cdots q_1) & i=n-1. \end{cases}$$

In terms of binary addresses of the communication links, $\mu_i$ can be described by

$$\mu_i(X)=\begin{cases} (x_{n-1}x_{n-2}\cdots x_0)\oplus(p_{n-2}p_{n-3}\cdots p_0 0) & i=0, \\ (x_{n-1}x_{n-2}\cdots x_0)\oplus(p_{n-i-2}\cdots p_0q_{n-1}\cdots q_{n-i}0) & 1\le i\le n-2, \\ (x_{n-1}x_{n-2}\cdots x_0)\oplus(q_{n-1}q_{n-2}\cdots q_1 0) & i=n-1. \end{cases}$$

and

$$\mu_i(Y)=\begin{cases} (y_{n-1}y_{n-2}\cdots y_0)\oplus(p_{n-2}p_{n-3}\cdots p_0 0) & i=0, \\ (y_{n-1}y_{n-2}\cdots y_0)\oplus(p_{n-i-2}\cdots p_0q_{n-1}\cdots q_{n-i}0) & 1\le i\le n-2, \\ (y_{n-1}y_{n-2}\cdots y_0)\oplus(q_{n-1}q_{n-2}\cdots q_1 0) & i=n-1. \end{cases} \qquad (5)$$

In addition, we define $\tau_i$ by

$$\tau_i(X)=(x_{n-1}x_{n-2}\cdots x_0)\oplus(0\ldots 0p_{n-i-1}) \qquad 0\le i\le n-1,$$

and

$$\tau_i(Y)=(y_{n-1}y_{n-2}\cdots y_0)\oplus(0\ldots 0q_{n-i-1}) \qquad 0\le i\le n-1. \qquad (6)$$

The equivalent function of $\tau_i$, as defined above, is to left-twist all the switching elements at stage i if $p_{n-i-1}=1$ and right-twist the switching elements if $q_{n-i-1}=1$. With $\tau_i$ and $\mu_i$ above, we now prove that there always exists an equivalence relationship between an omega network and its rearranged network by $\xi_{P:Q}$

**THEOREM 2.** $\xi_{P:Q}$, $0\le P,Q\le 2^n-1$, is an automorphism onto an omega network of size $2^n$.

### 4.3 Indirect n-Cube Network

In an indirect n-cube network of size $N=2^n$, the permutation patterns of n stages of links are defined by:

$$\gamma_i(Y)=x_{n-1}x_{n-2}\cdots x_0 = \begin{cases} y_{n-1}y_{n-2}\cdots y_0 & i=0, \\ y_{n-1}\cdots y_2y_0y_1 & i=1, \\ y_{n-1}\cdots y_{i+1}y_0y_{i-1}\cdots y_1y_i & 2\le i\le n-2, \\ y_0y_{n-2}\cdots y_1y_{n-1} & i=n-1, \\ y_0y_{n-1}y_{n-2}\cdots y_1 & i=n. \end{cases} \qquad (7)$$

Let P and Q be two n-bit numbers, and $P=p_{n-1}p_{n-2}\cdots p_0$ and $Q=q_{n-1}q_{n-2}\cdots q_0$. The function of $\mu_i$ is to move a switching element $(z_{n-1}z_{n-2}\cdots z_1)$ of stage i to a new location

$$\begin{matrix} (z_{n-1}z_{n-2}\cdots z_1)\oplus(p_{n-1}p_{n-2}\cdots p_1) & i=0, \\ (z_{n-1}z_{n-2}\cdots z_1)\oplus(p_{n-1}\cdots p_{i+1}q_{i-1}\cdots q_0) & 1\le i\le n-2, \\ (z_{n-1}z_{n-2}\cdots z_1)\oplus(q_{n-2}q_{n-3}\cdots q_0) & i=n-1. \end{matrix}$$

In terms of the binary addresses of communication links associated with switching elements of stage i, we have

$$\mu_i(X)=\begin{cases} (x_{n-1}x_{n-2}\cdots x_0)\oplus(p_{n-1}p_{n-2}\cdots p_1 0) & i=0, \\ (x_{n-1}x_{n-2}\cdots x_0)\oplus(p_{n-1}\cdots p_{i+1}q_{i-1}\cdots q_0 0) & 1\le i\le n-2, \\ (x_{n-1}x_{n-2}\cdots x_0)\oplus(q_{n-2}q_{n-3}\cdots q_0 0) & i=n-1, \end{cases}$$

and

$$\mu_i(Y)=\begin{cases} (y_{n-1}y_{n-2}\cdots y_0)\oplus(p_{n-1}p_{n-2}\cdots p_1 0) & i=0, \\ (y_{n-1}y_{n-2}\cdots y_0)\oplus(p_{n-1}\cdots p_{i+1}q_{i-1}\cdots q_0 0) & 1\le i\le n-2, \\ (y_{n-1}y_{n-2}\cdots y_0)\oplus(q_{n-2}q_{n-3}\cdots q_0 0) & i=n-1. \end{cases} \qquad (8)$$

In addition, we define

$$\tau_i(X)=(x_{n-1}x_{n-2}\cdots x_0)\oplus(0\ldots 0p_i) \qquad 0\le i\le n-1$$

and

$$\tau_i(Y)=(y_{n-1}y_{n-2}\cdots y_0)\oplus(0\ldots 0q_i) \qquad 0\le i\le n-1. \qquad (9)$$

The equivalent function of $\tau_i$, as defined above, is to left-twist all of the switching elements at stage i if $p_i=1$ and right-twist the switching elements if $q_i=1$. With $\tau_i$ and $\mu_i$ above, we now prove that there always exists an equivalence relationship between an indirect n-cube network and its rearranged network by $\xi_{P:Q}$

**THEOREM 3.** $\xi_{P:Q}$, $0\le P,Q\le 2^n-1$, is an automorphism onto an indirect n-cube network of size $2^n$.

## VI. CONCLUSION

**We have presented a one-to-one correspondence, $\xi_{P:Q}$, used to rename network inputs and outputs for a class of multistage interconnection networks. It is then proved that a renamed network is isomorphic to its parent network; that is, $\xi_{P:Q}$ is an automorphism. The class of multistage interconnection networks we have shown includes baseline network, omega network and indirect n-cube network. To illustrate that $\xi_{P:Q}$ is an automorphism, schemes of rearranging switching elements for each network are described and demonstrated in detail.**

## REFERENCES

[1] C. Wu and T. Feng, "On a class of interconnection networks," *IEEE Trans on Computers*, Vol. C-29, No. 8, pp. 694-702, Aug. 1980.

[2] C. Wu, et al., "Prototype of Star architecture-a statue report," *Proc. 1985 NCC*, pp.191-202.

[3] C. Wu and T. Feng, "The reverse-exchange interconnection network," *IEEE Trans on Computers*, Vol. C-29, No. 9, Sept. 1980. pp.801-811.

[4] K. Batcher, "The flip network in STARAN," *Proc. 1976 Int'l Conf. Parallel Processing*, pp.65-71.

[5] D. Lawrie, "Acess and alignment of data in an array processor, " *IEEE Trans. on Computers*, Vol. C-24, pp. 1145-1155, Dec. 1975.

[6] H. Siegel and S. Smith, "Study of multistage SIMD interconnection networks," *Proc. Fifth Annual Symp. Computer Arch.*, pp. 223-229.

[7] L. Goke and G. Lipovski, "Banyan networks for partitioining multiprocess-ing systems," *Proc. First Annual Symp. Computrt Arch.*, pp.21-28.

[8] M. Pease, III, "The indirect binary n-cube microprocessor array," *IEEE Trans. on Computers*, Vol. C-24, pp. 458-473, May 1977.

[9] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. on Computers*, Vol. C-20, pp. 153-161, Feb. 1971.

# EFFECT OF ARBITRATION POLICIES ON THE PERFORMANCE OF INTERCONNECTION NETWORKS

**Laxmi N. Bhuyan**

**Center for Advanced Computer Studies**
**University of Southwestern Louisiana**
**Lafayette, LA 70504-4330.**

## ABSTRACT

In performance evaluation of interconnection networks it is usually assumed that in case of conflicts a request is accepted with equal probability. This paper illustrates that this arbitration policy is discriminatory to remote or less frequent requests because they are rejected most of the time. The paper considers a favorite memory environment as an example and examines the network performance under various arbitration policies. Equal Acceptance, priority to favorite and non favorite request policies are examined by defining three probabilities of acceptance. The networks considered are crossbar, multiple-bus and multistage interconnection networks.

## 1. INTRODUCTION

A tightly coupled multiprocessor is usually categorized depending on the interconnection network (IN) that is used between the processors and memories [1]. Such IN's can be broadly divided into: (a) Crossbar (b) Multiple-bus (c) Multistage interconnection network (MIN). A crossbar interconnection allows all possible connections between the processors and memories [2]. When two or more processors try to access the same memory module only one of the requests is accepted and the others are blocked or rejected. A multiple-bus provides a fault-tolerant and cost effective interconnection between the processors and memories [3-5], but the number of simultaneous connections is dependent on the number of buses. When the number of buses is sufficient, a multiple-bus has the same performance as that of a crossbar. An MIN, on the other hand, is composed of several stages of switching elements (SE's) [6]. A conflict occurs when two or more requests contend for the same output of an SE. The cost and performance of such an IN is a reasonable balance between a shared bus and a crossbar. There is an extensive literature on the performance evaluation of these networks. All evaluations for synchronous multiprocessors measure Bandwidth (BW) which is defined as the number of memories remaining busy in a cycle.

The BW does not give us an idea about which requests are accepted and which are rejected. For equally likely case when a processor generates a request that is equiprobably directed to all the memories, it may not be essential to know the above details because all the processors perform similarly in the long term. However, when the processors request different memories with different probabilities, the equal acceptance (EA) rule appears discriminating because a processor with a remote request to a memory will be rejected most of the times. Rather, it should be given a priority over another processor which requests that particular memory more often. Hence, the previous analytical models of the BW do not reveal this important information that is so practical of the multiprocessor operation. As we know, no closed form solutions can be derived for general or arbitrary memory references. We will therefore restrict our analysis to favorite memory cases as depicted in Fig.1. For simplicity, we will restrict our analyses to $N \times N$ networks that connect $N$ processors to $N$ memories. Also, because of page restrictions some final equations will be given without derivations.

In Fig.1, a processor $P_i$ requests its favorite memory $MM_i$ with a probability of $m$ provided it generates a request. The processor requests all other $(N\text{-}1)$ memories with a probability of $1\text{-}m$. Assuming that those requests are uniformly distributed, the probability that a processor requests any one of those memories is $\frac{p_0(1\text{-}m)}{(N\text{-}1)}$, where $p_0$ is the probability of its request generation. When $m > \frac{1}{N}$, it is called a favorite memory case and by putting $m = \frac{1}{N}$, the analysis reduces to an equally likely case. The rate of request at a memory $MM_i$ is then $p_0.m$ due to its favorite processor $P_i$ and $\frac{p_0(1\text{-}m)}{(N\text{-}1)}$ due to any other processor provided there is no conflict in the $IN$.

## 2. ANALYSIS FOR CROSSBAR

The crossbar allows all processor requests to reach the memory modules. If more than one request reach a particular memory, one of them is accepted and the others are rejected. Which one is accepted depends on the arbitration policy of the controller. The rate of request $(p_f)$ at a memory $MM_i$ in an $N \times N$ crossbar is [7] :

$$p_f = 1 - (1 - p_0.m)(1 - p_0.\frac{1-m}{N-1})^{N-1}. \tag{1}$$

The BW, given by $N.p_f$, does not depend on the basis upon which the selection is made. The probability of favorite acceptance $(p_{fe})$ at a memory due to EA policy is given by:

$$p_{fe} = \frac{m.(N-1)}{N.(1-m)}.(1-(1 - p_0.\frac{1-m}{N-1})^N). \tag{2}$$

For large values of $N$, $\lim_{N\to\infty} p_{fe} = \frac{m}{1-m}(1-e^{p_0(m-1)})$. As an example, with $p_0 = 1$ and $m = 0.8$, $\lim_{N\to\infty} p_{fe} = 0.725$ The probability of nonfavorite acceptance,

$$p_{ne} = p_f - p_{fe}. \tag{3}$$

With $p_0 = 1$ and $m = 0.8$, $\lim_{N\to\infty} p_{ne} = 0.111$. Since most of the time a memory receives requests from its favorite processor, the requests from the other processors will be mostly unsuccessful in an EA policy. In a practical design a selection policy is fixed and is based on some priority structure. It is therefore reasonable to assign a priority to the non-favorite requests for higher values

This research is supported in part by NSF Grant No. DMC-8513041.

of $m$. Hence, whenever there is a nonfavorite request at $MM_i$, any request from $P_i$ will not be granted. We will assume that when more than one nonfavorite requests contend for a memory module, a random selection is made with an equal probability. A request from $P_i$ is granted only when there is no nonfavorite request. As a result the rate of request at a memory module due to its favorite processor is :

$$p_{fn} = p_0 \, m (1 - p_0 \frac{1-m}{N-1})^{N-1} . \tag{4}$$

The subscript $n$ stands for priority to nonfavorite requests. With $p_0 = 1$ and $m = 0.8$ , $\lim_{N \to \infty} p_{fn} = 0.655$. The rate of request at $MM_i$ due to other processors is

$$p_{nn} = p_f - p_0 \, m (1 - p_0 \frac{1-m}{N-1})^{N-1} . \tag{5}$$

With $p_0 = 1$ and $m = 0.8$ , $\lim_{N \to \infty} p_{nn} = 0.181$. The *Probability of Acceptance (PA)* of a request is the ratio of the average number of requests accepted to the number of requests generated per cycle. Thus

$$PA = \frac{BW}{p_0 * number \ of \ processors} = \frac{p_f}{p_0} . \tag{6}$$

We will define the *Probability of Favorite Acceptance (PFA)* as the ratio of the average number of favorite requests accepted to the number of favorite requests generated per cycle,

$$PFA = \frac{p_{fn} \ (or \ p_{fe} \ )}{p_0 . m} . \tag{7}$$

For the same example with $p_0 = 1$ and $m = 0.8$,

$$\lim_{N \to \infty} PFA = 0.906 \quad for \ EA \ policy.$$

$$= 0.819 \quad for \ priority \ to \ Nonfavorite \ requests.$$

Thus the *Probability of Non-favorite Acceptance* ,

$$PNFA = \frac{p_f - p_{fn} \ (or \ p_{fe} \ )}{p_0 \cdot (1-m)} \tag{8}$$

With $p_0 = 1$ and $m = 0.8$,

$$\lim_{N \to \infty} PNFA = 0.555 \quad for \ EA \ case.$$

$$= 0.905 \quad for \ priority \ to \ nonfavorite \ requests.$$

Hence by adopting priority to nonfavorite requests there is a remarkable improvement in $PNFA$ compared to the corresponding degradation in $PFA$ . The relationship between these probabilities is given by

$$PA = m \ . PFA + (1 - m) \ .PNFA \ . \tag{9}$$

It is noticed from the above examples that by allowing a priority to the nonfavorite requests, the degradation in $PFA$ is not substantial. Since the $PA$ remains the same, the $PNFA$ is increased to a large extent and for this particular example it jumps from 0.555 to 0.905 as calculated earlier. However, if $m$ is low, there will be a substantial degradation in the $PFA$ with priority to nonfavorite requests. Then $EA$ policy should be adopted. If priority is assigned to the favorite request, $PFA$ is unity for all values of $N$ and hence, the case is not considered.

## 3. ANALYSIS OF MULTIPLE-BUS

The BW analysis of multiple-bus architecture for the favorite memory case is given in [5]. There are $B$ buses in the system that connect $M$ processors to $N$ memories for $B \leq \min(M, N)$. A bus is connected to all the processors and memories. The arbiter (controller) cyclically allocates a bus to a memory that has an outstanding request. The BW of the multiple-bus structure is clearly a function of the number of buses $B$. When $B$ is equal to $min(M, N)$, the architecture has the same BW as that of a crossbar [5]. When the buses are insufficient, there will be bus conflicts in addition to the memory access conflicts. Hence, the multiple-bus controller can be designed as a cascade of a crossbar controller for resolving memory access conflicts and a bus controller to allocate buses to the memories. As a consequence, we can enumerate the following control policies.

1) Equal Acceptance - Equal Acceptance (EE)
2) Equal Acceptance - Nonfavorite (EN)
3) Nonfavorite - Equal Acceptance (NE)
4) Nonfavorite - Nonfavorite (NN)

For example, the EN policy states that EA rule is adopted for solving the memory access conflicts (crossbar controller) and Nonfavorite requests are given priority for the bus allocations. In case there are more than $B$ nonfavorite requests, $B$ of them are accepted with an equal probability. If there are less nonfavorite requests the extra buses are distributed to the favorite requests on an equally likely basis. This information is implicit in an arbiter policy and hence, is not explicitely mentioned in the above classification.

For simplicity, again, we will consider only $N \times N$ multiprocessor system. When the BW $BW_c$ of such a crossbar is less than or equal to $B$, the bus controller allows a bus to each of the selected requests and hence the system behaves as a crossbar. So for $BW_c \leq B$, the parameters derived in section II are true. For example, with $N$=16, $p_0 = 1$ and $m = 0.8$ , 14 buses are required for the multiple-bus architecture to have the same $BW$ as that of a crossbar. When $BW_c \geq B$, the bus controller plays a major role in deciding which of the requests should be allocated the buses. However, the bandwidth for the multiple-bus ($BW_m$) will remain equal to $B$ because of bus deficiency. The overall probability of acceptance is :

$$PA = \frac{Min(B, BW_c)}{p_0 . N} . \tag{10}$$

Hence, the following analyses are derived when $BW_m = B$.

(1) *Equal Acceptance - Equal Acceptance (EE) case*

The rate of request at a memory due to favorite requests is given by $p_{fe}$ in eq. (2). Hence $N.p_{fe}$ is the expected number of favorite requests out of $N.p_f$ requests accepted by the crossbar controller in total. With an EA policy at the bus controller, number of favorite requests allocated buses is $\frac{p_{fe}}{p_f} .B$. The total number of favorite requests generated at the processor side is $p_0 .m.N$. Hence,

$$PFA = \frac{p_{fe}}{p_f} \cdot \frac{B}{p_0 .m.N} . \tag{11}$$

And similarly

$$PNFA = \frac{p_{ne}}{p_f} \frac{B}{p_0 .(1-m)N} \tag{12}$$

where $p_f$ , $p_{fe}$ and $p_{ne}$ are given by equations 1, 2 and 3 respectively.

(2) *Equal Acceptance - Nonfavorite (EN) case*

Total number of nonfavorite requests accepted by crossbar controller is $p_{ne}.N$. They are all allocated buses subject to availability. Hence, the number of successful nonfavorite requests is $\min(p_{ne}.N, B)$. The number buses available for favorite requests is $B - min(p_{ne}.N, B)$
Then

$$PFA = \frac{B - min(p_{ne}.N, B)}{p_0.m.N} \qquad (13)$$

$$\text{and } PNFA = \frac{min(p_{ne}.N, B)}{p_0(1-m)N} \;. \qquad (14)$$

(3) *Nonfavorite - Equal Acceptance (NE) case*

All the nonfavorite requests are accepted by the crossbar controller. The rate of favorite request in this case is given by $p_{fn}$ in equation (4). With an equal acceptance of $p_{fn}.N$ favorite requests out of $p_f.N$ requests by the bus controller,

$$PFA = \frac{p_{fn}}{p_f} . \frac{B}{p_0.m.N} \;. \qquad (15)$$

The rate of request due to nonfavorite requests as selected by the crossbar controller is given by $p_{nn}$ in equation (5). With an EA policy by the bus controller,

$$PNFA = \frac{p_{nn}}{p_f} . \frac{B}{p_0.(1-m)N} \;. \qquad (16)$$

(4) *Nonfavorite - Nonfavorite (NN) case*

Here both the crossbar as well as the bus controllers give priority to nonfavorite requests while making selection. All the nonfavorite requests are allocated buses subject to availability and their number is given by $\min(p_{nn}.N, B)$. Rest of the buses are allocated to favorite requests. Hence,

$$PFA = \frac{B - \min(p_{nn}.N, B)}{p_0.m.N} \qquad (17)$$

$$\text{and } PNFA = \frac{\min(p_{nn}.N, B)}{p_0(1-m)N} \;. \qquad (18)$$

The $PFA$ and $PNFA$ for various selection policies, obtained for a $16 \times 16$ system, with $p_0 = 1$ and $m = 0.8$ , are plotted against the number of buses in Fig. 2 and 3 respectively. It can be observed in Fig. 2 that degradation in $PFA$ is not substantial by giving a priority to nonfavorite requests. Moreover, the $PFA$ linearly increases until it is saturated (after 14 buses) by the memory access conflicts. For $EN$ and $NN$ policies, buses are first allocated to nonfavorite requests. So the $PFA$ remains zero until more buses are available. In Fig. 3, the $PNFA$ increases linearly for $EE$ and $NE$ cases, but reaches saturation values quickly for other two policies. There is a substantial increase in $PNFA$ from 0.555 to 0.905 when priority is given to nonfavorite requests by the crossbar controller.

## 4. MULTISTAGE INTERCONNECTION NETWORKS (MIN's)

An MIN usually connects $N$ processors to $N$ memories through $\log_2 N$ stages of $2 \times 2$ switching elements (SE's). Each stage contains $\frac{N}{2}$ such SE's. Examples of such networks are Banyan, Omega, Cube and Baseline, etc. [6]. Although MIN's for $M \times N$, with $M \neq N$, systems have been proposed [8], we would limit our discussion to $N \times N$ systems for simplicity. We will assume that each processor can access its favorite memory through a straight connection of the switches on its path [7]. A $2 \times 2$ SE in an MIN may have a built-in priority structure to resolve the conflicts or it may choose one of the contending requests based on an EA policy. If a priority is given to straight connections, it may mean that more and more favorite requests will go through. However, nonfavorite requests are rejected most of the time, which should indeed be given priority over the favorite requests. We have analyzed the MIN performance for all the three different cases, namely : equal acceptance policy, priority to favorite requests and priority to nonfavorite requests. The analysis is an extension of our previous analysis for EA policy [7] and is not given here because of space restriction.

The BW obtained, with different priority assignment in a switch, are plotted in Fig. 4 for $p_0 = 1$ and $m = 0.8$ . The BW of an $MIN$ is affected by the switch arbitration policy unlike the crossbar or multiple-bus. When a priority is given to favorite requests, more and more requests are accepted giving rise to an increased BW. With a priority to nonfavorite requests more conflicts occur in the network and the BW reduces considerably. With decrease in the value of $m$ the BW is further reduced, but will be limited to an equally likely case. In an equally likely case, the priority assignment has no effect on the BW. The probability of acceptance (PA), probability of favorite acceptance (PFA) and probability of non favorite acceptance (PNFA) can be easily determined.

Unlike the crossbar or multiple-bus results, the PA of an MIN is different for different arbitration policies. This is evident from Fig. 4 because $PA = \frac{BW_0}{p_0.N}$ . The $PFA$ and $PNFA$ for different policies are drawn in Fig. 5 and 6 respectively against the size of the $MIN$ . As the size of the network increases, the probabilities reduce because of more and more conflicts. One can again observe that in a $1024 \times 1024$ network, with $p_0 = 1$ and $m = 0.8$ , there is about 40 % degradation in $PFA$ by adopting priority to nonfavorite requests (cross connection) compared to an $EA$ policy. However, the corresponding $PNFA$ increases by about 9 times in Fig. 6. The $PNFA$ is almost zero for large networks when priority is given to favorite (straight) connections.

## 5. CONCLUSION

The paper described the effect of various selection policies on the performance of crossbar, multiple-bus and MIN. The BW of crossbar and multiple-bus networks does not depend on the selection policy, but the probabilities of individual request acceptance do. It was shown that by giving a priority to remote or less frequent requests, the probability of their acceptance is dramatically improved. Unlike the above two networks, the selection policy in a $2 \times 2$ switch does affect the overall BW of an MIN. Although the BW decreases by giving priority to nonfavorite requests, it is an advisable policy because of the tremendous increase in PNFA. Overall, the paper described some interesting phenomena that were neglected before and are so practical for IN designs.

## 6. REFERENCES

[1] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing," Mc-Graw Hill, 1984.

[2] W. A. Wulf and C. G. Bell, "C.mmp- A Multi-mini processor," Proc. AFIPS Conf., Fall 1972.

[3] T. Lang et. al., "Bandwidth of Crossbar and Multibus connections for Multiprocessors," IEEE Trans. on Computers, Dec. 1982, pp. 1227-1234.

[4] T. Mudge et. al., "Analysis of Multiple-bus Interconnection Networks," IEEE Tran. on Computer, Vol. C-34, No. 10, Oct. 1985, pp.934-942.

[5] C. R. Das and L. N. Bhuyan, "Bandwidth Availability of Multiple-bus Multiprocessors," IEEE Trans. on Computers, Special Issue on Parallel Processing, Oct. 1985, pp. 918-926.

[6] T. Y. Feng, "A Survey of Interconnection Networks," IEEE Computer, Dec. 1981, pp. 12-27.

[7] L. N. Bhuyan, "An Analysis of Processor-Memory Interconnection Networks," IEEE Trans. on Computers, March 1985, pp. 279-283.

[8] L. N. Bhuyan and D. P. Agrawal, "Design and Performance of Generalized Interconnection Networks," IEEE Tran. on Comput., Dec. 1983, pp.1081-1090.

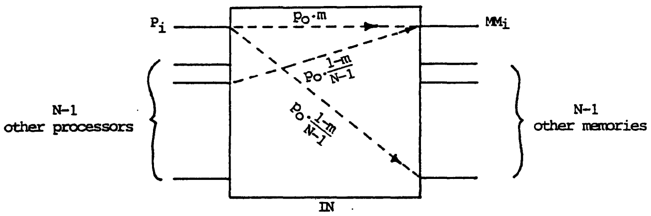


Fig. 1. N*N crossbar system with favorite memories.

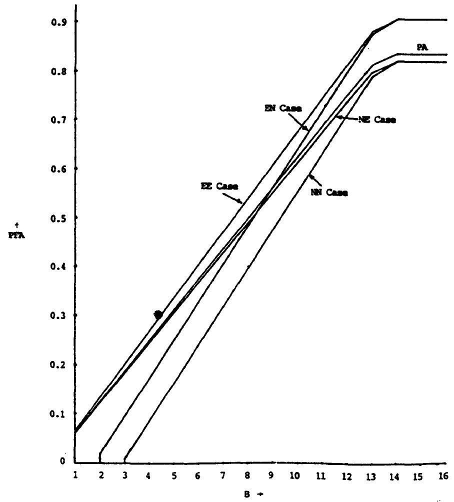


Fig. 2 PFA in a multiple-bus architecture with different selection policies.

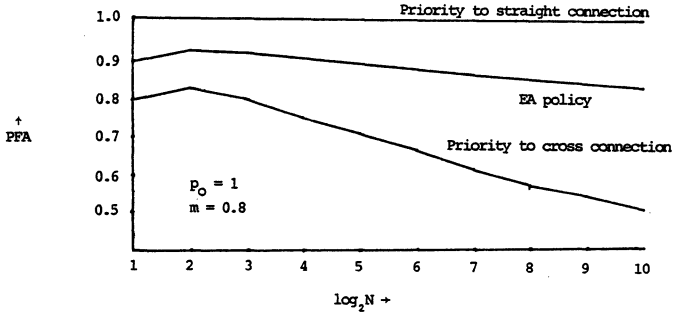


Fig. 5 PFA in an MIN with different selection policies.

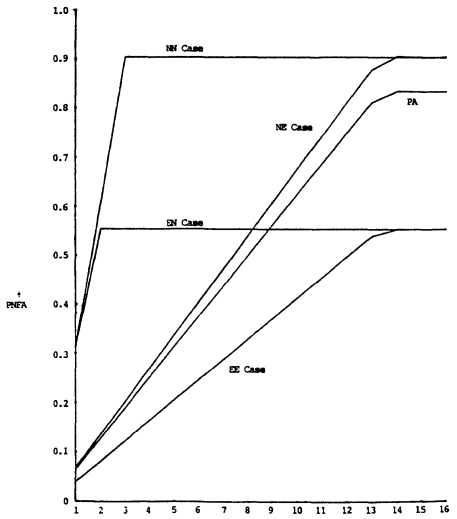


Fig. 3 PNFA in multiple-bus architecture with different selection policies.

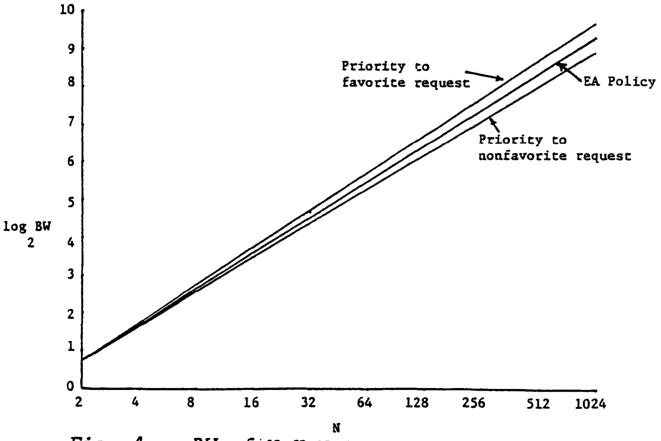


Fig. 4. BW of N X N Omega networks with different priority assignments.

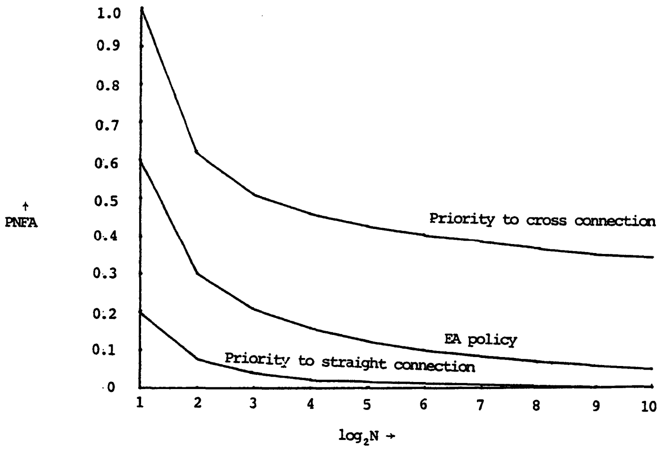


Fig. 6 PNFA in an MIN with different selection policies.

# EQUIVALENCE BETWEEN CUBE-CONNECTED CYCLES NETWORKS AND CIRCULAR SHUFFLE NETWORKS

*Bijendra N. Jain* †
Department of Computer Science
University of Maryland Baltimore County
Catonsville, Md 21228

*Satish K. Tripathi*
Institute for Advanced Computer Studies, and
Department of Computer Science
University of Maryland
College Park, Md 20742

## ABSTRACT

This paper is concerned with demonstrating the topological equivalence between two classes of computer architectures that support parallel computation, viz. Cube-Connected Cycles network (CCC) and Homogeneous Circular Shuffle Network (HCSN). The latter is based on the Perfect Shuffle Connection. By developing a suitable and common notation for addressing processing elements and specifying interconnections in the two networks, it is shown that these are topologically equivalent. The implications of such an equivalence are described. Known properties and algorithms about HCSN networks, in respect of routing, and fault tolerance, thereby, immediately become applicable to CCC networks. It is also shown that a large class of algorithms that run on a CCC network can also be implemented, with slight modification, on an HCSN network.

## 1. INTRODUCTION

Within the context of parallel computation a number of computer architectures have been proposed. These include systolic arrays, associative processors, vector processors, SIMD machines with or without shared memory, and MIMD machines. Our interest here is with SIMD architectures that do not share memory. An SIMD machine consists of a number of identical Processing Elements (PEs) each with its own local memory. The PEs communicate with each other through an interconnection network. A variety of such networks have been proposed, and parallel algorithms to solve various problems on them have been developed. Hypercube, Mesh-Connected, Cube-Connected Cycles (CCC), Perfect Shuffle Connection (PSC) networks (see Figures 1, 2) are some of the networks that have been extensively studied (see references [1]-[3]). More recently, a Homogeneous Circular Shuffle Network (HCSN) was proposed and studied (see reference [4]). This network has PSC as its basis (see Figure 4).

This paper demonstrates topological equivalence between the CCC network and the HCSN network. We develop a notation for addressing PEs in each of the networks and for specifying interconnections between PEs (see Section 3). This topological equivalence has a number of implications for both HCSN networks as well as CCC networks. These relate to routing algorithms, fault tolerance and VLSI layout (see Section 4). It is also shown that the equivalence is more than simply topological, in that the algorithms that run on an HCSN network also run on a CCC network. However, algorithms that run on a CCC network may require minor modifications when implemented on an HCSN network.

## 2. THE CCC AND HCSN NETWORKS

A CCC is a network of identical PEs, where each PE has three interconnection ports. Each interconnection linking two PEs may be used for bi-directional transmission of operands. The CCC network has $N=2^k$ PEs, where $1 \leq k \leq r+2^r$, $r \geq 1$, and $r$ is the smallest integer. Here each PE is addressed as $m$, $0 \leq m < N$. A PE with an address $m$ is alternatively represented as a tuple $(l,p)$, such that $m=l*2^r+p$, $0 \leq p < 2^r$, $0 \leq l < 2^{k-r}$. $l$ and $p$ have $k-r$ bit and $r$ bit representations, respectively. The interconnection between the PEs is described as follows: each PE has three ports, namely $F$, $B$, and $L$. Thus $F(l,p)$ is the $F$ port of a PE numbered $(l,p)$, etc. The PE with address $(l,p)$ is connected to the three other PEs as follows:

$$(B(l,p),F(l,(p-1) mod\, 2^r)),\ 0 \leq l < 2^{k-r},\ 0 \leq p < 2^r \qquad (1(a))$$

$$(L(l,p),L(l+e*2^p,p)), \qquad (1(b))$$

where $e=1-2*bit_p(l)$, and $bit_p(l)$ is the $p$-th bit of $l$. The notation used is as follows: $(P,Q)$ indicates a bi-directional communication link between the ports $P$, and $Q$ of some PEs, while $<P,Q>$ indicates a uni-directional link from port $P$ to port $Q$. (Note: The connection from $F(l,p)$ to $B(l,(p+1) mod\, 2^r)$ is implied by (1(a)), above.) See Figure 2 for an example of a CCC network where $N=32$, $k=5$, and $r=2$. Preparata, et al. (see reference [2]) have proposed a VLSI layout for CCC networks. See Figure 3 for the layout of a CCC network where $N=64$, $k=6$, and $r=2$. Looking at the layout it is reasonable to generalize the CCC network by dropping the requirement that $N=2^k$. Instead, we assume $N=n \log n$ where $n$ is a power of 2, and $n \geq 2$. As a consequence, each PE in a CCC is numbered $(l,p)$, $0 \leq l < n$, $0 \leq p < \log n$, and the interconnections are

$$(B(l,p),F(l,(p-1) mod \log n)) \qquad (2(a))$$

$$(L(l,p),L(l+e*2^p,p)) \qquad (2(b))$$

where $e=1-2*bit_p(l)$.

The Perfect Shuffle Connection (PSC) is a network of $n$ PEs, where $n$ is a power of 2 and $n \geq 2$. Each PE has three ports $O$, $I$, and $L$. The PEs are numbered as $l$, $0 \leq l < n$. The interconnection is described as:

$$<O(l),I(2*l\ mod(n-1))>,\ 0 \leq l < n-1, \qquad (3(a))$$

† On leave from Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi

0190-3918/86/0000/0008 $01.00 © 1986 IEEE

$$<O(n-1), I(n-1)> \qquad (3(b))$$

$$(L(l), L(l+1)),\ l\ mod\ 2=0,\ 0 \leq l < n. \qquad (3(c))$$

Based upon the PSC network, Tripathi and Huang (see reference [4]) have proposed a Homogeneous Circular Shuffle Network (HCSN). An HCSN network, parameterized by two numbers $(b, r)$, has $r$ columns with $b^{r-1}$ processors in each column, with a total of $r*b^{r-1}$ processors. We shall restrict $b=2$. Processors in an HCSN network are homogeneous with 2 input ports and 2 output ports. The interconnection pattern is a 2-shuffle connection described below in terms of addresses associated with input (and output) ports.

For a given column, the column-wide address of an input port (or an output port) is $x$, $0 \leq x < 2^r$, and given by its representation $(x_{r-1}, ..., x_0)$. The network-wide address of an input port (or an output port) $x$ in column $i$, $0 \leq i < r$, is $(a_{r-1}, ..., \mid a_i, \ldots, a_0)$, obtained by circularly left-shifting the column-address until the last tuple, $x_0$, is shifted into position $i$. The symbol "|" is put before $a_i$ to indicate the column of the input port (or the output port). Since, for various values of $a_i$, the notation refers to the two input ports (or output ports) of the same processor, the network-wide address of a processor in column $i$ is given by $(a_{r-1}, ..., a_{i+1}, \mid, a_{i-1}, ..., a_0)$. As an example, if the network-wide address of a processor $S$ is $(0, \mid, 1, 1)$, then its two input ports have addresses $I(0, \mid 0,1,1)$ and $I(0, \mid 1,1,1)$, while its output ports are addressed $O(0, \mid 0,1,1)$ and $O(0, \mid 1,1,1)$. See Figure 4 for an HCSN network where $b=2$, and $r=4$. The interconnection in an HCSN network may now be specified:

$$\left\langle O(a_{r-1}, ..., a_{i+1}, \mid a_i, a_{i-1}, ..., a_0), I(a_{r-1}, ..., a_{i+1}, a_i, \mid a_{i-1}, ..., a_0)\right\rangle. \qquad (4)$$

## 3. TOPOLOGICAL EQUIVALENCE

Before we demonstrate the topological equivalence between the CCC network with $n \log n$ PEs and the HCSN network, where $r = \log n$, the following comments are made:

(1) All interconnection links in a CCC network are bi-directional, while in an HCSN network the links are uni-directional, that is data may flow along a link from an output port of a processor to an input port of some processor.

(2) The notion of PEs in a CCC network and that of a processor in an HCSN network is somewhat different. In an HCSN network a processor may operate on two data objects that it receives at its input ports, and upon processing, produces two data objects at its two output ports. On the other hand, in a CCC network two PEs that are directly connected may communicate data with each other, and any processing on these data objects may be performed in either (or both) of the PEs.

To reconcile the two notions, we may take either of the approaches given below:

(A) Split each processor in an HCSN network into two PEs, each being associated with a pair of input port and output port. Further, the two PEs are connected to each other by a bi-directional link (see Figure 5(a)).

(B) Combine two neighboring PEs in a CCC network into one processor capable of operating on the data objects within the PEs (see Figure 5(b)).

We shall take the former approach. The resulting view of the HCSN network of Figure 4 is given in Figure 6. As such an HCSN network has $r = \log n$ columns, each containing $n = 2^r$ PEs. Thus an HCSN network has a total of $n \log n$ PEs. Further, it is easy to see that an HCSN network is an unfolding of a PSC network with $n$ PEs, the unfolding is to the extent of $\log n$ stages. The last stage is circularly connected to the first. The addresses associated with the PEs thus obtained from splitting processors in an HCSN network may now be obtained from the network-wide address of the corresponding input port (or output port). To show topological equivalence between a CCC network and an HCSN network, we rewrite the address of PEs in the HCSN network as $((a_{r-1}, ..., a_i, ..., a_0), c)$, where $c$ is the column number, and is equal to the index where the symbol "|" appeared in the original network-wide address. Each PE in an HCSN network has three ports, viz. $I$, $O$, and $L$. $I$ is an input port, $O$ is an output port, and $L$ is bi-directional. With this the interconnections in the HCSN network are described as:

$$<O(a, c), I(a, (c-1) mod \log n)>, \qquad (5(a))$$

$$(L(a, c), L(a + e * 2^c, c)), \qquad (5(b))$$

where $e = 1 - 2*bit_c(a)$.

The equivalence between the CCC network and the HCSN network is now evident. The CCC network and the HCSN network each have $n \log n$ PEs. That is, in the HCSN network there are $n$ PEs in each of the $\log n$ columns, while in a CCC network there is a hypercube of $n$ cycles each containing $\log n$ PEs. The processors in them are numbered as $(l, p)$, $0 \leq l < n$, $0 \leq p < \log n$. The interconnections are identical (see interconnections (2) and (5), above). The only difference is that in an HCSN network the interconnection from an output port to an input port of a PE is uni-directional.

## 4. IMPLICATIONS OF EQUIVALENCE

There are a number of implications of the above topological equivalence. First, a VLSI layout for an HCSN network is now evident. In fact, the layout of Figure 3 is a layout for the HCSN network of Figure 6, or equivalently that of Figure 4. The layout has an area $n*(n-1)$, or more precisely $O(N^2/\log^2 N)$. If the known layout for CCC networks is of minimum area, then the layout for HCSN networks is also optimal.

From the layout of Figure 3 it is evident that an HCSN network, and similarly a CCC network, has a recursive definition for certain values of n. Further, an HCSN network is symmetric not only with respect to a column (because of circular connection), but is also symmetric with respect to a PE within a column. The latter is a consequence of the fact that in a CCC network any cycle of PEs may be assumed to be at the origin of the hypercube. This symmetry allows one to make statements about the HCSN network assuming that a PE has an address (0,0) "without loss of generality", and enables one to provide new and simpler proofs for known results for HCSN networks in respect of routing algorithms and fault-tolerance analysis.

Properties that have been developed for the HCSN network now readily become applicable for CCC networks. One set of results known for the HCSN network are in respect of message routing using control tags (see reference [4]). A control tag, which is part of the message header, is a sequence of control numbers. The first remaining control number is extracted from the control tag upon receipt by a PE, and is used to select the outgoing route or link. The rest of the control tag is used to guide the routing following this PE. When the control tag is empty, the message is at its destination. Algorithms can now be obtained for routing data through a CCC network using a scheme discussed above. Further, it can be shown that the length of a control tag is no more than $2*\log n$ in a CCC network with $n \log n$ PEs. Also, results in respect of fault-tolerance of HCSN networks can be applied to CCC networks.

In view of the topological equivalence, an HCSN network may be treated as an implementation of a CCC network (provided bi-directional communication between $B$ and $F$ ports of a CCC network is not insisted upon), or vice versa. And, since a Perfect Shuffle Connection (PSC) network can always simulate an HCSN network, it is reasonable to expect that most algorithms for CCC networks are similar to those for PSC networks or even for Hypercube networks (see references [1], [2], and [5]).

The equivalence between HCSN networks and CCC networks is more than simply topological. For sure, any algorithm that runs on an HCSN network may be made to run on a corresponding CCC network. The reverse is, however, not true. The reason why such functional equivalence cannot be established is that, in a CCC network, transmission of data between PEs within a cycle is instead bi-directional. The latter implies a capability of processing data objects contained in neighboring PEs within a cycle.

Preparata, et al. (see reference [2]) have presented a generic DESCENT algorithm that is useful in solving a large class of problems, including Bitonic Merge, FFT, Sorting, Matrix operations. This version of the algorithm for CCC networks is immediately applicable to HCSN networks. A difficulty, however, does arise in implementing the operations on data objects within PEs of a cycle, viz. LOOPOPER(.). An alternative implementation of LOOPOPER for HCSN networks can be obtained. This algorithm assumes that transmission is uni-directional, but that each PE is capable of storing log$n$ data objects. The time complexity of the DESCENT algorithm running on an HCSN network can be shown to be the same as that when it runs on a CCC network. The consequence of this is that many algorithms available for CCC networks can now be suitably recoded for implementation on HCSN networks.

## REFERENCES

[1] Preparata, F.P., and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation", *Comm. of ACM,* Vol. 24, No. 5, pp. 300-309, May 1981.

[2] Stone, H.S., "Parallel Processing with the Perfect Shuffle", *IEEE Trans. on Computers,* Vol. C-20, No. 2, pp.153-161, Feb. 1971.

[3] Thompson, C.D., and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer", *Comm. of ACM,* Vol. 20, No. 4, pp. 263-271, Apr. 1977.

[4] Tripathi, S.K., S.T. Huang, *Circular Shuffle Network and Distributed Resource Scheduling,* Tech. Rep. TR-1406 (Revised), Univ of Maryland, Dept. of Comp. Sc., College Park, Md., July 1985.

[5] Dekel, E., D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms", *SIAM J. Computing,* Vol. 10, No. 4, pp. 657-675, Nov. 1981.

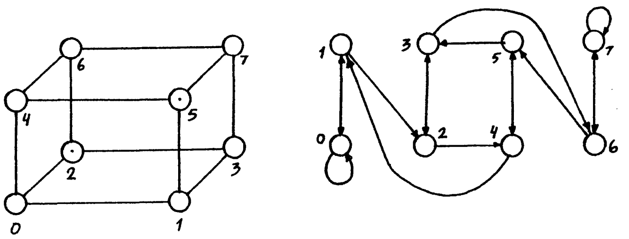


**Figure 1** The Hypercube and PSC networks.

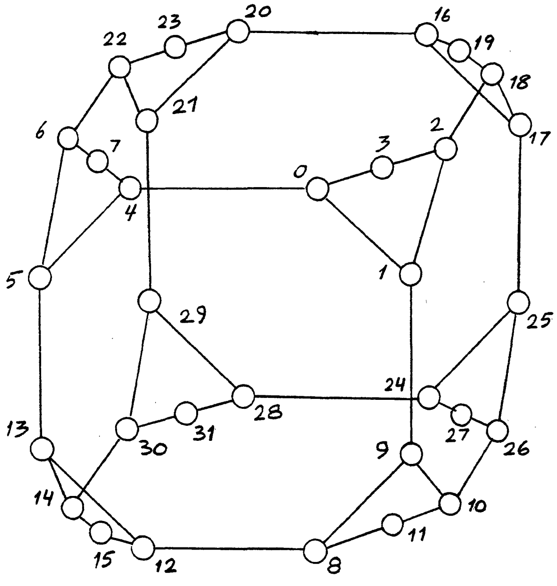


**Figure 2** A CCC network.

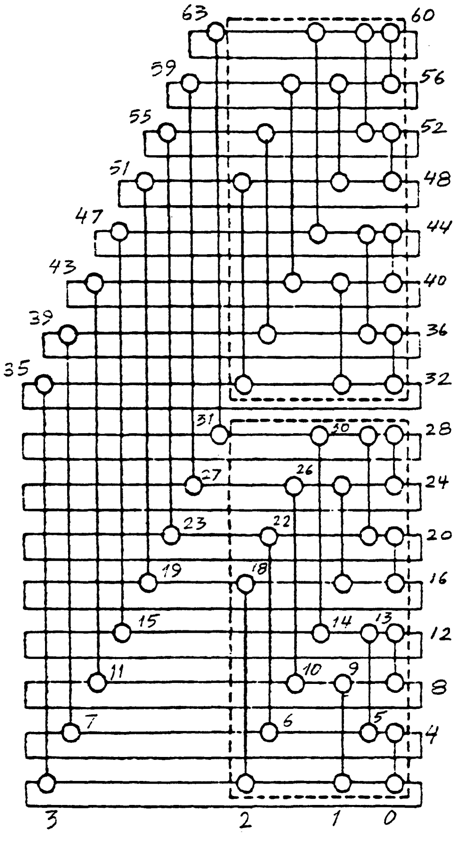


Figure 3 Layout for a CCC network.

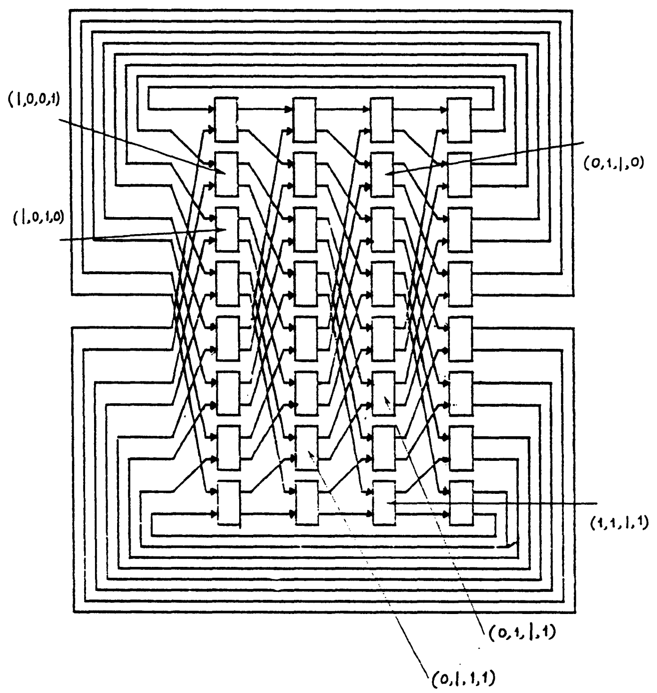


Figure 4 An HCSN network.

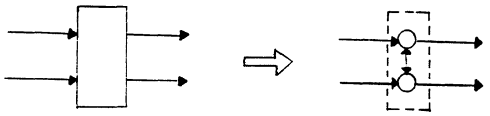


Figure 5(a) Splitting a processor into two interconnected PEs.

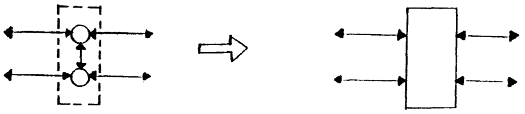


Figure 5(b) Merging of two interconnected PEs into a processor.

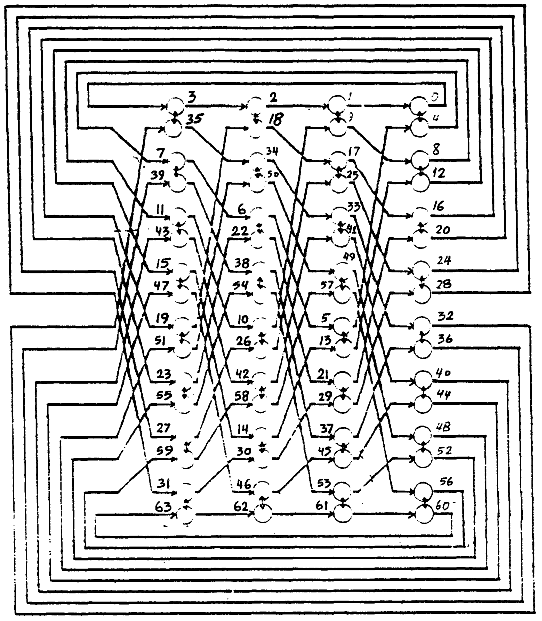


Figure 6 The HCSN network of Figure 3 with each processor split into two PEs.

# THE DISTRIBUTION OF WAITING TIMES IN CLOCKED MULTISTAGE INTERCONNECTION NETWORKS

**Clyde P. Kruskal**
Department of Computer Science
University of Maryland
College Park, Maryland 20742

**Marc Snir**
Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

**Alan Weiss**
AT&T Bell Laboratories, 2C118
600 Mountain Avenue
Murray Hill, NJ 07974

**ABSTRACT** We analyze the random delay experienced by a message traversing a buffered, multistage packet-switching banyan network. We find the generating function for the distribution of waiting time at the first stage of the network for a very general class of traffic, assuming messages have discrete sizes. For example, traffic can be uniform or nonuniform, messages can have different sizes, and messages can arrive in batches. For light to moderate loads, we conjecture that delays experienced at the various stages of the network are nearly the same and are nearly independent. This allows us to approximate the total delay distribution. Better approximations for the distribution of waiting times at later stages of the network are attained by assuming that in the limit a sort of spatial steady state is achieved. Extensive simulations confirm the formulas and conjectures.

## 1. INTRODUCTION

Buffered interconnection networks are receiving increasing consideration for use in parallel computers. They are integral components of several machines currently under development, including the Cedar machine at the University of Illinois [7], the NYU Ultracomputer at New York University [9], and the RP3 machine [17] at IBM, where they are used to interconnect processors to shared memory. In order to study the multitude of options available in actually building a machine, it is extremely useful to have formulas that approximate the performance of an interconnection network. In fact, formulas derived in a previous paper by two of the authors [12] have been heavily used in designing both the NYU Ultracomputer [9] and RP3 [15]. While simulation results are often more accurate, they are time consuming and expensive. In this paper, we analyze the random delay experienced by a message traversing a buffered, multistage packet-switching banyan network, for a very general class of traffic. For example, traffic can be uniform or nonuniform, messages can have different sizes, and messages can arrive in batches.

Interconnection networks connect processing elements to memory modules through stages of switches (Figure 1). Early work in describing these networks was done by Goke and Lipovski [8], Lawrie [14], and Patel [16], among others. For more full explanations of interconnection networks see [6] or [18], for example. There have been a number of performance analyses of interconnection networks (e.g. [4,5,12,13]).

The basic building block of an interconnection network is a $k$-input, $s$-output ($k \times s$) buffered switch (Figure 2). Each input port can accept one packet per clock cycle, and route it to the appropriate output port. Each output port has a FIFO buffer. Conflicts between messages simultaneously routed to the same output port are resolved by queueing the messages. We idealize this structure by assuming that the output buffers have infinite length. While this is clearly infeasible in practice, it is well known that, for light to moderate loads, moderate sized buffers provide approximately the same performance as infinite buffers. We also assume that each output port buffer can accept any number of messages from the input ports in a clock cycle, and that arriving messages do not interfere with departing messages. Each output port can be viewed as a discrete queueing system.

We make the following probabilistic assumptions concerning traffic at the first stage of a network:

(1) The number of messages arriving at successive cycles to an output port are independent, identically distributed random variables. These random variables may have a different distribution at different ports, and are clearly dependent from port to port.

(2) The service requirements (the number of cycles required to forward a packet) for successive messages at an output port are independent, identically distributed random variables. This distribution may vary from port to port.

Constant service time is usually the appropriate assumption for interconnection networks realized with synchronous logic.

Assuming that the traffic is uniform (e.g. each request is equally likely to go to each output node) and that at each cycle a packet arrives at an input node with a fixed probability $p$, the expected delay has been computed [12]. This analysis is based on Little's identity, and it is not obvious that it can be extended to obtain more information about the delay distribution, such as the variance. Such information is quite important for two reasons: First, to obtain good performance on a parallel machine, it is not sufficient to have a low expected memory access time; high variance will impede performance, as it is often the case that the speed of the slowest processor dictates the system speed. Second, as we discuss in Section 5, the variance can be used to obtain an approximate formula for the waiting time distribution of a message through the entire network.

Exact formulas for both the average and variance of the waiting time at the first stage when all messages take a single cycle to service were obtained in a previous paper [13]. This was used to obtain approximate formulas for longer messages of constant size. Using stronger methods we obtain in this paper exact formulas for the average and variance of the waiting time at the first stage for long messages of constant size; in fact, we obtain the entire distribution of waiting times for any discrete service time distribution. In the previous paper [13], we also suggested a method for analyzing the waiting time at later stages of the network, by assuming that the output of a queue can be modeled by a Markov process; the approximations were in practice hard to obtain and not very accurate. This paper gives an alternative method for approximating the waiting time at later stages; it is easy to use and provides extremely good approximations, as evidenced by a comparison to simulation results.

In Section 2, we analyze the performance of the first stage of an interconnection network, by calculating the $z$-transform of the distribution of waiting times. This enables us to compute higher moments of that distribution. In particular we present explicit formulas for the expected value and variance. In Section 3, the formulas are used to find the expected value and variance of the waiting time under various

Part of the work was done while the first author was at the University of Illinois. The first author was supported in part by a grant from IBM.

standard assumptions. For light to moderate loads, we conjecture that waiting time experienced at the the later stages of the network are nearly the same as for the first stage. In Section 4, we obtain better approximations for the waiting time at the later stages. In Section 5, we discuss how to analyze the total delay through the network. Section 6 gives some concluding remarks.

## 2. ANALYSIS

Our model for the first stage of switching comes under the general rubric of a discrete time queueing system. We compute in this section the $z$-transform for the waiting time, and use it to derive the expectation and variance of the waiting time, for general discrete service and arrival distributions. The solution method we use was indicated by Kobayashi and Konheim [11]. We are not, however, aware of a complete solution to this problem in the literature. As will be seen in the remainder of this paper, for the queueing systems we are interested in, it is useful to carry out the calculations in their entirety.

We start with some definitions. Let $\lambda$ be the average number of arrivals at any cycle and $m$ be the average service time of a message. The *traffic intensity* is then $\rho = m\lambda$.

Let $f_j$ be the probability that $j$ messages arrive at any cycle, and

$$R(z) = \sum_{j=0}^{\infty} f_j z^j .$$

Then

$$R'(1) = \lambda .$$

Let $g_j$ be the probability that a message requires $j$ time units to serve and

$$U(z) = \sum_{j=0}^{\infty} g_j z^j .$$

Then

$$U'(1) = m .$$

THEOREM 1: Let $w$ be the steady state waiting time for a message. The $z$-transform of the waiting time distribution of an output queue at the first stage is

$$\begin{aligned} t(z) &= \mathrm{E}(z^w) \\ &= \frac{1-m\lambda}{\lambda}\,\frac{1-z}{R(U(z))-z}\,\frac{1-R(U(z))}{1-U(z)} . \end{aligned} \qquad (1)$$

PROOF: Let $s_n$ be the unfinished work at the end of the $n$th cycle, $a_n$ number of messages arriving at the $n$th cycle, and $c_n$ be the total service time for messages arriving at the $n$th cycle. Let $s$, $a$, and $c$ be the steady state variables corresponding to $s_n$, $a_n$, and $c_n$. Note that

$$\mathrm{E}(z^a) = \mathrm{E}(z^{a_n}) = R(z) ,$$

$$\mathrm{E}(z^c) = \sum_{j=0}^{\infty} \mathrm{E}(z^c | a=j) f_j = \sum_{j=0}^{\infty} (U(z))^j f_j = R(U(z)) ,$$

and

$$s_n = \max(0, s_{n-1}+c_n-1) .$$

Since $c_n$ is independent of $s_{n-1}$ we obtain the identity

$$\begin{aligned} \mathrm{E}(z^s) &= \mathrm{E}(z^{s+c-1} \mid s>0)\mathrm{P}(s>0) \\ &\quad + \mathrm{E}(z^{c-1} \mid s=0, c>0)\mathrm{P}(s=0, c>0) \\ &\quad + \mathrm{E}(z^0 \mid s=0, c=0)\mathrm{P}(s=0, c=0) \\ &= \mathrm{E}(z^c)\mathrm{E}(z^{s-1} \mid s>0)\mathrm{P}(s>0) \\ &\quad + \mathrm{E}(z^{c-1} \mid c>0)\mathrm{P}(s=0)\mathrm{P}(c=0) \\ &\quad + \mathrm{P}(s=0)\mathrm{P}(c=0) . \end{aligned}$$

Let

$$\Psi(z) = \sum_{j=0}^{\infty} h_j z^j = \mathrm{E}(z^s) .$$

The previous identity implies

$$\Psi(z) = R(U(z))\frac{(\Psi(z)-h_0)}{z} + h_0\frac{R(U(z))-R(U(0))}{z} + h_0 R(U(0)) ,$$

so that

$$\Psi(z) = \frac{h_0(1-z)R(U(0))}{R(U(z))-z} .$$

We compute, using L'Hospital's rule,

$$\Psi(1) = 1 = \frac{h_0 R(U(0))}{1-m\lambda} ,$$

so that

$$h_0 = \frac{1-m\lambda}{R(U(0))} ,$$

and

$$\Psi(z) = \frac{(1-z)(1-m\lambda)}{R(U(z))-z} .$$

Since the arrival process is memoryless, arriving batches see the steady state unfinished work distribution $s$. Thus the steady state waiting time for a message $w = s + w'$, where $w'$ is the steady state service time for messages arriving at the same cycle, but served first. We have

$$\mathrm{E}(z^w) = \mathrm{E}(z^s)\mathrm{E}(z^{w'}) = \Psi(z)\mathrm{E}(z^{w'}) .$$

Let $d$ be the steady state number of messages that arrive at the same cycle with any message, but are served before it; let $\phi(z) = \mathrm{E}(z^d)$. Then, using the same derivation as before, we get

$$\mathrm{E}(z^{w'}) = \phi(U(z)) .$$

We shall now compute $\phi(z)$. The probability that any message arrives in a batch of $j$ messages is equal to $j f_j / \lambda$. Thus

$$\begin{aligned} \mathrm{P}(d=j) &= \sum_{k=j+1}^{\infty} \mathrm{P}(d=j \mid \text{message arrives in a batch of } k \text{ messages}) \, k f_k / \lambda \\ &= \sum_{k=j+1}^{\infty} \frac{1}{k}(k f_k / \lambda) , \end{aligned}$$

so that

$$\begin{aligned} \phi(z) &= \frac{1}{\lambda}\sum_{j=0}^{\infty}\sum_{k=j+1}^{\infty} f_k z^j = \frac{1}{\lambda}\sum_{k=1}^{\infty} f_k \sum_{j=0}^{k-1} z^j \\ &= \frac{1}{\lambda(z-1)}\sum_{k=1}^{\infty} f_k (z^k-1) = \frac{R(z)-1}{\lambda(z-1)} . \end{aligned}$$

The $z$-transform of the waiting time distribution is

$$\begin{aligned} t(z) &= \mathrm{E}(z^w) = \Psi(z)\phi(U(z)) \\ &= \frac{1-m\lambda}{\lambda}\,\frac{1-z}{R(U(z))-z}\,\frac{1-R(U(z))}{1-U(z)} . \end{aligned} \qquad (1)$$

□

In principle this gives the complete distribution of the waiting time.

COROLLARY 2:

$$\mathrm{E}(w) = t'(1) = \frac{mR''(1) + \lambda^2 U''(1)}{2\lambda(1-m\lambda)} . \tag{2}$$

COROLLARY 3:

$$\begin{aligned}\mathrm{Var}(w) &= t''(1) + t'(1)-(t'(1))^2 \\ &= \frac{\begin{array}{c}(6m\lambda R''(1) + 4m^2\lambda R'''(1) + 6\lambda^3 U''(1) \\ + 4\lambda^3 U'''(1))\,(1-m\lambda) - 3m^2 R''(1)^2(1-2m\lambda) \\ + 3\lambda^4 U''(1)^2 + 6\lambda R''(1)U''(1)\end{array}}{12\lambda^2(1-m\lambda)^2} .\end{aligned} \tag{3}$$

(The derivation of $t''(1)$ used six applications of L'Hospital's rule, and took Macsyma all night on a minicomputer.)

## 3. EXAMPLES

We now apply the above formulas to derive the expected value and the variance of the waiting time for messages in several standard and important queueing systems. Note that the expected value formulas only give the waiting time of a message. To obtain the *delay* of a message in a queue, one must add to these formulas the service time. For the queueing systems in this section, message arrivals are independent of queue length. Thus, the variance of the delay of a message in a queue is simply the sum of the variance of the waiting time and the variance of the service time.

### 3.1. Service Time One

Suppose that all messages take exactly one unit of time to be serviced. Then $m = 1$ and $U(z) = z$. Thus,

$$U'(z) = 1 \quad \text{and} \quad U''(z) = U'''(z) = 0 .$$

Substituting into Eq. (1), we obtain

$$t(z) = \mathrm{E}(z^w) = \frac{1-\lambda}{\lambda}\,\frac{1-R(z)}{R(z)-z} .$$

Substituting into Eq. (2) we get

$$\mathrm{E}(w) = \frac{R''(1)}{2\lambda(1-\lambda)} \tag{4}$$

and substituting into Eq. (3) we get

$$\mathrm{Var}(w) = \frac{2(3R''(1) + 2R'''(1))\lambda(1-\lambda) - 3(1-2\lambda)(R''(1))^2}{12\lambda^2(1-\lambda)^2} \tag{5}$$

We analyze some special cases of this for $k$-input, $s$-output ($k \times s$) switches.

#### 3.1.1. Uniform Traffic, Single Arrivals

Suppose that each input port has a probability $p$ of receiving one message at each unit of time, and that each incoming message has an equal chance of going to any of the output ports. Then

$$f_j = \binom{k}{j}\left(\frac{p}{s}\right)^j \left(1 - \frac{p}{s}\right)^{k-j} .$$

This quickly yields

$$R(z) = \left(1 - \frac{p}{s} + \frac{pz}{s}\right)^k .$$

Calculating $R'(1)$, $R''(1)$, and $R'''(1)$ and substituting into Eqs. (4) and (5) yields

$$\mathrm{E}(w) = \frac{(1-\frac{1}{k})\lambda}{2\,(1-\lambda)} \tag{6}$$

and

$$\mathrm{Var}(w) = \frac{(1-\frac{1}{k})\,\lambda\,[6 - 5\lambda(1+\frac{1}{k}) + 2\lambda^2(1+\frac{1}{k})]}{12\,(1-\lambda)^2} . \tag{7}$$

#### 3.1.2. Bulk Arrivals

In many systems the size of a message exceeds the size of a transmission packet; a message is transmitted in several packets. These packets arrive at the first stage of the network in one bulk. This can be modeled as in the previous example, except arrivals at input ports are in batches. Assuming a constant batch size of $b$ messages,

$$R(z) = \left(1 - \frac{p}{s} + \frac{pz^b}{s}\right)^k .$$

Using Eqs. (4) and (5), this gives

$$\mathrm{E}(w) = \frac{(b-1) + (1-\frac{1}{k})\lambda}{2(1-\lambda)}$$

and

$$\mathrm{Var}(w) = \frac{b^2-1 + 2\lambda(b^2+2-\frac{3b}{k}) - 5\lambda^2(1-\frac{1}{k^2}) + 2\lambda^3(1-\frac{1}{k^2})}{12(1-\lambda)^2} .$$

These agree with our previous formulas for the case $b = 1$.

#### 3.1.3. Nonuniform Traffic

In many practical situations each input is likely to have a distinct favorite output port (e.g. the output port connecting a processor to its private memory — see [1]). We assume that $k = s$. (It is not hard to generalize this for $k \neq s$, but the equations become quite lengthy.) We do assume bulk arrivals. Each input port sends arriving messages to its favorite output port with probability $q$, and sends them with probability $(1-q)/k$ to each output port (including its favorite output). The distribution of messages at the output ports is the product of two terms: the first term accounts for normal messages and is essentially the same as given in 3.1.2, with $p$ replaced by $p(1-q)$; the second term accounts for favored messages. We get

$$\begin{aligned}R(z) = {} & (1 - p\frac{1-q}{k} + p\frac{1-q}{k}z^b)^{k-1} \\ & (1 - p(q+\frac{1-q}{k}) + p(q+\frac{1-q}{k})z^b) .\end{aligned}$$

Substituting into Eq. (4),

$$\mathrm{E}(w) = \frac{\lambda(1 - q^2)(1 - \frac{1}{k}) + b - 1}{2\,(1 - \lambda)} .$$

Note that, for $q = 1$ we get $\mathrm{E}(w) = 0$, and for $q = 0$ we obtain the same formula as in 3.1.1 (with $k = s$), as it should be. The general formula for the variance is quite lengthy.

### 3.2. Constant Service Times

We now consider the situation when messages can have one of several constant service times. We will only consider uniform traffic and single arrivals. Thus, as in Section 3.1.1,

$$R(z) = \left(1 - \frac{p}{s} + \frac{pz}{s}\right)^k .$$

### 3.2.1. Single Size

First, suppose that each message takes exactly $m$ units of time to transmit. This will occur, for instance, when each message is composed of an equal number $(m)$ of packets, and the constituent packets of a message are transmitted at consecutive cycles. Then

$$U(z) = z^m .$$

The traffic intensity is now

$$\rho = \frac{mpk}{s} .$$

Substituting into Eqs. (2) and (3), we obtain

$$E(w) = \frac{\rho(m - \frac{1}{k})}{2(1-\rho)} \tag{8}$$

and

$$\mathrm{Var}(w) = \frac{\rho \left( \begin{array}{c} (1-\frac{1}{k}) \, [6m - 5\rho(1+\frac{1}{k}) + 2\rho^2(1+\frac{1}{k})] \\ + (m-1) \, [2(2m-1) - \rho(m+1)] \end{array} \right)}{12\,(1-\rho)^2} . \tag{9}$$

These coincide, for $m = 1$, with the equations of 3.1.1.

### 3.2.2. Multiple Sizes

Now suppose there are $n$ service times $m_1,\ldots,m_n$, and service time $m_i$ occurs with probability $g_i$. This will occur when there are different kinds of requests. For example, read requests are likely to have different sizes than write requests.

We get

$$U(z) = \sum_1^n g_i z^{m_i} .$$

Thus

$$\rho = \frac{kp}{s} \sum_1^n m_i g_i$$

Substituting into Eq. (2), we obtain

$$E(w) = \frac{\lambda \sum_1^n m_i (m_i - \frac{1}{k}) g_i}{2\,(1-\rho)} = \frac{\rho \sum_1^n m_i (m_i - \frac{1}{k}) g_i}{2\,(1-\rho) \sum_1^n m_i g_i} .$$

The formula for the variance could also be obtained, but it is quite lengthy and not particularly enlightening.

## 4. LATER STAGES

We do not know how to analyze the later stages exactly as the inputs at successive cycles are not independent. We have, however, developed some very useful approximate formulas for the average and variance of the waiting time. These are based on two observations: First, as we progress through the network, the waiting time statistics quickly approach a limiting distribution. Second, nearly every waiting time distribution in queueing theory has an average on the order of $1/(1-\rho)$ as $\rho$ tends to one; that is, if $w_i(\rho)$ is the average waiting time at the $i$ th stage, and $w_\infty(\rho)$ is the limit as $i$ gets large, we expect $\lim\limits_{\rho \to 1} (1-\rho) w_\infty(\rho)$ to exist. We calculated $w_1(\rho)$ exactly in Section 3, and we expect $w_1(\rho)$ and $w_\infty(\rho)$ to have similar qualitative properties, i.e., they should depend on most parameters in roughly the same way. Hence, it seems reasonable to estimate

$$r(\rho) \overset{\text{def}}{=} \frac{w_\infty(\rho)}{w_1(\rho)} .$$

It is clear that $r(0) = 1$. We use simulations to estimate $r(1/2)$, and then simply linearly interpolate to obtain to obtain a value $a$ such that

$$r(\rho) \approx 1 + a\rho . \tag{10}$$

Then

$$w_\infty(\rho) \approx (1 + a\rho)\, w_1(\rho) .$$

We will also generalize the formulas to take into account the dependence of $w_i(\rho)$ on the stage, the switch size, and the message size distribution. This method of interpolation was previously applied to queueing systems by Burman and Smith [3] using light and heavy traffic theory. The light traffic limit is obvious in our case. We do not have a heavy traffic analysis for our process, so we rely on simulation instead.

Using the same ideas, we can obtain an approximation for the variance. Let $v_i$ be the variance of the waiting time at stage $i$, and $v_\infty$ be the variance in the limit. Since the formulas for variance have one higher power of $\rho$, we expect a good approximation of $v_\infty(\rho)/v_1(\rho)$ to contain (at least) one higher power of $\rho$, i.e. we obtain a quadratic interpolation for the variance. The variance after several stages can be approximated by

$$v_\infty \approx (1 + a\rho + b\rho^2) v_1 ,$$

where $a$ and $b$ are constants to be determined.

In the remainder of this section, we obtain our expression for the waiting time and variance in step-by-step generalizations. We first estimate them for uniform traffic when messages have size one, then size $m$, and then general size. Finally, we consider nonuniform traffic.

### 4.1. Service Time One

Consider service time one, 2×2 switches, single arrivals, and uniform traffic. For $p = .5$, $w_1 = .25$ (Eq. (6)), and, from the simulations in Table I, $w_\infty$ seems to be about .3. Substituting into Eq. (10) and solving for $a$ gives $a \approx 2/5$. Thus, we find $r(\rho) \approx 1 + 2p/5$, so the waiting time

$$w_\infty \approx (1 + \frac{2p}{5}) \frac{p}{4(1-p)} .$$

Table I compares the simulation results with our formulas. The waiting time values in the ANALYSIS row are from the exact formula for the first stage (Eqs. (6) and (7)), and the waiting time values in the ESTIMATE row are from the above approximation for the waiting time in the limit. Note that the approximation seems to be slightly low for $p$ small and slightly high for $p$ large. More complete simulation results (not included for brevity) show that $r(\rho)$ is actually slightly concave. An even better estimate could be obtained by using a quadratic approximation.

Using the same technique for 4×4 and 8×8 switches gives $a$ a bit less than .2 and $a$ a bit less than .1, respectively (see Eq. (6) and Table II). This suggests that the above formula can be (crudely) extended to $k \times k$ switches by linearly including $k$ as a parameter. This gives the waiting time

$$w_\infty \approx (1 + \frac{4p}{5k}) \frac{(1 - \frac{1}{k})\,p}{2\,(1-p)} . \tag{11}$$

Table II compares the simulation results with our formulas.

In Tables I and II it looks as if $w_i$ approaches $w_\infty$ geometrically. This suggests a formula of the form $r_i(\rho) = 1 + (1-\alpha^{i-1})(r(\rho)-1)$ for some $\alpha<1$, yielding

$$w_i \approx (1 + \frac{4p}{5k}(1 - \alpha^{i-1})) \frac{(1 - \frac{1}{k}) p}{2(1-p)} \tag{12}$$

as the expected waiting time at the $i$ th stage. Looking once again at the formula for $k = 2$ and $p = .5$ (Table I), gives $\alpha = 2/5$ as a good approximation. It turns out that this value of $\alpha$ works reasonably well for general $k$ and $p$. For brevity, we do not explicitly compare this formula to the simulations (although the interested reader can easily do the calculations). It is by no means surprising that, for a given $p$ and $k$, $w_i$ approaches $w_\infty$ geometrically; what is perhaps surprising is that a single value of $\alpha$ works well for all $p$ and $k$.

Applying the same techniques to variance, we find that a reasonable formula for the variance after several stages is

$$v_\infty \approx (1 + \frac{p}{2k} + \frac{2p^2}{k}) \frac{(1-\frac{1}{k})\, p\ [6 - 5p\,(1+\frac{1}{k}) + 2p^2(1+\frac{1}{k})]}{12(1-p)^2} . \tag{13}$$

(Since this is only an approximation and since the simulation results do not give exact answers, there is quite a bit of freedom in choosing coefficients $a$ and $b$ for the $p$ and $p^2$ terms. Other choices will surely work just as well or better.) We can also estimate the variance at stage $i$ to be

$$v_i \approx \left(1 + (\frac{p}{2k} + \frac{2p^2}{k})(1-\alpha^{i-1})\right) \frac{(1-\frac{1}{k})\, p\ [6 - 5p\,(1+\frac{1}{k}) + 2p^2(1+\frac{1}{k})]}{12(1-p)^2} , \tag{14}$$

where $\alpha = 2/5$.

### 4.2. Single Service Time

Consider the case when messages have a single constant size. Our model of the first stage is not a particularly good model for the later stages: At the first stage a source after sending a message can send a new message on the next or any later cycle. At later stages, since sources are outputs from queues, once a source sends a message, that source will not send a message for at least $m$ cycles. This will tend to reduce queueing delays at the later stages.

Later stages can be better modeled by assuming that messages take one cycle to be processed, but the cycle time is $m$ times as long. Following [12], we use the formula for service time one (Eq. (11)), and, for a fixed $p$, multiply the time to process a message by a factor $m$, and also multiply the average number of packets per cycle by $m$. In other words, for a fixed traffic intensity $\rho$, the cycle time is $m$ times as large. This gives the average waiting time

$$w_\infty \approx (1 + \frac{4mp}{5k}) \frac{(1 - \frac{1}{k})\, m^2 p}{2(1 - mp)} . \tag{15}$$

For $m \geq 2$, this formula is a reasonable approximation at all stages after the first, and, of course, we have an exact formula for the first stage. Table III compares the simulation results with our formulas.

Let us examine the behavior of the interior stages in light traffic. If we allow $m$ to increase and $p$ to decrease with $mp = \rho$ constant, then in time scaled by $m$, the first stage output queues become M/D/1 queues with arrival rate $\rho$ and service time 1. (Actually, the well-known waiting time statistics of M/D/1 queues can be obtained as limits of (8) and (9).) Now the interior stages are not precisely M/D/1 queues in this scaling, because the packets output from previous stages must be spaced by at least $m$ time units. Nevertheless, it is clear that in light traffic the interior stages will resemble M/D/1 queues, but the congestion will be lower than at the first stage, since packets will be very unlikely to collide with other packets from the same source. That is, the congestion will be as if the arrival rate were $(1-1/k)\rho$. Using the M/D/1 light traffic results

$$\mathrm{E(w)} = \frac{(1-\frac{1}{k})\rho}{2} + O(\rho^2)$$

and

$$\mathrm{Var}(w) = \frac{(1-\frac{1}{k})\rho}{3} + O(\rho^2) .$$

Our approximations should have these properties, too. Eq. (15) does satisfy this.

To obtain an approximation for the variance, we argue as before: start with formula for the variance at the first stage for unit size messages (Eq. (7)), multiply by $m^2$, change $p$ to $mp$, and then use the light traffic analysis and the simulations to interpolate. Our heuristic formula is (see Eq. (13))

$$v_\infty \approx \left(\frac{2}{3} + \frac{C_1\rho}{k} + \frac{C_2\rho^2}{k}\right) \frac{(1-\frac{1}{k})\, m^3 p\ [6 - 5mp\,(1+\frac{1}{k}) + 2(mp)^2(1+\frac{1}{k})]}{12(1 - mp)^2} ,$$

where 2/3 was obtained from light traffic analysis. Light traffic analysis is a limiting case for $m$ large; in practice we found that 7/10 works better than 2/3 for small and moderate message sizes. We match the constants $C_1$ and $C_2$ to simulation results, giving

$$v_\infty \approx \frac{7}{10}(1 + \frac{2mp}{3k} + \frac{4(mp)^2}{k}) \frac{(1-\frac{1}{k})\, m^3 p\ [6 - 5mp\,(1+\frac{1}{k}) + 2(mp)^2(1+\frac{1}{k})]}{12(1 - mp)^2} . \tag{16}$$

This approximation is still slightly low for $m$ small, as can be seen in Table III. Better approximations can be obtained for each individual value of $m$; in particular, Eq. (13) is a much better approximation for $m = 1$. As with waiting times, for $m \geq 2$, this formula can be used to approximate variances for all stages after the first.

### 4.3. Multiple Service Times

As in Section 3.4.2, suppose there are $n$ service times $m_1,\ldots,m_n$, and service time $m_i$ occurs with probability $g_i$. The average service time is $m = \sum_{i=1}^{n} g_i m_i$. To obtain an approximate formula for the average waiting time, replace the size of all messages by their average size $m$ and use the approximate waiting time formula from the previous section (Eq. (15)). This gives the average waiting time

$$w_\infty \approx (1 + \frac{4mp}{5k}) \frac{(1 - \frac{1}{k})\, m^2 p}{2(1 - mp)} .$$

The values obtained from this formula tend to be a bit low. The reason is that we are approximating multiple size messages by their average size. Since we are able to calculate everything at the first stage exactly, we know how much off such an assumption would be at the first stage: simply the ratio of the actual expected waiting time and the waiting assuming all messages have their average size. Assuming this ratio is fairly constant at the different stages, multiplying the above formula by this ratio gives a very good approximation:

$$w_{\infty} \approx (1 + \frac{4mp}{5k}) \frac{(1 - \frac{1}{k})mp \sum_{i=1}^{n} m_i (m_i - \frac{1}{k}) g_i}{2(1 - mp)(m - \frac{1}{k})} .$$

An approximate formula for the variance $v_{\infty}$ could be obtained similarly, but, as with the variance formula for the first stage, it is quite lengthy. We have, however, obtained numerical values from both variance formulas, i.e. for $v_1$ and $v_{\infty}$. Table IV compares the simulation results with our formulas.

## 5. TOTAL DELAY

Once we have formulas for the expected value and variance of the waiting time at a stage, these can be used to obtain approximations for the total waiting time. The expected value of the total waiting time is simply the sum of the average waiting time at the different stages. In particular, for messages of size one, summing the $w_i$ in Eq. (12) approximates the total waiting time for an $n$ stage network as

$$n \left( 1 + \frac{4p}{5k}(1 - \frac{1 - \alpha^n}{n(1-\alpha)}) \right) \frac{(1 - \frac{1}{k})\, p}{2(1 - p)} ,$$

where $\alpha = 2/5$. For $m \geq 2$ the average total waiting time can be approximated as the average waiting time from the first stage (Eq. (8)) plus $n-1$ times the waiting time at the later stages (Eq. (15)), which is

$$\frac{(m - \frac{1}{k})mp}{2(1 - mp)} + (n-1)(1 + \frac{4mp}{5k}) \frac{(1 - \frac{1}{k})\, m^2 p}{2(1 - mp)} .$$

If the waiting times from stage to stage were independent, as is the case with Poisson arrivals and exponential service times, the variance of the total waiting time would simply be the sum of the variances at the different stages. Simulations show that waiting times at neighboring stages have fairly low correlation, and the correlation seems to drop geometrically as stages become further apart. Thus summing the variances should be a good approximation.

To obtain a better approximation, note that the total variance is actually the sum of the covariances between stages. Let $v_{ij}$ be the covariance between stage $i$ and stage $j$. Covariances seem to drop geometrically as stages become further apart. In particular, the $v_{ij}$ can be approximated as follows: $v_{ii} = v_i$, $v_{i,i+1} \approx av_i$, $v_{i,i+2} \approx abv_i$, $v_{i,i+3} \approx ab^2 v_i$, $v_{i,i+4} \approx ab^3 v_i$, ..., where $a = (1 - \frac{2mp}{5})\frac{3mp}{5k}$ and $b = (1 - \frac{2mp}{5})/k$. Now summing all of the covariances approximates the total variance as

$$\sum_{i=1}^{n} \left( 1 + \frac{2a(1 - b^{n-i})}{1-b} \right) v_i .$$

For $m = 1$, we use the $v_i$ from Eq. (14). For $m > 1$, $v_1$ is the true variance for the first stage (Eq. (9)), and $v_i$, $i > 1$, can be approximated by the formula for $v_{\infty}$ (Eq. (16)). Tables VI and VII compare the simulation results with our formulas.

The distribution of waiting times seems to be about the same for all stages. If the distributions were independent, then by the central limit theorem, the total waiting times for a large number of stages could be approximated by a (truncated) normal distribution, whose sum is the sum of the expected values and variance is the sum of the variances. The central limit theorem actually holds under much weaker hypotheses than independence (see, for example, [2]), and we expect it essentially to apply here. Now, typically in queueing systems, the distribution of waiting times has an exponential or geometric tail. Thus, for few stages, we expect a gamma distribution with the proper expected value and variance to be even a better approximation. Figures 3 and 4 show an incredibly good match between the gamma and the observed distributions, especially at the tails. The gamma distributions were formed with the means and variances given by the estimates from Tables VI and VII. In practice, these moments and the tail of the waiting time distribution are the quantities of interest; we believe our formulas are accurate enough for all practical purposes.

So far we have obtained formulas for the total waiting time. In order to obtain the *total delay* in the network one has to add to the total waiting time the total service time. If service time is one, then the total service time is simply the number of stages. In general it is the sum of service times at the successive stages.

The waiting time at one stage may depend on service time at a previous stage. However, the correlation is weak, so that these random variables are stochastically nearly independent. Thus, the variance of the total delay is approximately the variance of the total waiting time plus the sum of the variances of the service times. If the service times are constant, then their variances are zero, so the variance of the total delay is exactly the variance of the total waiting time. In general, the distribution of the total delay can easily be approximated by looking at the distributions of individual service times and the distribution of the total waiting time.

## 6. CONCLUSION

We have analyzed the delay experienced by a message in a buffered, multistage, packet-switching banyan network. For the first stage, we were able to derive the complete distribution of the delay for a very general class of distributions, assuming messages have discrete sizes. The result is quite general: for example, one can use it to derive the Pollaczek-Khinchin formulas for M/G/1 queues. The result was used to determine exactly the average and variance of the delay for several commonly considered distributions. Using the delay formulas for the first stage, we developed extremely good approximations for the average and variance of the delay at later stages. Finally, this allowed us to obtain good approximations for the full distribution of the total delay of a message through the entire network.

In order to approximate the delay after the first stage, it was essential to have good formulas for the delay at the first stage. It was only by building on them that we were able to make educated guesses as to the delays at later stages.

One aspect of our results that is worth stressing is the dependency of waiting time on the message size $m$: For a *fixed* traffic intensity $\rho$, the average waiting time increases linearly in $m$ (see Eqs. (8),(15)), and the variance increases quadratically in $m$ (see Eqs. (9),(16)). Thus, while using larger messages may save the overhead of duplicating the same routing information over several packets, it may dramatically increase delays in all but very lightly loaded networks. This point has already been made in [9,12,13], but does not seem to be widely appreciated.

While our simulations seem to indicate clearly that average waiting time at successive stages converge, it would be nice to be able to prove this result formally, i.e. to show that average delay at successive stages can be bounded, independent of the network size.

## ACKNOWLEDGEMENTS

The authors thank Seth Abraham, Israel Mitrani, and Byram Muchmore for their help.

## REFERENCES

[1] L. N. Bhuyan and C. W. Lee, An Interference Analysis of Interconnection Networks, *1983 ICPP,* pp. 2-9.

[2] J. Blum, H. Chernoff, M. Rosenblatt, and H. Teicher, Central Limit Theorems for Interchangeable Processes, *Canadian J. Math.,* v. 10, pp. 222-229, 1958.

[3] D. Y. Burman and D. R. Smith, Asymptotic Analysis of a Queueing Model with Bursty Traffic, *The Bell System Technical Journal,* v. 62, no. 6, July 1983, pp. 1433-1453.

[4] S. Cheemalavagu and M. Malek, Analysis and Simulation of Banyan Interconnection Networks with 2×2, 4×4, and 8×8 Switching Elements, *Proc. Real-Time Systems Symp.,* Dec. 82, 83-89.

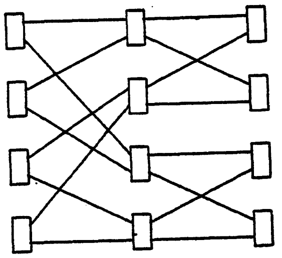
[5] D. M. Dias and J. R. Jump, Packet Switching Interconnection Networks for Modular Systems, *IEEE Computer,* v. 14, Dec. 1981, pp. 43-54.

[6] T-Y. Feng, A Survey of Interconnection Networks, *IEEE Computer* 14, Dec. 1981, pp. 12-27.

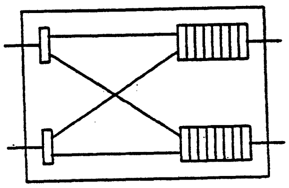
[7] D. Gajski, D. Kuck, and D. Lawrie, Construction of a Large Scale Multiprocessor, *1983 ICPP,* pp. 524-529.

[8] G. R. Goke and G. J. Lipovski, Banyan Networks for Partitioning Multiprocessor Systems, *1st Ann. Symp. on Computer Architecture,* pp. 21-28, 1973.

[9] A. Gottlieb, R. Grishman, C. P. Kruskal, Kevin P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer *IEEE Trans. on Computers,* v. C-32, pp. 175-189, Feb. 1983.



Interconnection Network
Figure 1.



2×2 switch.
Figure 2.

[10] L. Kleinrock, "Queueing Systems", Vol. 1, J. Wiley, NY, 1975.

[11] H. Kobayashi and A. G. Konheim, Queuing Models for Computer Communications System Analysis, *IEEE Trans. on Comm.,* v. COM-25, pp 2-29, 1977.

[12] C. P. Kruskal and M. Snir, The Performance of Multistage Interconnection Networks for Multiprocessors, *IEEE Trans. on Computers,* v. C-32, pp. 1091-1098, Dec. 1983.

[13] C. P. Kruskal, M. Snir, and A. Weiss, On the Distribution of Delays in Buffered Multistage Networks for Uniform and Nonuniform Traffic (Extended Abstract), *1984 ICPP,* pp. 215-219.

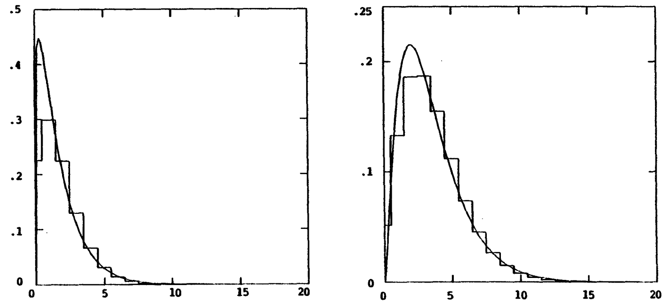
[14] D. H. Lawrie, Access and Alignment of Data in an Array Processor, *IEEE Trans. on Computers,* v. C-24, pp. 1145-1155, 1975.

[15] A. Norton and G. Pfister, A Methodology for Predicting Multiprocessor Performance, *1985 ICPP,* pp. 772-781.

[16] J. A. Patel, Performance of Processor-Memory Interconnections for Multiprocessors, *IEEE Trans. on Computers,* v. C-30, pp. 771-780, 1981.

[17] G. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *1985 ICPP,* pp. 764-771.

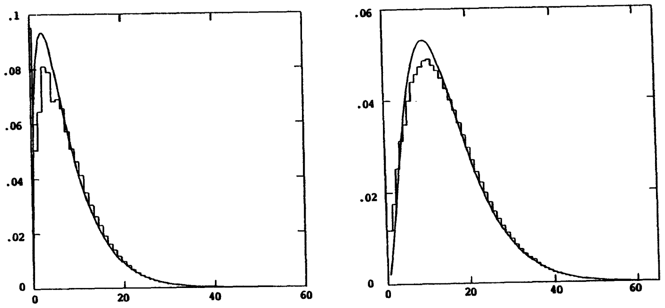
[18] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, Lexington, MA, 1985.



6 stages

12 stages

Distribution of delays, simulation and prediction, $m = 1$.
Figure 3.



6 stages

12 stages

Distribution of delays, simulation and prediction, $m = 4$.
Figure 4.

| | $p = 0.2$ | | $p = 0.5$ | | $p = 0.8$ | |
|---|---|---|---|---|---|---|
| | E | Var | E | Var | E | Var |
| ANALYSIS | 0.0625 | 0.0602 | 0.2500 | 0.2500 | 1.0000 | 1.6000 |
| SIMULATION | | | | | | |
| 1st stage | 0.0625 | 0.0602 | 0.2501 | 0.2503 | 0.9946 | 1.5824 |
| 2nd stage | 0.0657 | 0.0639 | 0.2809 | 0.3042 | 1.1884 | 2.3589 |
| 3rd stage | 0.0670 | 0.0656 | 0.2929 | 0.3267 | 1.2537 | 2.6956 |
| 4th stage | 0.0676 | 0.0663 | 0.2970 | 0.3359 | 1.2772 | 2.8459 |
| 5th stage | 0.0676 | 0.0663 | 0.2985 | 0.3391 | 1.2807 | 2.8554 |
| 6th stage | 0.0675 | 0.0662 | 0.2992 | 0.3409 | 1.2874 | 2.9004 |
| 7th stage | 0.0679 | 0.0667 | 0.2997 | 0.3428 | 1.2877 | 2.9138 |
| 8th stage | 0.0682 | 0.0669 | 0.2996 | 0.3410 | 1.2912 | 2.9227 |
| ESTIMATE | 0.0675 | 0.0656 | 0.3000 | 0.3438 | 1.3200 | 2.9440 |

Waiting times and variances: $p$ varying ($k = 2$, $m = 1$, and $q = 0$).
Table I.

| | $k = 2$ | | $k = 4$ | | $k = 8$ | |
|---|---|---|---|---|---|---|
| | E | Var | E | Var | E | Var |
| ANALYSIS | 0.2500 | 0.2500 | 0.3750 | 0.4375 | 0.4375 | 0.5469 |
| SIMULATION | | | | | | |
| 1st stage | 0.2501 | 0.2502 | 0.3740 | 0.4350 | 0.4371 | 0.5465 |
| 2nd stage | 0.2812 | 0.3048 | 0.4005 | 0.4941 | 0.4532 | 0.5848 |
| 3rd stage | 0.2928 | 0.3258 | 0.4060 | 0.5055 | 0.4559 | 0.5934 |
| 4th stage | 0.2977 | 0.3372 | 0.4067 | 0.5112 | 0.4562 | 0.5934 |
| ESTIMATE | 0.3000 | 0.3438 | 0.4125 | 0.5195 | 0.4597 | 0.5982 |

Waiting times and variances: $k$ varying ($p = .5$, $m = 1$, and $q = 0$).
Table II.

| | $m = 2$<br>$p = 1/4$ | | $m = 4$<br>$p = 1/8$ | | $m = 8$<br>$p = 1/16$ | | $m = 16$<br>$p = 1/32$ | |
|---|---|---|---|---|---|---|---|---|
| | E | Var | E | Var | E | Var | E | Var |
| ANALYSIS | 0.750 | 1.500 | 1.750 | 7.500 | 3.750 | 33.50 | 7.750 | 141.50 |
| SIMULATION | | | | | | | | |
| 1st stage | 0.749 | 1.500 | 1.750 | 7.535 | 3.752 | 33.68 | 7.791 | 142.58 |
| 2nd stage | 0.588 | 1.158 | 1.203 | 4.708 | 2.435 | 19.23 | 4.889 | 77.35 |
| 3rd stage | 0.592 | 1.186 | 1.198 | 4.710 | 2.412 | 18.81 | 4.815 | 74.71 |
| 4th stage | 0.601 | 1.217 | 1.200 | 4.718 | 2.402 | 18.77 | 4.807 | 75.27 |
| 5th stage | 0.603 | 1.230 | 1.203 | 4.736 | 2.402 | 18.81 | 4.772 | 73.90 |
| 6th stage | 0.601 | 1.224 | 1.206 | 4.800 | 2.401 | 18.77 | 4.788 | 74.72 |
| 7th stage | 0.600 | 1.219 | 1.200 | 4.754 | 2.399 | 18.79 | 4.773 | 73.78 |
| 8th stage | 0.603 | 1.234 | 1.204 | 4.777 | 2.396 | 18.73 | 4.778 | 74.35 |
| ESTIMATE | 0.600 | 1.167 | 1.200 | 4.667 | 2.400 | 18.67 | 4.800 | 74.67 |

Waiting times and variances: $p$ and $m$ varying with $\rho = .5$ ($k = 2$ and $q = 0$)
Table III.

| | $g_1 = 0.0$<br>$g_2 = 1.0$<br>$p = 1/16$ | | $g_1 = 0.25$<br>$g_2 = 0.75$<br>$p = 1/14$ | | $g_1 = 0.5$<br>$g_2 = 0.5$<br>$p = 1/12$ | | $g_1 = 0.75$<br>$g_2 = 0.25$<br>$p = 1/10$ | | $g_1 = 1.0$<br>$g_2 = 0.0$<br>$p = 1/8$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | E | Var | E | Var | E | Var | E | Var | E | Var |
| ANALYSIS | 3.750 | 33.50 | 3.464 | 29.30 | 3.083 | 23.94 | 2.550 | 16.94 | 1.750 | 7.50 |
| SIMULATION | | | | | | | | | | |
| 1st stage | 3.752 | 33.77 | 3.461 | 29.16 | 3.092 | 24.11 | 2.552 | 16.98 | 1.749 | 7.46 |
| 2nd stage | 2.454 | 19.53 | 2.255 | 16.64 | 2.039 | 13.72 | 1.706 | 9.87 | 1.199 | 4.67 |
| 3rd stage | 2.417 | 18.95 | 2.235 | 16.43 | 2.010 | 13.48 | 1.691 | 9.67 | 1.197 | 4.70 |
| 4th stage | 2.405 | 18.88 | 2.241 | 16.45 | 2.018 | 13.51 | 1.689 | 9.75 | 1.204 | 4.76 |
| 5th stage | 2.392 | 18.73 | 2.256 | 16.83 | 2.025 | 13.58 | 1.696 | 9.89 | 1.193 | 4.68 |
| 6th stage | 2.409 | 18.09 | 2.249 | 16.64 | 2.023 | 13.65 | 1.691 | 9.78 | 1.202 | 4.76 |
| 7th stage | 2.408 | 18.01 | 2.249 | 16.62 | 2.016 | 13.50 | 1.690 | 9.71 | 1.199 | 4.76 |
| 8th stage | 2.396 | 18.68 | 2.238 | 16.41 | 2.023 | 13.70 | 1.695 | 9.78 | 1.196 | 4.70 |
| ESTIMATE | 2.400 | 18.67 | 2.238 | 16.58 | 2.018 | 13.84 | 1.700 | 10.08 | 1.200 | 4.67 |

Waiting times and variances: $m_1 = 4$; $m_2 = 8$; and
$p$, $g_1$, and $g_2$ varying with $\rho = .5$ ($k = 2$ and $q = 0.0$).
Table IV.

| | $q = 0.0$ | | $q = 0.25$ | | $q = 0.50$ | | $q = 0.75$ | |
|---|---|---|---|---|---|---|---|---|
| | E | Var | E | Var | E | Var | E | Var |
| ANALYSIS | 0.2500 | 0.2500 | 0.2344 | 0.2344 | 0.1875 | 0.1875 | 0.1094 | 0.1094 |
| SIMULATION | | | | | | | | |
| 1st stage | 0.2501 | 0.2502 | 0.2341 | 0.2340 | 0.1886 | 0.1890 | 0.1093 | 0.1089 |
| 2nd stage | 0.2811 | 0.3047 | 0.2598 | 0.2771 | 0.2041 | 0.2136 | 0.1138 | 0.1157 |
| 3rd stage | 0.2919 | 0.3240 | 0.2683 | 0.2934 | 0.2087 | 0.2218 | 0.1174 | 0.1200 |
| 4th stage | 0.2967 | 0.3330 | 0.2730 | 0.3036 | 0.2116 | 0.2275 | 0.1171 | 0.1200 |
| 5th stage | 0.2983 | 0.3387 | 0.2741 | 0.3050 | 0.2117 | 0.2275 | 0.1172 | 0.1209 |
| 6th stage | 0.2992 | 0.3399 | 0.2739 | 0.3030 | 0.2108 | 0.2265 | 0.1160 | 0.1185 |
| 7th stage | 0.2998 | 0.3436 | 0.2754 | 0.3069 | 0.2100 | 0.2256 | 0.1163 | 0.1190 |
| 8th stage | 0.3000 | 0.3426 | 0.2748 | 0.3079 | 0.2085 | 0.2228 | 0.1162 | 0.1195 |
| ESTIMATE | 0.3000 | 0.3438 | 0.2695 | 0.3003 | 0.2063 | 0.2227 | 0.1148 | 0.1196 |

Waiting times and variances: $q$ varying ($p = .5$, $k = 2$, and $m = 1$).
Table V.

| | ESTIMATE | | SIMULATION | |
|---|---|---|---|---|
| | E | Var | E | Var |
| 3 stages | 0.822 | 1.033 | 0.825 | 1.018 |
| 6 stages | 1.717 | 2.426 | 1.723 | 2.377 |
| 9 stages | 2.617 | 3.864 | 2.624 | 3.775 |
| 12 stages | 3.517 | 5.307 | 3.528 | 5.180 |

Comparison of predictions with simulations:
$k = 2$, $p = .5$, $m = 1$, and $q = 0$.
Table VI.

| | ESTIMATE | | SIMULATION | |
|---|---|---|---|---|
| | E | Var | E | Var |
| 3 stages | 4.15 | 20.47 | 4.15 | 20.73 |
| 6 stages | 7.75 | 40.06 | 7.75 | 41.42 |
| 9 stages | 11.35 | 59.66 | 11.34 | 62.20 |
| 12 stages | 14.95 | 79.26 | 14.93 | 82.99 |

Comparison of predictions with simulations:
$k = 2$, $p = .125$, $m = 4$, and $q = 0$.
Table VII.

# THE SNEPTREE – A VERSATILE INTERCONNECTION NETWORK[†]

Peyyun Peggy Li[‡] and Alain J. Martin

Computer Science
California Institute of Technology
Pasadena CA 91125

**Abstract** — A new interconnection network, the *Sneptree*, is investigated. The Sneptree consists of $2^n - 1$ identical nodes and each node has four links. The links are connected to form an augmented complete binary tree where the outgoing links of the leaves are connected to all the nodes in the network. We prove that a complete binary tree with arbitrary size can be mapped onto a Sneptree optimally. Hence, the Sneptree is well suited for distributed computations with tree-structured computation graph, such as divide-and-conquer and backtracking. One type of Sneptree, which contains two disjoint spanning cycles and is thus called the *Cyclic Sneptree*, is of particular interest since it can simulate a fully unbalanced tree optimally, such as a left/right skewed tree.

A recursive method is given to generate the H-structure layout of the Cyclic Sneptree. The number of crossings and the length of the longest wires in the H-structure layout are analyzed. A message routing algorithm between any two leaf nodes is presented. The routing algorithm, which is of O(n) complexity, gives a good approximation to the shortest path. The traffic congestion in the nodes at the upper levels is also significantly reduced compared to the binary tree case.

## 1. Introduction

Due to the development of VLSI technology, it is now possible to construct powerful computers by connecting thousands of small identical processors into a so-called "processor network." Each processor has independent control and local memory. Hence, each processor can run its own program independently and asynchronously. Synchronization and communication are done by message passing between neighboring processors. The computation is distributed over the network. Hence, high concurrency can be achieved.

Many different interconnection networks have been studied, such as the binary tree [1,11], the mesh, the systolic array [4], the boolean n-cube [10], etc.. Some machines are dedicated to some special applications; some are designed as a general purpose computing engine. One interesting problem which hasn't been investigated profoundly is the mapping of the computation graph onto the implementation network, and in particular the mapping of an over-sized problem onto a fixed size network to keep the load of each processor balanced. In [7], a double-twisted torus simulates an unbounded mesh perfectly. The torus introduced a homogeneous processor network which relieves the boundary problem from a regular mesh so that a bigger mesh can be mapped onto this network automatically and optimally.

In this paper, another homogeneous processor network is presented. The *Sneptree* [12] is a class of augmented binary trees with homogeneous nodes. Each node, including the root node and the leaf node, has four links. The links are connected such that a complete binary tree of arbitrary size can be mapped onto the Sneptree optimally.

The binary tree has the property that the distance between any two nodes is at most $2\log_2 n$, where n is the size of the tree. Such a network is called "logarithmic." The Sneptree is an expanded binary tree with more links in each node. Some connection patterns of the Sneptree are regular and symmetric and hence well suited for VLSI implementation. Furthermore, it can simulate an unbounded binary tree so that it is best for divide-and-conquer type applications. There are some other augmented binary tree networks been investigated, such as the X-tree[2], the Hypertree[3], and the De Bruijn Network[9]. Those networks are different in their connection patterns and applications. The comparison between the Sneptree and other networks will be given in the conclusion.

In section 2, the definitions of the Sneptree and the Cyclic Sneptree are given and different connection patterns are presented. The mapping of a complete binary tree onto the Sneptree is proven to be optimal in section 3. Like a binary tree, the Sneptree can be laid out into an H-structure plane nicely. Section 4 presents a recursive method to construct the H-structure Sneptree. In section 5, a routing algorithm that routes a message from a leaf node to another leaf node is presented. In the conclusion, the Sneptree is compared to other networks, and future research directions are discussed.

## 2. Definition of the Sneptree

**Definition:** An *n-level Sneptree* is a complete binary tree of $2^n - 1$ nodes, links directed from root to leaves, augmented with $2^n$ additional *Snep* links directed out of the leaves, such that each node has 4 incident links: 2 directed in and 2 directed out. Each node in the tree has an incoming Sneplink, except for the root, which has 2 incoming Sneplinks.

Notice that the Sneptree is defined to be a directed graph here for easier understanding. In the real implementation, the links should be bidirectional. Furthermore, we call the outgoing link which points to the left descendant of a node the "left link" and the one pointing to the right descendant the "right link."

There are many possible ways to connect those $2^n$ Sneplinks. One example of a planar connection is shown in Figure 1. This connection is not of particular interest because it ends up with a very unbalanced mapping for a highly unbalanced binary tree, such as a left skewed tree. Another type of Sneptree whose Sneplinks are connected to form two spanning cycles (i.e., Hamiltonian Cycles) renders an optimal mapping

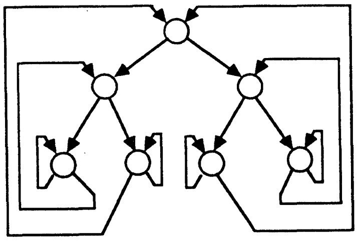


*Figure 1.* A Three–level Sneptree

[†] This research was sponsored by the Defense Advanced Research Projects Agency, ARPA Order No. 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

[‡] Present address: Ametek, Computer Research Division, 610 N. Santa Anita Ave., Arcadia, CA 91006

0190-3918/86/0000/0020 $01.00 © 1986 IEEE

for a left(right) skewed tree of any size (i.e., a linear array). This special type of Sneptree is called the *Cyclic Sneptree.*

**Definition:** A *Cyclic Sneptree* is a Sneptree containing two link-disjoint spanning cycles. The "left cycle" contains only left links and the "right cycle" contains only right links.

**Theorem 1.** *There are $[(2^{n-1}-1)!]^2$ connection patterns for the n-level Cyclic Sneptree.*

The proof is given in [6]. Notice that many of these $[(2^{n-1}-1)!]^2$ connection patterns are isomorphic because the left and the right links are indistinguishable in practice.

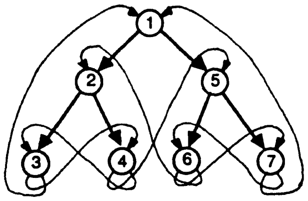


*Figure 2.* A Three-level Cyclic Sneptree

Figure 2 shows one connection pattern of the Cyclic Sneptree. The numbers attached to the nodes show the node ordering in the left spanning cycle. Symmetrically, the right spanning cycle of Figure 2 can be represented by node sequences (1,5,7,6,2,4,3,1). Such a connection pattern is regular and symmetric and it can be generated recursively from the smaller structures. This connection pattern is not planar; the two crossing Sneplinks between two adjacent subtrees make it possible to extend one subtree to the other subtree. This particular Cyclic Sneptree is chosen due to its regularity and extensibility, which are crucial properties for VLSI implementation. Another connection pattern with the same properties is compared in the conclusion. We'll show later that the connection we choose here is better than the other one.

## 3. Mapping of a Binary Tree onto a Sneptree

The mapping from a complete binary tree onto a Sneptree is independent of the connection patterns of the Sneptree. In other words, no matter how the Sneplinks are connected in a Sneptree, the mapping of a complete binary tree is always optimal. This is not true for an incomplete binary tree. The performance of the mapping of an unbalanced tree is affected by the connection pattern of the Sneptree.

Before describing the mapping performance, we shall define the ***computation graph*** and the ***implementation graph*** first. The computation graph represents the structure of the distributed computation and the implementation graph represents the network topology of the parallel machine. ***Cell*** and ***node*** are the names for a vertex of the computation graph and the implementation graph, respectively. An ***optimal mapping*** is defined as a mapping such that (1) the adjacent cells are mapped onto the adjacent nodes, and (2) the number of cells in a single node differs by at most one.

From now on, we assume that the computation graph is an m–level complete binary tree, the implementation graph is an n–level Sneptree and m ≥ n. Again, a cell denotes a node in the binary tree and a node denotes a node in the Sneptree. For this particular mapping problem, we use two figures to measure the mapping performance. The first figure is the total number of cells mapped onto one single node, so-called the *load factor*, which indicates the total work load of each node. The second figure is the number of cells of the same height in the binary tree mapped onto one single node, which is an useful measure when the computation wavefront goes downward and upward in the tree so that only the nodes at one particular level are active at a time. In the following, we are going to show that both measures are minimal for the mapping of a complete binary tree onto a Sneptree. Therefore, the mapping is optimal.

In an n–level complete binary tree, the root is at level 0 and the leaves at level (n–1). The height of a node is defined to be the distance of that node to the leaves. The height of a binary tree is the distance of the root to the leaves, which is (n–1) for a n–level complete binary tree. The above definitions also apply to a Sneptree.

The optimal mapping from a complete binary tree onto a Sneptree is to map the root of the binary tree onto the root of the Sneptree and the two children of a cell onto the two direct descendants of a node.

**Theorem 2.** *All the nodes in the Sneptree contain the same number of cells of one particular level, say $k$, except that each node at ($k$ mod $n$) level contains one more cell, when mapping an m-level complete binary tree onto an n-level Sneptree, $m \geq n$ and $0 \leq k < m$.*

*Proof:* If $k < n$, one cell will be mapped onto one node at the k-th level in the Sneptree. For $k \geq n$, the theorem can be proven by observing the construction of the Sneptree. A node at the j-th level of the Sneptree has two direct ancestors; one is its father at (j–1)-th level, and the other is a leaf, i.e. a node at the (n–1)-th level. Moreover, the number of cells of level k which are mapped onto this node is the sum of the cells of level (k–1) which are located in its direct ancestors. In other words,

$$T_k(j) = T_{k-1}(j-1) + T_{k-1}(n-1), \qquad j > 0 \qquad (1)$$

where $T_k(j)$ is the total number of cells at the k-th level of the binary tree which are mapped onto one node located at the j-th level of the Sneptree for $0 \leq k < m$ and $0 \leq j < n$.

The root node, i.e., the node at the 0-th level, has no upper level and its two direct ancestors are both from the bottom level, i.e. the (n–1)-th level. Combine with Eq.(1), $T_k(j)$ can be recursively defined by

$$T_k(j) = T_{k-1}((j-1) \bmod n) + T_{k-1}(n-1) \qquad (2)$$

where $n \leq k < m$ and $0 \leq j < n$. For $k < n$,

$$T_k(j) = \begin{cases} 0, & \text{if } 0 \leq k < n, 0 \leq j < n \wedge j \neq k; \\ 1, & \text{if } 0 \leq k < n, 0 \leq j < n \wedge j = k. \end{cases}$$

By induction, we assume that the theorem holds for $k$, $k \geq n$. Let $k = q \times n + r$, then $T_k(r) = T_k(j) + 1$, for all $j$ and $j \neq r$. We now prove that the theorem holds for $k+1$. From Eq.(2), when $r \neq n-1$,

$$\begin{aligned} T_{k+1}((k+1) \bmod n) = T_{k+1}(r+1) &= T_k(r) + T_k(n-1) \\ &= 2 \times T_k(n-1) + 1, \quad \text{and} \end{aligned}$$

$$\begin{aligned} T_{k+1}(j) &= T_k((j-1) \bmod n) + T_k(n-1) \\ &= 2 \times T_k(n-1), \quad j \neq (k+1) \bmod n. \end{aligned}$$

If $r = n-1$,

$$\begin{aligned} T_{k+1}((k+1) \bmod n) = T_{k+1}(0) &= T_k(n-1) + T_k(n-1) \\ &= 2 \times T_k(j) + 2, \quad \text{and} \end{aligned}$$

$$\begin{aligned} T_{k+1}(j) &= T_k((j-1) \bmod n) + T_k(n-1) \\ &= 2 \times T_k(j) + 1, \quad j \neq (k+1) \bmod n. \end{aligned}$$

Therefore, $T_{k+1}((k+1) \bmod n) = T_{k+1}(j)+1, j \neq (k+1) \bmod n$, holds for any k.

By induction, the theorem holds. ▮

**Theorem 3.** *All the nodes at the top ($m \bmod n$) levels of the Sneptree contain the same number of cells, Similarly, the rest of the nodes also contains the same number of cells and the amount is one less than that in the top level nodes, when mapping an m-level complete binary tree onto an n-level Sneptree, $m \geq n$.*

*Proof*: Let $T(j)$ be the total number of cells mapped onto a node at the $j$-th level of the Sneptree, i.e., $T(j) = \sum_{k=0}^{m-1} T_k(j)$. Let $m = q \times n + r$ and consider a node at one particular level $j$. Such a node contains one more cell at $(n+j)$-th, $(2n+j)$-th, ..., and $(q \times n + j)$-th levels respectively than the nodes not in level $j$, when $j \leq r$. In other words, there are $q$ such levels in the binary tree, in which one extra cell is assigned to the nodes at level $j$, for $j \leq r$. For $j > r$, there exists only $q - 1$ such levels. Therefore, we can conclude that all the nodes at the top $r = (m \bmod n)$ levels of the Sneptree contain the same number of cells, Similarly, the rest of the nodes also contains the same number of cells, and the amount of cells is one less than that in the top level nodes. ▮

**Corollary 4.** *For* $k = n, n+1, \ldots, m-1$

$$T_k(j) = \begin{cases} \left\lfloor \dfrac{2^k}{2^n - 1} \right\rfloor, & \textit{if } 0 \leq j \leq n-1 \textit{ and } \\ & \quad j \neq k \bmod n \\ \left\lfloor \dfrac{2^k}{2^n - 1} \right\rfloor + 1, & j = k \bmod n. \end{cases}$$

*and*

$$T(j) = \begin{cases} \left\lfloor \dfrac{2^m - 1}{2^n - 1} \right\rfloor + 1, & \textit{for } 0 \leq j < (m \bmod n) \\ \left\lfloor \dfrac{2^m - 1}{2^n - 1} \right\rfloor, & \textit{for } (m \bmod n) \leq j < n. \end{cases}$$

The above corollary can be derived immediately from Theorem 2 and Theorem 3. We now get to the conclusion which has been addressed at the beginning of this section.

**Theorem 5.** *The mapping of a complete binary tree onto a Sneptree is always optimal for any connection patterns of the Sneptree.*

From the above discussion, it is clear that an arbitrary size complete binary tree can be mapped onto a Sneptree optimally no matter how it is connected. On the contrary, the performance of mapping an unbalanced binary tree is dependent on the connection pattern of the Sneptree.

**Theorem 6.** *A left or right skewed tree of any size can be mapped onto the Cyclic Sneptree optimally.*

*Proof*: The theorem is true from the definition of the cyclic Sneptree. ▮

**Remark** A linear array of any length can be mapped onto the Cyclic Sneptree optimally if we map the linear array onto the left cycle or the right cycle of the Cyclic Sneptree.

## 4. Layout of a Cyclic Sneptree

In this section, we discuss how to layout a Cyclic Sneptree (shown in Figure 2) on a plane. From now on, we call the Cyclic Sneptree in Figure 2 "Sneptree" since all the discussions in Section 4 and 5 are based on this particular connection pattern. The H-structure layout for a binary tree [8] is modified to layout a Sneptree. The recursive rule to generate the layout is described. Also, the number of crossings and the length of the longest wires are analyzed. Finally, we will present a way to extend the size of the Sneptree by connecting two identical smaller Sneptrees.

### Recursive Generation of H-structure Layout

Like the binary tree, the Sneptree can be laid out into an H-structure plane. Because of the Sneplinks in the Sneptree, it is not that straightforward to build the H-structure Sneptree. The major concern is to minimize the number of crossings in the layout and keep the length of the Sneplinks as short as possible. With the two criteria in mind, a recursive construction algorithm is designed.

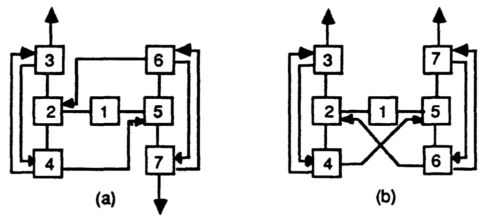


*Figure 3.* Two basic three–level H–Sneptrees

The n–level Cyclic Sneptree can be constructed recursively into an H–structure layout with two given basic three–level H–Sneptrees, $A_3$ and $B_3$ (Figure 3). In Figure 3, the node numbering is compatible with that of Figure 2. The dangling arrows out of node 3 and node 7 are the two links incident into the root in a regular 3-level Cyclic Sneptree. These two links are dangling in order to extend to bigger structures.

Let's define two basic operations on the layout G:

(a) mirror along x axis : $G^x$
(b) mirror along y axis : $G^y$

The recursive rules are as follows:

1. Construct two 3-level H-structure Sneptrees $A_3$ and $B_3$ as shown in Figure 3. $A_3$ is the one we intend to construct and $B_3$ is an auxiliary layout to be used in constructing bigger Sneptrees. Now we like to construct $A_n$ for all $n \geq 3$ and $B_n$ for $n \geq 3$ and n odd.
2. Given two k-level H-structure Sneptrees, $A_k$ and $B_k$, for $k \geq 3$ and odd, the (k+1)-level and (k+2)-level H-structure Sneptrees can be constructed as shown in Figure 4. A (k+1)-level Sneptree, $A_{k+1}$, can be constructed by two k-level subtrees, namely $A_k$ and $B_k$. And a (k+2)-level Sneptree, $A_{k+2}$, can be constructed by four k-level Sneptrees, $A_k$, $B_k$, $B_k^y$ and $A_k^x$. The auxiliary (k+2)-level Sneptree, $B_{k+2}$, can be constructed by $A_k^x$, $B_k$, $A_k^y$ and $B_k$.

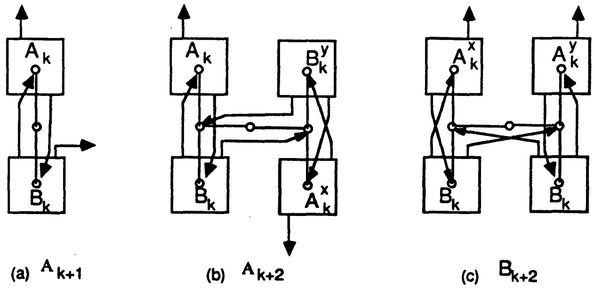


*Figure 4.* Construction of $A_{k+1}$, $A_{k+1}$ and $B_{k+2}$

For example, a 4–level Sneptree is constructed by connecting $A_3$ and $B_3$ to an extra node and by directing the Sneplinks as shown in Figure 5.a. Notice that $A_3$ is planar and $B_3$ has one crossing. $A_4$ has five crossings due to the introduction of new links, including the two links incident to the root, shown as dotted lines in Figure 5.a. The dotted arrows into the root node should be connected to the two dangling links coming out of the leaf nodes to make it a complete 4-level Sneptree. A 5–level Sneptree can be constructed as in Figure 5.b. There are in total eleven crossings in the layout; five in the left half of the graph, which is exactly $A_4$ except for the two links of the root coming out to the right; four in the right half as shown in the figure and two from the incoming links (dotted lines) of the root in the middle of Figure 5.b.

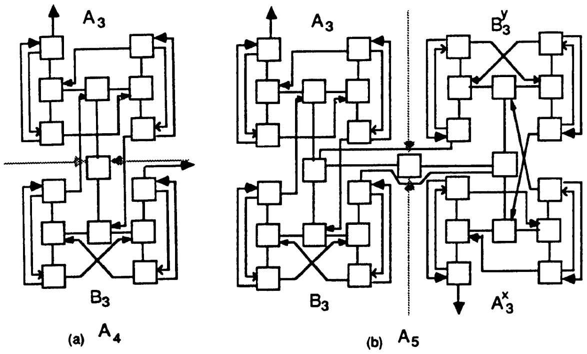


*Figure 5.* Construction of $A_4$, $A_5$ using $A_3$ and $B_3$

Since the Cyclic Sneptree is not planar and the Sneplinks are not of constant length, we would like to know the number of crossings and the maximum length of the Sneplinks.

Theorem 7 gives the number of crossings in $A_n$ and $B_n$. $B_n$ has one more crossing than $A_n$ and both figures are approximately 3/8 of the total number of nodes in the Sneptree. The two crossings introduced by the incoming links of the root node in $A_n$ are not counted. The proof of Theorem 7 is given in [6].

**Theorem 7.** *The number of crossings in $A_n$ is $3 \times (2^{n-3} - 1)$ and the number of crossings in $B_n$ is $3 \times (2^{n-3} - 1) + 1$.*

Assume that any single node in the layout is a square with area $a$, i.e., each side of the node is $\sqrt{a}$ in length and the wire width is negligible compared to $\sqrt{a}$. The area of the H-structure Sneptree is the function of node area $a$ and the height of the Sneptree $n$. The zero wire width assumption is reasonable because the number of wires passing through any two adjacent nodes in the layout is bounded by a fixed number.

Furthermore, we assume that the four links of each node may be pulled out of any side of the node. Two or more links may come out of the same side. The length of the wire connecting two nodes is the shortest distance from the center of one side of the source node to the center of the nearest side of the other node. The wire has to route around all the nodes in the way.

Theorem 8 shows that the length of the longest internal wire in an H-layout Sneptree is about 1/4 of the width of the layout. It is only $(3/2)\sqrt{a}$ longer than the longest wire of the H-layout binary tree of the same size. The proof of Theorem 8 is given in [6].

**Theorem 8.** *The longest internal wires of an n-level H-structure Sneptree are the two wires connecting the root and two leaf nodes at the left and the right corners. The length is*

$$l_n = \begin{cases} \sqrt{a}(3 \times 2^{n/2-3} + 1/2), & n > 3 \text{ and even} \\ \sqrt{a}(2^{(n-3)/2} + 1/2), & n > 3 \text{ and odd} \\ 2\sqrt{a}, & n \leq 3 \end{cases}$$

### Extension of the Sneptree

From the above discussion, it is clear that we can layout a Sneptree of any size onto a single chip as long as the chip capacity is not exceeded. Here we present a method to extend the Sneptree by connecting two identical H-structure Sneptrees, which is modified from the recursive construction technique of binary tree from a single chip proposed by [5].

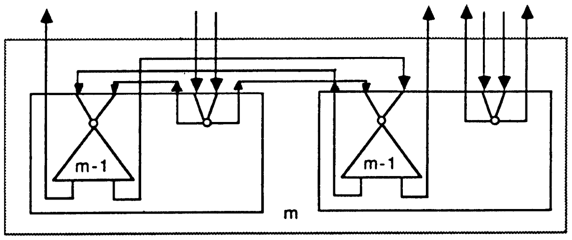


*Figure 6.* Extension of the two (m-1)-level Sneptrees to an m-level Sneptree

Let one chip consist an (m–1)-level H-structure Sneptree with four dangling links and a single processor with its four links. Two of the four dangling links in the Sneptree are out of the leftmost and the rightmost leaf nodes, respectively. The other two are the incoming links to the root node. There are eight connectors in a single chip as shown in the solid box in Figure 6.

Figure 6 illustrates how to connect two such chips into an m-level Sneptree. The resulting layout contains one m-level Sneptree with four dangling links and a single processor, which is now able to extend to bigger structures recursively.

## 5. Routing Algorithm for Leaf Nodes

In this section, a leaf node routing algorithm for the Sneptree is presented. It is motivated by the opportunities to map a linear array onto the leaf nodes of the Sneptree and to utilize the Sneplinks to shorten the communication distance between two arbitrary leaf nodes.

The design of the routing algorithm is constrained by the following criteria: *communication distance* (to find a route as short as possible), *congestion constraint* (to use the extra links to avoid traffic jam at the upper level nodes), and *time constraint* (to keep the routing time as low as possible).

The time to route a message from a node x to another node y is the sum of the message transmission time and the processing time at the source node and each intermediate node. Let $t_p$ and $t_c$ be the time of one processing step and the time of one message transmission between adjacent nodes, respectively. Suppose $< x = x_0, x_1, \ldots, x_{k-1}, x_k = y >$ is the route which the message is sent through and $f(i)$ is the number of processing steps necessary to compute the following route at the intermediate node $x_i$. The total routing time is

$$\sum_{i=0}^{k-1} f(i)t_p + k \times t_c \qquad (3)$$

In an n-level binary tree, $k$ is bounded by $2(n-1)$ and $f(i)$ is constant for all $i$ so that the routing time is O(n). Notice that the bitwise operations, such as detecting if one node is in the subtree of another node, is assumed to take constant time. In a Sneptree, it is obviously that the shortest distance of any two nodes is not longer than that in the binary tree due to the Sneplinks of the Sneptree. Hence, the second term, k, of Eq.(3) for the Sneptree is smaller than that for the binary tree. To keep the total routing time for the Sneptree in the same order of magnitude as for the binary tree, we have to keep $f(i)$ constant in the intermediate nodes. As a consequence, the algorithm can't always find a shortest route. It finds a route which is shorter and less congested than the one in a pure binary tree in O(n) time.

The leaf node routing algorithm is presented in the following subsection and the program is given in [6].

### The Routing Algorithm

Before describing the algorithm in detail, we shall first define the *breadth-first normal ordering* of the nodes.

**Definition.** The *breadth-first normal ordering* is an addressing method for a binary tree. The nodes in an n-level binary tree are numbered from 1 to $2^n - 1$. The root node has address 1, and the left descendant and the right descendant of a given node $x$ have addresses $2x$ and $2x + 1$, respectively.

Suppose that each address is represented by an n-bit binary number; the addresses of the left and the right descendants of a nonleaf node are derived by shifting its address one bit to the left and adding 0 or 1 to it.

With such an addressing scheme, the binary address of the lowest common ancestor of any two leaf nodes can be easily decided, which is an n-bit binary number with leading zeros followed by the common prefix of the binary addresses of the two leaf nodes. Furthermore, the binary addresses of the left and the right corner leaf nodes of a subtree with height h have h trailing 0's and h trailing 1's, respectively.

In our routing algorithm, the source node computes and selects the shortest route between the source node and the destination node. Then, a four-variable message carries all the routing information necessary for a receiving node to determine the next node on the route. The intermediate nodes need not recompute the shortest routes.

Suppose a message is sent from a leaf node x to a leaf node y in a Sneptree of height n. Without loss of generality, we assume x<y; i.e., node x is to the left of y. Let A be the lowest common ancestor of x and y, B and E be two direct descendants of A, and triangles BCD and EFG be the two subtrees containing x and y (see Figure 7). In the sequel, UV denotes the shortest path between two nodes U and V, and |UV| represents the length of this path. Four possible routes between x and y are (xB,BAE,Ey), (xD,DE,Ey), (xB,BF,Fy) and (xD,DEABF,Fy). The lengths of these four routes are |xB|+|Ey|+2, |xD|+|Ey|+1, |xB|+|Fy|+1, and |xD|+|Fy|+4, respectively. In order to find the shortest route among the four candidates, we need to compute |xB| ,|Ey|, |xD|, and |Fy|. Notice that |xB| and |Ey| are bounded by the height of the triangle BCD (or EFG), and |xD| and |Fy| are bounded by twice of the height of the minimal subtree containing x and D (or y and F). The routing algorithm takes advantage of the Sneplinks within the triangles to find the shortest paths xB,xD,yE, and yF.

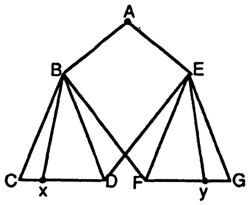


*Figure 7.* Four Possible Routes Between x and y

The length of xB (or yE) can be computed recursively. The shortest distance between B and a leaf node is 2 regardless of the height of B. The leaf nodes that have distance 2 from B are the two inner corner leaf nodes of the two subtrees of node B, and the shortest paths take one of the treelinks to a descendant of B and then take the Sneplink to the leaf (see Figure 8.a). Let a and b be the two direct descendants of B and let c and d be the two corner leaf nodes that have distance 2 from B. Then the leaf nodes that are at distance 3 from B are the nodes at distance 2 from nodes a or b, as well as the nodes at distance 1 from nodes c or d. There are six such nodes: two (a1 and a2 in Figure 8.b) are at distance 2 from a, two (b1 and b2) at distance 2 from b, and c1 and d1 are at distance 1 from c and d, respectively. Applying this technique recursively, we can find the shortest path between any leaf node and node B. The shortest path xB for an arbitrary leaf node x is a path starting from node B, following the treelinks down to a certain level of the tree, then taking the Sneplink to a leaf node and following the shortest route from this leaf node to node x (see Figure 9.a).

To find xB, the algorithm finds a sequence of leaf nodes whose binary addresses differ from x only in trailing bits, and their trailing bits are all 1's or all 0's. Those leaf nodes can be routed to node B through a Sneplink so that the distance to B is shorter that the height of B. The length of a route from x via one of such leaf node, say $x_j$, to B is the sum of the shortest distance of x to $x_j$ and the distance of $x_j$ to B.

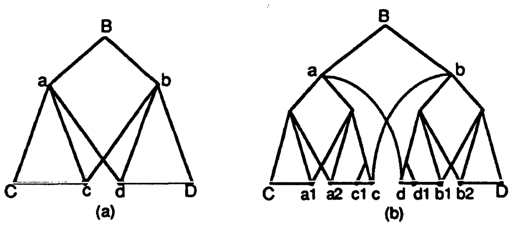


*Figure 8.* The Leaf Nodes with Distance 2 or 3 from Node B

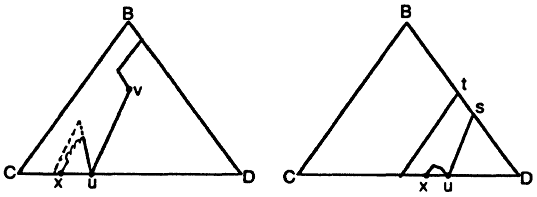


*Figure 9.* The Shortest Paths xB and xD

After all such routes have been computed, the shortest one is the candidate to the shortest route of xB. If the shortest distance is longer than the height of B, then the direct route from x to B (going upwards through treelinks) is the shortest route. For instance, let the binary address of x be 00111010, where we ignore the leading bits that are the common prefix of the binary address of x and y because they are irrelevant in computing xB. The height of B is 7 so that the shortest distance of x and B won't exceed 7. The first leaf node which can take advantage of one of the Sneplinks is $x_1$=00111011 ($x_1$ is derived by changing the LSB of x to 1), which is of distance 1 from x and 6 from B by taking the Sneplink. Hence, the distance is 7 by routing through $x_1$. The second leaf node is $x_2$=00111111, which happens to be node c in Figure 8.a and has distance 2 from B. The distance of x and $x_2$ can be computed recursively and it turns out to be 4. Therefore, the distance of xB by routing through $x_2$ is 6, which is shorter than 7. Since there are no other leaf nodes which can take advantage of the Sneplinks, we can conclude that the shortest route of xB is from x to $x_2$, taking the Sneplink to the right descendant of B, and then up to B.The length of the shortest route is 6.

The distance of xD can be derived during the computation of xB. Let the lowest common ancestor of x and D be t, the right descendant of t be s, and the leaf node at the other end of the Sneplink out of s be u. Then, the route (Bs,su,ux) is one of the candidates for the shortest path Bx. Hence, the distance of ux is computed while computing xB and the shortest path between x and D is (Ds,su,ux) whose distance can be derived immediately (see Figure 9.b).

In the computation described above, we need to find the shortest route from x to another leaf which is a corner node of a subtree containing x. Again, this distance can be computed recursively. For instance, to route xu in Figure 9.a, we try to find a sequence of leaf nodes starting with x and ending with u. Each pair of adjacent nodes in the sequence has Hamming distance 1. We now route the message through the nodes in the sequence. The distance from x to any intermediate node can be computed recursively by the previous value. Let u be a right corner leaf node of a subtree and let the node sequence be $(x = x_0, x_1, \ldots, x_k = u)$, where $x_i$ is derived from $x_{i-1}$ by changing the least significant 0 of $x_{i-1}$ to 1, and k is the Hamming distance of x and u. Notice that the address of u has m trailing 1's, where m is the height of the minimal subtree containing x and u. The recurrence relation is

$$|x_0 x_i| = \min(|x_0 x_{i-1}| + j,\ \ 2 \times j) \tag{4}$$

where $j$ is the position of the bit that differs in $x_{i-1}$ and $x_i$. All the bits to the right of the $j$-th bit of $x_{i-1}$ and $x_i$ are 1's (Figure 10). The shortest route from $x_{i-1}$ to $x_i$ takes the left Sneplink of $x_{i-1}$ to an ancestor node of $x_i$ and then takes the right tree links down to $x_i$. The distance is j, i.e., the height of the lowest common ancestor of $x_{i-1}$ and $x_i$. Furthermore, the shortest route between $x_0$ and $x_i$ could be either the route $(x_0 x_{i-1},\ x_{i-1} x_i)$ or the route containing only treelinks and passing through the lowest common ancestor of $x_0$ and $x_i$ (the distance of this route is $2 \times j$). In the previous example, the shortest route from x=00111010 to $x_2$=00111111 is taking the Sneplink to leaf node $x_1$=00111011, then Sneplink again to the ancestor of $x_2$ and taking the right treelinks down to $x_2$. The distance is 1(x to $x_1$)+3($x_1 x_2$)=4.

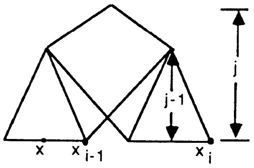


*Figure 10.* The Shortest Path Between $x_{i-1}$ and $x_i$

Similarly, if u is a left corner leaf node (with trailing 0's), |xu| can be derived by computing the distance of x and a series of intermediate nodes whose addresses are derived by changing the least significant nonzero bits of x to 0 until it reaches u.

Now, we can select the shortest path among the four candidates, (xB,BAE,Ey), (xD, DE,Ey), (xB,BF,Fy) and (xD,DEABF,Fy). So far, all the computation and decision being made are accomplished at the source node x. To achieve O(n) time performance, we don't want to repeat the computation in any other intermediate nodes along the route. Hence, the routing information should be sent to the intermediate nodes to guide them to select the proper next node along the route. It appears that a four-variable message is enough to carry the route information and avoid extra computation.

When the shortest route goes through xB, two figures are needed to guide the route. Suppose the message is routed from x to another leaf node u, then take the Sneplink to a nonleaf node v and send up to node B, (Figure 8.b), we first need to know the route information of xu. If the route xu follows the treelinks up and down to some intermediate leaf node, (i.e., when $2j$ is the smaller one in Eq.(4).) we record the highest point of this route, otherwise it is zero. The second figure we need to know is the distance of B and v. Let's call these two figures $m1$ and $h1$. When xB contains treelinks only, $m1$ is zero and $h1$ is the height of node B. Similarly, $m2$ and $h2$ are the corresponding information needed to describe the route yE. Furthermore, we need a direction flag to guide the message to either the left descendant, the right descendant or the father node.

When route xD is in the shortest path, the only information we need to know is the highest point that the route reaches through the treelinks. (i.e., when $2j$ is the smaller one in Eq.(4).) Let's call it $l1$ and when the route contains no upward treelinks, $l1$ becomes zero. Similarly, $l2$ is the corresponding information needed to describe route yF.

Let a four-variable message be $(level1, dir, level2, dest)$. In general, the routing information for the route in triangle BCD is carried in the first two variables of the message. The third variable carries the information for the route in triangle EFG. The last variable is always the destination. More specifically, $level1$ carries the value of $l1$ or $m1$ depending on which route is selected. Variable $dir$ is usually a three-value variable used to select the next node in the route when a leaf node or the highest nonleaf node is reached. When xB is selected, $h1$ is also carried by $dir$. Variable $level2$ carries either the value of $h2$ or $l2$ depending on whether yE or yF is selected in triangle EFG. The value of $m2$ has to be reproduced by a specific node in route Ey when Ey is selected, since there is no room to carry the value in a message. That specific node is the lowest nonleaf node traveling down from E through treelinks, from where the route takes the Sneplink to a leaf node and then takes the shortest route to y. Such a node corresponds to node v in route xB (see Figure 9.b). The second variable is also used to select one of the four possible routes. When $dir$ is used to carry the direction information of the route in the triangle BCD, the route information for node A,B and E is carried by the fourth variable instead. For instance, the value of the fourth variable is negative when yF is selected. The routing information for triangle EFG will be resumed at node A,B or E by moving the information carried by the third and the fourth variables back to the first two variables.

### Performance

The computation time in the source node $x$ is O($n$). When Ey is selected, one of the intermediate nodes along route Ey needs to reproduce the value of $m2$ in $k$ steps, where $k$ is the height of this specific node. When xB is selected, a few nonleaf nodes along the route need to compute the height of the lowest common ancestor of themselves and the destination node. Such bitwise operation is again assumed to take constant time. In conclusion, only the source node and at most one intermediate node need to do some computation in O(n) time. From Eq.(3), we can conclude that the routing algorithm takes O(n) time to route the message from the source to the destination.

The result of the routing algorithm gives a good approximation to the shortest path of xy. Furthermore, the routing algorithm always find the shortest path within the triangle ACG. This routing algorithm uses only the links local to the minimal subtree containing the source and the destination nodes. The Sneplinks external to this subtree are never considered. As a consequence, the two Sneplinks of the root node are never used for routing. Because of this restriction, the routing algorithm does not always compute the shortest path. (For example, the route from the left corner leaf to the right corner leaf has a distance 2, whereas our algorithm chooses a route of length twice the height of the tree.) However, this restriction has many advantages. The algorithm is simple and yet computes nearly optimal routes, and the traffic of the upper level nodes is reduced.

In a binary tree, the nodes at the upper levels are the most congested nodes because half of the leaf nodes have to route through the root node to communicate with the other half of the leaf nodes. In case any leaf node is communicating with all the other leaf nodes, the root node has to transmit about half of the messages and the nodes one level down the root have to transmit 5/8 of the messages. Then, the traffic at each node decreases level by level from 13/32, to 29/128,.... In a Sneptree, four routes may be chosen to route a message between two arbitrary leaf nodes and only two of them pass through the lowest common ancestor of the two leaf nodes. Assuming the four alternatives are equally possible, the traffic at the common ancestor is reduced to a half of the binary tree case and the traffic at the nodes one level down the common ancestor is reduced to three quarters since three of the four routes pass through that node (see Figure 7). In case any leaf node is communicating with all the other leaf nodes, the traffic at the top level nodes become 1/4, 7/16, 19/64, 43/256,... of the total amount of messages. The figures show that the traffic at the top level nodes is reduced to about a half of the binary tree case. The actual figures depend on the height of the Sneptree. The traffic at the nodes of the same height is no longer the same and the exact figures need more analysis.

The simulation result shows that the average routing distance of any two leaf nodes is getting closer to the optimal average distance when the Sneptree is bigger. The simulation also shows that for some specific communication patterns, the routing result is almost optimal, such as shift by $2^k$ operations, i.e. routing one leaf to another leaf at $2^k$ distance apart. Figure 11 shows the optimal results and our routing algorithm results of the average distance of any two leaf nodes and the average distance of a perfect shuffle operation. Figure 12 shows the average distance of shift by $2^k$ operations, the curves for routing results and optimal results overlap in Figure 12.

## 6. Conclusion

The Sneptree is a versatile interconnection network for distributed computation. The boundary problem of a binary tree is eliminated in the Sneptree so that the mapping of an over-sized computation tree is done automatically. Moreover, a complete binary tree of arbitrary size can be mapped onto a Sneptree optimally. And a left/right skewed tree can be mapped onto a Cyclic Sneptree optimally.

The Sneptree is also suitable for VLSI implementation. It is possible to build a Sneptree of any size in a single chip with area proportional to the total number of processors. The H-structure layout of the Sneptree is regular and can be recursively constructed. The number of crossings due to the extra links is proportional to the number of nodes of the Sneptree. The longest wire length is about the same as that in an H-structure binary tree. Furthermore, the Sneptree can be expanded easily by connecting two or more chips together.

The leaf node routing algorithm allows us to take advantage of the extra links in the Sneptree. Hence, a shorter and less congested route between any two leaf nodes can be found in O(n) time. The routing algorithm gives a good approximation to the optimal solution. In some special communication pattern, such as shifting by $2^k$, the average routing result is almost optimal. Besides, the traffic at the upper-level nodes is reduced to about a half of the traffic in a pure binary tree.

### Comparison with Other Networks

Like the Sneptrees, the X-tree [2] is an augmented binary tree with identical nodes. Three ports per node, four ports per node and five ports per node are considered. The degree of each node is not fixed but the maximal degree is limited by the number of ports per node. Besides the binary tree connection, the extra ports can be connected arbitrarily. The main purpose of the X-tree is to provide fault-tolerance and uniform message traffic.

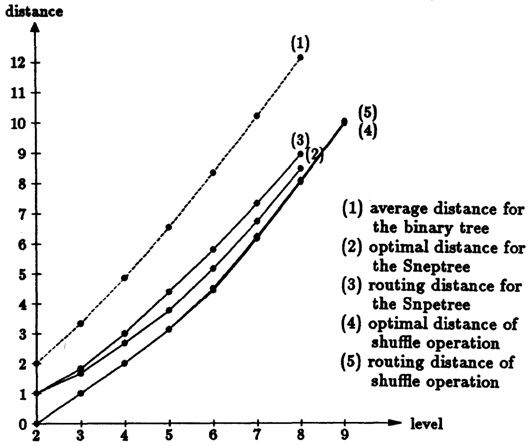


*Figure 11.* The average routing distances

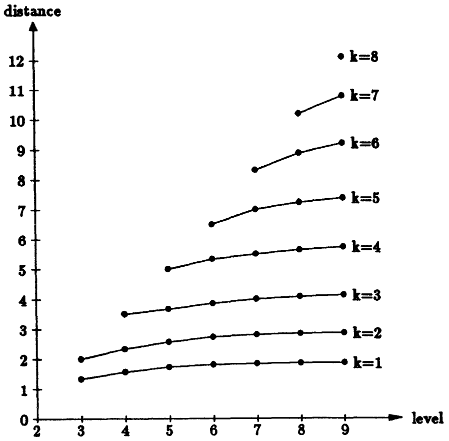


*Figure 12.* The ave. routing dist. of shifting by $2^k$ ops.

The Hypertree [3] is a binary tree with extra horizontal links (i.e., the links connecting the nodes located at the same level). The horizontal links provide a set of n-cube connections. Four ports per node and five port per node are considered. Similarly, the main concern of the Hypertree is to provide fault-tolerance and shorten the distance between two arbitrary leaf nodes.

De Bruijn Networks [9] are a class of fixed degree logarithmic networks with arbitrary number of nodes and degree. A De Bruijn Network with $(2^n - 1)$ nodes and degree 4 happens to be a Sneptree. Such networks are good for a communication network since the optimal routing path can be decided with local information and fault-tolerance is easily provided.

Comparing with the other similar networks, the Sneptree is the only network which can simulate an over-sized binary tree. The X-tree and the Hypertree contain extra links between sibling nodes so that it can simulate ring connection or n-cube connection. They cannot handle the mapping of an over-sized problem well. The de Bruijn network with degree 4 is one type of Sneptree. The connection pattern is neither cyclic, symmetric nor regular. There is no way to layout the network or extend the network.

### Different Connection Patterns

From Theorem 1, we know there are many different connection patterns for the Cyclic Sneptree. It will be interesting to compare the performance of different connection patterns of the Cyclic Sneptree in terms of the communication distance and the mapping performance of an unbalanced tree.

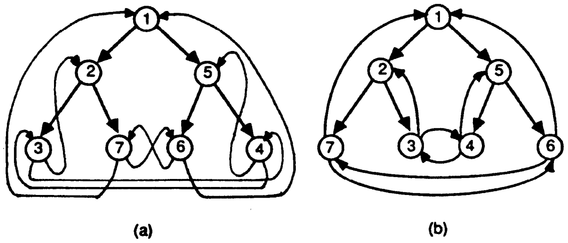


*Figure 13.* A Planer Cyclic Sneptree

Figure 13.a shows another Cyclic Sneptree. the numbers attached to the nodes show the node ordering in the left spanning cycle. Symmetrically, the right spanning cycles can be represented by node sequence (1,5,4,3,2,7,6,1). Such connection pattern also has regular structure and hence can be generated recursively. It is interesting to observe that this connection is planar if we switch the position of the leaf node pairs (3,7) and (6,4) as shown in Figure 13.b. Comparing the Cyclic Sneptree shown in Figure 2 with this one, the latter one (Figure 13.b) contains four duplicate links, i.e., (2,3), (4,5), (3,4) and (6,7) while the former one (Figure 2) has only two duplicate links, i.e., (3,4) and (6,7). The duplicate links prevent the Sneptree from connecting more nodes together. Hence, the second connection pattern doesn't perform as well as the first one in terms of communication.

There are many other connection patterns for the Cyclic Sneptree, some of them may perform better than the one we chose in terms of the communication distance between arbitrary two nodes and the mapping performance of an unbalanced tree. But only the two connection patterns discussed above can be constructed recursively from the smaller Sneptrees without breaking the internal Sneplinks in the smaller structures. This property is important for VLSI implementation.

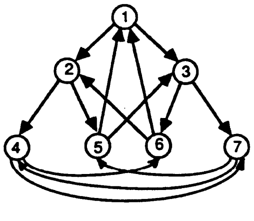


*Figure 14.* A three–level Exchange Sneptree

The *Exchange Sneptree* is a different type of Sneptree. In a Exchange Sneptree, the outgoing Sneplinks of the leaves in the left half of the Sneptree are directed to the incoming Sneplinks of the nodes in the right half plus one incoming link of the root, and similarly for the other half of the Sneplinks.

One example of the Exchange Sneptree is shown in Figure 14. This connection is symmetric but neither extensible nor cyclic. This connection has a very nice property: no matter which node the root is mapped onto, it results in a nearly optimal mapping for a complete binary tree. As a consequence, we found that mapping performance of a unbalanced binary tree onto such a Exchange Sneptree is better than the same mapping onto a Cyclic Sneptree. The properties of the Exchange Sneptree need further investigation.

## Reference

[1] Browning, S.A., *The Tree Machine: A Highly Concurrent Computing Environment,* Ph.D. thesis, Caltech, Computer Science, 1980.

[2] Despain, A.M. and D.A. Patterson, *X-tree: A Tree Structured Multi-Processor Computer Architecture,* Conference Proceeding of the 5th Symposium on Computer Architecture, 1978, pp. 144-151.

[3] Goodman, J.R. and C.H. Sequin, *Hypertree: A Multiprocessor Interconnection Topology,* Computer Science Technical Report #4227, Apr. 1981.

[4] Kung, H.T., *Why Systolic Architectures,* IEEE Computers 15(1), January, 1980, pp 37-56.

[5] Leiserson, C.E., *Area-Efficient VLSI Computation,* Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, Oct. 1981.

[6] Li, P., *A Parallel Execution Model for Logic Programming,* Ph.D. Dissertation, Computer Science, Caltech, 1986.

[7] Martin, A.J., *The TORUS: an Exercise in Constructing a Processing Surface,* Proceedings of Second Caltech Conference on VLSI, Jan. 1981, pp. 527-538.

[8] Mead, C. and M. Rem, *Cost and Performance of VLSI Computing Structures,* IEEE Journal of Solid State Circuits, Vol. SC-14, No. 2, April 1979, pp. 455-462.

[9] Schlumberger, M.L., *De Bruijn Communications Networks,* Ph.D. Dissertation, Computer Science, Stanford University, 1974.

[10] Seitz, C.L., *The Cosmic Cube,* CACM, Jan. 1985.

[11] Shaw, D.E., *The NON-VON Supercomputer,* Dept. of Computer Science, Columbia Univ., Aug. 1982.

[12] van de Snepscheut, J.L.A., *Mapping a Dynamic Tree on a Fixed Graph,* unpublished article, Feb. 1981.

# The Onset of Hot Spot Contention

Manoj Kumar and Gregory F. Pfister

IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598

**Abstract:** Non-uniform traffic distributions in a multistage network characterized by "hot spots" — destinations getting more than their share of traffic—can cause dramatic reductions in the maximum throughput of the network. In this paper we develop an analytical model predicting how long a "hot spot" must be persist in shuffle/exchange networks before its its full effect is felt. The model predicts that hot spots will disrupt network traffic severely in a very short time: 10 to 50 instruction execution times in a shared-memory machine. This result, verified by simulation, leads to the conclusion that if stringent measures are not taken to ensure uniformity, the performance of large multistage networks will be substantially worse than has been previously predicted.

## 1.0 Introduction

Multistage interconnection networks with distributed routing have often been proposed as a means of connecting large parallel or distributed computing systems. However, for such networks it has been shown that statistically non-uniform traffic patterns—patterns containing a **hot spot** [6] that gets more than its share of the traffic—can cause severe performance degradation for all network traffic, not just traffic to the hot spot. For example, as little as 0.125% imbalance in a 1000-way network can limit network throughput to less than 50% of its maximum value. This is independent of network topology, redundant paths, or mode of use of the network (e.g.: message passing, shared memory, circuit vs. packet switching, etc.). This effect was discovered in the IBM RP3 project [5,1,4], and first reported in [6]. It was also reported there that the technique of "combining" messages in the interconnection network could solve the problem for some cases of interest.

However, the analysis and simulation reported in [6] does not address a crucial issue: Over what time interval must a non-uniform pattern be sustained in order to reach tree saturation? This is important, because it is a critical measure of how uniform the traffic must be to avoid hot spot problems. Statistical uniformity is much more easily achieved when averaged over hours (for example) than when averaged over microseconds.

We address that issue here by developing a model, verified by simulation, of how long a hot spot must persist before its effects are fully felt. This provides a lower bound on the interval over which uniformity must be measured.

Unfortunately, the result is that the required interval is quite short indeed. For example, with a 1024-way network of 4-way switches containing 4-element queues, a 0.125% hot spot non-uniformity will have its full effect within (approximately) 10 to 50 times the minimum time to traverse the switch. (See Figure 4.)

We can only derive a crude lower bound for the time for a network to recover from a hot spot; that appears to be a more complex process. However, both that lower bound and our simulations demonstrate that the recovery time is much longer than the onset time.

Before deriving these results and comparing them with simulation, a brief overview of the hot spot effect will be given. A discussion of possible remedies, and of our conclusions, ends the paper.

## 2.0 Hot Spot Contention and Tree Saturation

Here we summarize the results presented in [6], with a slight addition.

Consider a two-sided packet-switched multistage network, with $p$ ports on each side, connected to message sources on one side and message sinks on the other, such as the Omega network [3] illustrated in Figure 1.

Suppose the traffic pattern is initially uniform, with messages emitted from each source at a rate $r$ $(0 \leq r \leq 1)$. Then, at some time after a steady state has been achieved, the traffic pattern is altered to direct a fraction $h$, $0 \leq h \leq 1$, of all references are aimed at a specific sink: the **hot sink.** I.e., each source emits $r(1-h)$ messages uniformly distributed, and $rh$ messages to the hot sink. $h$ is the hot spot rate. As a result, the hot sink receives two components of traffic: $r(1-h)$ from the uniform background, and $rhp$ from the hot spot.

If $h$ is large enough, the rate into the hot sink will be unity due to the $rhp$ term. If this happens, the queues in

0190-3918/86/0000/0028 $01.00 © 1986 IEEE

the network switch closest to that sink will fill. This causes the preceding switches' queues to fill; then the next preceding; etc. Finally, a tree of switches rooted at the sink and extending to every source is saturated. This is called **tree saturation**, and is illustrated by the marked switch queues in Figure 1.

Once tree saturation is in effect, every message from any source to any sink must cross the saturated tree and so is delayed. In effect, all the network traffic is gated by the speed at which the single hot sink can dispose of its messages. In the steady state, this occurs when the total traffic rate into the hot sink ($r(1 - h) + rhp$) equals unity. In [6] it was shown that this has a dramatic effect as the system is scaled up in size, as noted in the examples cited in the present paper's introduction. In the steady state, hot spot effects are independent of network topology, finite queue size, etc. (However, the timing analysis presented here does depend on these factors.)

Beyond what was presented in [6], we note here that tree saturation is a finite queue effect. But in order to eliminate it, the queues in the final stage of the network must be large enough to accommodate the maximum hot spot traffic of the entire system. In other words, their size must be equal to $M$ × the number of network ports, where $M$ is the maximum number of messages that can be simultaneously outstanding from each source. Thus if queue sizes are taken into account, the total network size has another factor equal to the number of network ports. This raises the network size to $O(M \times N^2 \log N)$, rather than the usually-cited $O(N \log N)$. The factor of $N^2$ negates any size advantage over a full crosspoint switch.

## 3.0 Modelled Behavior

The network behavior modelled here assumes that the network is initially in a steady state with uniformly distributed traffic flowing through it at a rate $r$. At time 0, all the sources simultaneously change their traffic patterns to include a hot spot. $r$ does not change, but a fraction $h$ of $r$ sufficient to cause tree saturation is now directed at a hot sink. The throughput of the network now declines until at a time $T$ it reaches a steady-state minimum caused by tree saturation. We wish to estimate T as a function of r, h, network size, etc.

Packet switching is assumed, with one packet per message. In the analysis, one time unit is required for a packet to move from one switch to the next. For reasons explained later, the results shown in the figures are scaled to be in units equal to the minimum time to traverse the switch.

## 4.0 Model of Onset

It is convenient to imagine the messages sent to the hot sink after time 0 as being colored red, and all other messages colored white. The total rate at which each source emits red messages is

$$R = rh + \frac{r(1 - h)}{p}$$

$R$ is not simply $rh$ because $1/p$ of the messages from the uniform background are sent to the hot sink, where $p$ is the number of network ports.

Rather than dealing directly with the complex dynamics of message flow through the network, we will count the red messages in the network. On the one hand, the number $N$ of red messages is a function of time and of their total arrival and departure rates from the network as a whole. On the other hand, when tree saturation is reached there is a steady-state number $N$ of red messages in the network that is a function of the input traffic and the amount of buffering available. So equating $N$ in both formulations can tell us how long it takes to reach saturation.

### 4.1 Arrival and Departure

To estimate the arrival and departure rates, we make a set of assumptions that, overall, amount to the general assumption that the onset of tree saturation happens fast and suddenly — too fast and suddenly for the internal dynamics of the network to have much effect on arrival and departure rates until the point of saturation is reached.

We assume that the total arrival rate of red messages is constant (i.e., is $Rp$) until tree saturation is reached; and then it drops instantly to the tree saturation value. Under this assumption, the number of red messages generated by time $T$ is simply $RpT$. Our simulations do not verify a constant arrival rate: the input rate does decline with time. Nevertheless the final results are adequate.

For the departure rate, we assume the following:

At $T$, the first red messages enter the first switch stage. They make their way through the tree of switches that will be saturated, gradually becoming more and more concentrated, until they reach the hot sink. Then:

1. Until the hot sink is reached, there is no effect on the rate at which messages are transported. The first red messages reach the hot sink at a time $D(r)$ that equals the average delay through the network at a total input rate of $r$.
2. At D(r), the concentration of red messages is immediately sufficient to saturate the hot output port; i.e., after D(r) that port emits messages at a rate of unity.
3. After D(r), all the messages emitted by the hot output port are red.

While somewhat unrealistic, these assumptions are conservative. They overestimate the departure rate, and thus indicate that the saturated tree fills with red messages sooner than it actually will.

With those assumptions, the number of messages that have left the network at time $T$ is simply $T - D(r)$. Then, since the number of red messages arriving by time $T$ is $RpT$, $N = RpT + T - D(r)$. Solving this for $T$ yields

$K$ is the size of each individual switch, i.e., number of input and output ports, so $K^i$ is the number of switches at each stage that lie within the saturated tree. $q$ is the size of each queue; (K + q) is the total storage available for messages aimed at the hot spot in each switch stage. (The additional K is due an additional buffer on each input port used in the simulation; this is discussed later).

For each switch at stage i in the saturated tree, $mix_i$ is the fraction of red messages in its queue during steady-state tree saturation:

$$mix_i = \frac{AR_i}{AR_i + AW_i}$$

$AR_i$ and $AW_i$ are respectively the arrival rates of red and white messages at stage i:

$$AR_i = R \times \frac{p}{K^i}$$

because $p/K^i$ is the number of sources in the subtree leading to each switch in stage i.

$$AW_i = r(1 - h) \times \frac{K^i - 1}{K^i}$$

Recall that $r(1 - h)$ is the total rate of uniform background traffic, part of which is directed at the hot sink. Since there are $K^i$ possible destinations for the cool traffic

$$T = \frac{N - D(r)}{Rp - 1}$$

The denominator becomes zero as the combinations of R and p reaches a point inadequate to sustain an output rate of unity. I.e., as that point is approached the time to saturation approaches infinity, which is expected.

To estimate $D(r)$ we use the well-known formula for the average queue length in a switch stage [2]:

$$B(r) = \frac{r}{2(1 - r)}\left(1 - \frac{1}{K}\right)$$

Where r is the steady state rate and K is the number of ports of each switch in the network. Then $D(r) = S(1 + B(r))$. Because it assumes infinite queues, $B(r)$ will again produce conservative results (faster transportation than reality) for high total rates $r$.

## 4.2 Steady-State Population

In the steady state of tree saturation, $N$ is the sum of the number of red messages $n_i$ at each network stage (we count stages from 0, starting at the stage closest to the sinks):

$$N = \sum_{i=0}^{S-1} K^i \times (K + q) \times mix_i$$

at stage i, and one of them is the hot sink, the fraction above follows.

If we substitute back into the expression for $mix_i$, substitute the original expression for $R$, and simplify, we obtain

$$mix_i = \frac{Hp + 1}{Hp + K^i}$$

where $H = h/(1 - h)$, the ratio of hot to cool packet generation.

Finally substituting back into T, with slight simplification, we get

$$T = \frac{\left((K + q)\sum_{i=0}^{S-1} K^i mix_i\right) - S(1 + B(r))}{Rp - 1}$$

## 5.0 Simulation

To verify the above model, we ran simulations of the situation described above for a number of cases. These results are plotted with the analytical predictions in Figure 2 through Figure 4.

The switches simulated had two non-standard characteristics that match those of [6], and serve to make the simulated network act more like the ideal network modelled:

1. Each output queue can simultaneously accept $K$ messages in one time unit. While fairly realistic for $K = 2$, this is undoubtedly unrealistic for larger switches.
2. Each input port to a switch has an additional one-message lookaside buffer that is not counted in the queue size. This allows the queues to be more fully utilized, since without it there must be $K$ empty positions in every queue of a switch for any of the switch's predecessors to be enabled to send messages. This is the source of the additional $K$ buffers per switch that was included in the prior analysis.

A complication arose in deciding what to measure as the time at which the switch reaches saturation. Our formula effectively assumes that all the queues in the saturated tree fill up simultaneously, and this is clearly not the case. What we did was find, from the simulations, the average red message occupancy of the queues in steady-state tree saturation. Then the time to saturate was taken to be the time at which 80% of that steady-state value was reached. 80% was chosen because at approximately that value there is a single message slot unused in each queue.

All plotted points are the means of 200 simulation runs each.

## 6.0 Results

A surprisingly short amount of time is required to reach tree saturation.

Figure 2 shows the time to tree saturation as a function of the initial uniform background rate $r$ for values of $h$ ranging from 0.125% to 16% in factors of two. It assumes a 64-way Omega network where each 2-way switch has queues of size 4. The time unit is not $T$ as derived above, but rather $T/S$, the minimum time required for a message to traverse the switch in one direction; in our formulation, that equals the the depth of the network. This unit was chosen for two reasons: First, it allows meaningful comparison across different network and switch sizes; it turns out that, when expressed in this unit, the time to saturate is relatively constant across network and switch sizes (10-50, for 4 element queues). Second, in a shared memory system, it is typically comparable to the time required to execute a single instruction in a processor. (It is not identical to the time required to perform a complete memory reference; that time includes two trips across the network—request and reply—as well as memory access time and other delays.)

The dotted lines at the top of the figure mark the minimum background rate below which each plotted hot spot rate will not cause tree saturation; this is equivalent to the maximum rate sustainable with the associated hot spot rate. Thus the graphs can be interpreted as follows: Pick a given initial background rate (point on the lower axis). Proceed vertically to the curve corresponding to the hot spot rate of interest. The time that curve indicates (shown on the vertical axis) is the amount of time required for that hot spot rate to drive the network throughput from the initial rate to the asymptote associated with the hot spot rate (dotted line).

Figure 3 shows the same information, but in this case for a 256-port network with 4-way switches. Figure 4 shows the predicted results for a large (1024 port) network; this was not simulated. As can be seen, the onset time is very short.

As can be seen, our predicted results match the simulated results reasonably well except for two situations: very high $r$ in the 256-port network; and very low $r$ in the 64-port network. At high $r$ in the 256-port network we are pushing the actual maximum capacity of the network and would expect all the approximations we are using to break down. At low $r$ in the 64-port network, there is a breakdown in our assumptions about total arrival and departure rates: Onset is a more diffuse process with more time for complex internal feedback effects. However, in this case our estimates are on the optimistic side.

## 7.0 Recovery Time

Figure 5 shows the throughput of a 64-way network as a hot spot of 16% is applied and then removed within a total background rate of 0.4. As can be seen, the recovery time is substantially longer than the onset time. A rationale for this follows.

Intuitively, many sources "cooperate" to saturate the tree; but only one sink (the hot one) operates to eliminate saturation. The time to recover "normal" traffic flow should be related to the time to remove all red messages from the switch after the sources stop generating them. If the hot output port runs at the maximum rate (unity), this time is simply equal to the steady-state count of red messages $N$ during tree saturation, derived previously. This is longer than the time to saturation, since the onset time $T$ is $(N - D(r))/(Rp - 1)$. One might imagine that after a time $2mix_{S-1}N/K$, for K-way switches, all the switches closest to the sources would be clear; so after that time, $1 - 1/K$ of the traffic would resume its normal rate. Then after an additional time equal to $2mix_{S-2}N/K$, $1 - 1/K^2$ of the traffic will resume its normal rate; etc. The slow rise of throughput shown in Figure 5 tends to indicate that something like this is occurring. However, we have not yet compared this to simulation results, and consider it unlikely to be correct, for the following reason: Until all the red packets are gone, they should tend to delay uniformly distributed messages that happen to be directed to the hot sink; and by filling queues, this will affect other messages. This may cause continued congestion even after all the red messages have left the network.

## 8.0 Discussion

The time required to reach tree saturation is distressingly short. What points of leverage can be used to improve the situation?

Increasing the switch size ($K$) actually makes the situation worse: Even when, as we have done, the units are the depth of the network—giving larger switches a large advantage—smaller switches saturate more slowly.

Increasing the queue size also helps. But the queue size is a linear factor in the total saturation time, so very large queues are necessary to make a substantial difference. E.g., to get saturation time up to the range of 100 switch traversals, queues on the order of 40 elements are needed. This is unreasonable with present technology.

The addition of redundant paths will also help, but only because the total queue storage available rises with the number of paths. So this decrease is also linear.

One thing that certainly can help is over-design, in the sense of using the network only at at traffic rates less than the rates where the expected hot spot activity will cause tree saturation. This adds significantly to the expense of the network. Furthermore, at present there is little experience available to define exactly what degree of non-uniformity to expect in general.

As discussed in [6], "combining" of identical messages within the switch nodes themselves can eliminate the problem completely. However, combining only works when the the hot spot is caused by references to identical entities at the sinks (e.g., identical memory locations).

When the hot spot occurs because many sources are accessing many different entities that happen to occupy the same sink, combining cannot help. What may help in that case are techniques to ensure that non-identical references are scattered uniformly among the destinations, such as the combination of interleaving and randomization used by RP3 [5]. How well this will work in practice is not yet known; if it does, simultaneously using both this technique and "combining" (also present in RP3) may solve the problem, at least for shared memory systems. It is not obvious at this time how to solve the problem for systems based on other computational models.

Global control over routing can avoid the problem completely, of course, as discussed in [6].

To summarize:

1. Very little perturbation, over a very short time, can drastically reduce network throughput.
2. Recovery after the perturbation takes much longer than than the onset of the problem.

This leads to the conclusion that multistage networks with distributed routing are unstable under non-uniform traffic loads, in the sense that they tend to "fall into" tree saturation easily. For large networks in particular, e.g., networks of size 512 or greater, if stringent measures are not taken to maintain a uniform traffic pattern, swift onset and slow recovery makes it very probable that at least partial tree saturation will always be present; and thus large multistage networks may not perform anywhere near as well as has previously been predicted.

## References

[1] W.C. Brantley, K.P. McAuliffe, J. Weiss, "The RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference in Parallel Processing,* August 1985, pp. 782-789.

[2] C. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. on Computers,* Vol. C-32(12), 1983, pp.1145-1155.

[3] D. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans. on Computers,* No. C-24, 1975, pp. 1145-1155.

[4] V.A. Norton and G.F. Pfister, "A Methodology for Predicting Multiprocessor Performance," *Proceedings of the 1985 International Conference in Parallel Processing,* August 1985, pp. 772-781.

[5] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference in Parallel Processing,* August 1985, pp. 764-771.

[6] G.F. Pfister and V.A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Trans. Comp.,* Vol. C-34, No. 10, October 1985, pp. 934-948; also *Proceedings of the 1985 International Conference in Parallel Processing,* August 1985, pp. 790-797.

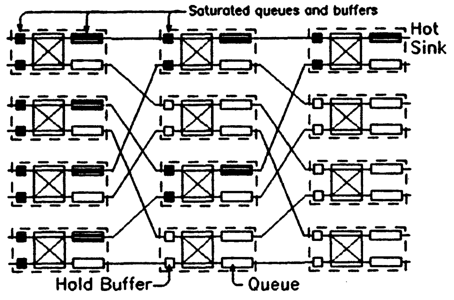


Figure 1 : An 8x8 shuffle-exchange network, depecting the saturation caused by a hot sink.

Figure 1. An 8-port Omega network. The marked switches are saturated, as discussed in the text.

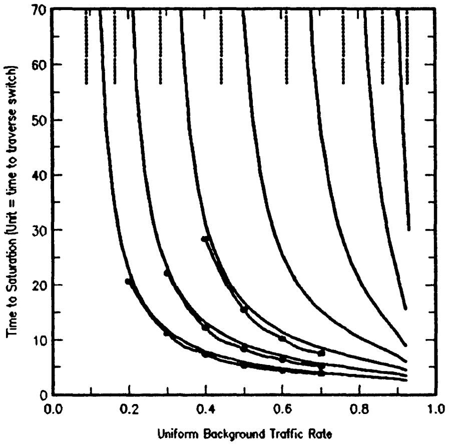


Figure 2. Time required to saturate a 64-port Omega network: The switches have queue size 4, and 2 inputs and outputs. The solid curves are the predicted values for *h* ranging from 0.125% to 16% by factors of 2. the dots connected by dashed lines show simulation results. The dotted lines show the sustainable throughput after tree saturation for the values of *h* used.

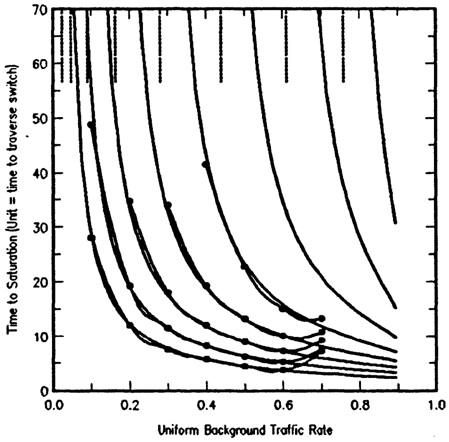


Figure 3. Time to saturate a 256-port Delta network: The switches have queue size 4; other elements are the same as Figure 2.

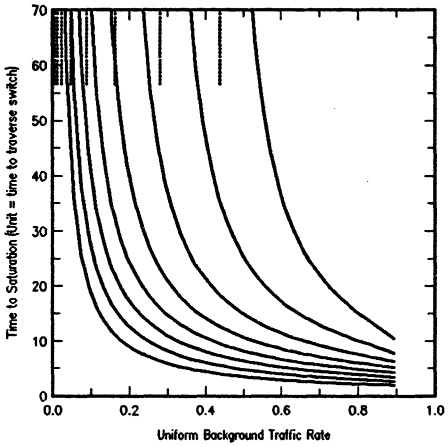


Figure 4. Time to saturate a 1024-port Omega network: Information and are the same as Figure 2.

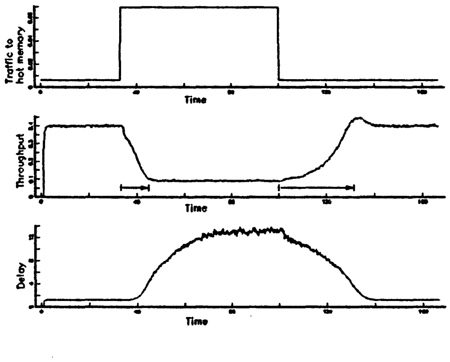


Figure 5. Onset and Recovery from a Hot Spot: Hot spot percentage, throughput, and delay as a function of time. The network is the same as Figure 2's, and the hot spot is 16%. The arrows indicate the time for onset and recovery.

# The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot Spots'

*Gyungho Lee*

Center for Supercomputing
Research and Development
University of Illinois
Urbana, IL. 61801

*Clyde P. Kruskal*

Department of Computer Science
University of Maryland
College Park, MD. 20742

*David J. Kuck*

Center for Supercomputing
Research and Development
University of Illinois
Urbana, IL. 61801

## Abstract

Concurrent requests to a shared variable by many processors on a shared memory machine can create contention that will be serious enough to stall large machines. This idea has been formalized in the "hot spot" traffic model [PfNo85], where a fixed fraction of memory requests is for a single shared variable. "Combining," in which several requests for the same variable can be combined into a single request, has been suggested as an effective method of alleviating this contention. The NYU Ultracomputer [GGKM83] and the IBM RP3 [PBGH85] machine use "pairwise" combining, in which only two requests for the same variable can be combined at a switch. We study the effectiveness of combining. In particular, it turns out that pairwise combining cannot handle hot spots if the machine size is large enough. We suggest ways to overcome this weakness.

## 1. Introduction

The popularity of shared memory parallel computers, where processors and memory modules are interconnected through a multistage network, can be seen in several current projects, including the University of Illinois Cedar machine [GKLS83] [KDLS86], the NYU Ultracomputer [GGKM83] [EGKM85], and the IBM RP3 machine [PBGH85]. Sharing the memory in a parallel computer suggests that there is a possibility of many processors requesting the same variable at the same time (*concurrent requests*). This can create congestion in a machine, and the congestion becomes more serious as the number of processors in the machine (*machine size*) increases. To reduce congestion, when several requests directed at the same shared variable meet at a switch, they can be combined into a single request, which is forwarded toward the shared memory. When the response from the memory returns, the switch satisfies all of the requests, one at a time. The idea of reducing the congestion in this way, known as "combining," has been suggested as an effective way of allowing concurrent requests to a common location; combining can be found in the Columbia CHoPP [SuBK77], the NYU Ultracomputer, and most recently the IBM RP3 machine. (See [KrRS86] for a general discussion of what types of memory requests can be combined.)

Pfister and Norton [PfNo85] suggested that the effectiveness of combining could be studied with the "hot spot" traffic model: a fixed fraction of the total memory traffic is concurrent requests to a single shared variable. Hot spots capture the effect of all of the processors continually accessing a common variable. Pfister and Norton argue that hot spots will seriously degrade the performance of any machine that lacks combining, and that this effect is quite general. They also discuss how well the hot spot model captures reality.

In this paper, we study the effectiveness of several different combining schemes. In particular, we will see that the pairwise combining scheme used in the NYU Ultracomputer and in the IBM RP3 machine is not powerful enough to handle hot spots. We suggest ways to modify their designs in order to overcome this weakness.

## 2. The Model

There have been many studies on the performance of multistage interconnection networks for processor-memory connection (see [Sieg85] and the references therein). One common traffic model for these studies is: A stream of memory requests from each processor is an independent, identically distributed random process; each processor's requests are uniformly distributed to all of the memory modules. This *uniform* traffic model does not capture the effect of traffic with requests to a single shared variable. To represent such traffic patterns, we use the *hot spot* model [PfNo85]: each request has a (finite) probability $q$ of being headed to the same shared variable. The hot spot model is *nonuniform* in the sense that the requests are not uniformly distributed onto the memory modules. There are two types of request streams: the *noncombinables*, which are uniformly distributed to the memory modules as in the (usual) uniform model, and the *combinables*, which are headed to the same shared variable (and hence the same memory module).

We consider a buffered square banyan network [GoLi73] as the multistage network for interconnecting the processors and the memory modules. Square banyan networks include Omega networks [Lawr75] and Delta networks [Pate81]. (For details and general characteristics of multistage networks, see, for example, [Feng81], [KrSn82], [Sieg85].)

A network is composed of $n$ stages of $2 \times 2$ (crossbar) switches with FIFO queues (i.e. buffers) at each output port. We assume that the network is packet-switched and synchronous, so that packets can be sent only at times $t_c$, $2t_c, \cdots$, where $t_c$ is the *network cycle time.* Without loss of generality, we assume $t_c = 1$. We make the following further

This work was supported in part by the National Science Foundation under Grant No. US NSF DCR84-10110, and the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001. Part of this work was done while the second author was at the University of Illinois.

0190-3918/86/0000/0035 $01.00 © 1986 IEEE

assumptions:

- Each request is a single packet.
- Each queue can accept at each cycle up to two distinct requests, one from each input port. If at some cycle a queue has only one free location and two requests are directed to it, the queue randomly accepts one of the two (the other request remains on the queue of the previous stage).
- The enqueuing process of a request and the dequeuing process are overlapped (i.e. while the request in front of the queue, if there is one, is being removed, other requests can be inserted onto the queue).
- The service time of a request in a queue is the same as the cycle time. So, the delay of a request at a switch is the number of requests ahead of it in the queue.
- Each processor has an infinite queue for requests. If a request is blocked from entering the first stage it is placed on the queue, and the processor continues issuing requests.

A square banyan network has a complete tree leading from the processors to each memory module (Figure 1). The tree that combinable requests traverse will be called the *fan-in tree*. Our main concern is with the average queuing delay in the fan-in tree.

*Combining* works as follows: When several combinable requests meet at a switch they are combined into a single request, which is forwarded toward the shared memory. A record of this is kept at the *wait buffer*. When the response from the memory returns, the switch satisfies all of the requests, one at a time (and the record is removed from the wait buffer). To concentrate our attention on queuing delays, we assume in Sections 2-6 that wait buffers have infinite size. Also, we will consider the delay of a request only from the processors to the memory modules, temporarily ignoring the delay on the return trip. Section 7 considers finite wait buffers, and their effect on the delay of a request in both directions of the network.

We will distinguish queue size and queue length. *Queue size* is the number of requests a queue can store at one time. We use *infinite queue* to mean that the queue size is infinite, and *finite queue* to mean that the queue size is finite. *Queue length* is the number of requests stored on a queue at some particular time. We will use equivalent definitions for wait buffer size and wait buffer length.

We consider several different combining schemes. In each case, we will consider what happens both with finite queues and with infinite queues. Infinite queues provide a nice yardstick to compare the more practical finite queue schemes. For finite queues, unless otherwise specified, we will always consider queue size four. This is large enough so that for the traffic loads considered, the performance under uniform traffic is almost as good as with infinite queues.

A network is *stable* if in steady state average delays in the network are uniformly bounded. This is an important property of a network. Under the *uniform* traffic model, buffered multistage interconnection networks are generally believed to be stable for "light" traffic (see [KrSn83],[KrSW86]).

We assume that at each cycle each processor issues a request with probability $r$, i.e. $r$ is the rate of requests. Each request has probability $q$ of being a combinable request. Let $r_c$ be the rate of combinables (i.e. hot spot requests), and $r_n$ be the rate of noncombinables. Then

$$r_c = qr \qquad \text{and} \qquad r_n = (1-q)r \ .$$

Let $r^i$ be the rate of requests at the stage $i$ of the fan-in tree ($r^0 = r$). Let $r_c^i$ and $r_n^i$ be the rate of combinable requests and noncombinable requests, respectively, at the stage $i$ of the fan-in tree ($r_c^0 = r_c$ and $r_n^0 = r_n$).

## 3. No Combining

In this section we will consider the performance of systems without combining. It is obvious that the combinables will create congestion in the network. The question is, how much will this degrade performance?

### 3.1. Infinite Queues

Assume the queues have infinite size and there is no combining. Recall that the rate of requests at the first stage is $r_n = r(1-q)$ and $r_c = rq$. Since there is no combining, the rate of combinable requests keeps doubling at each stage approaching the root of the fan-in tree. In particular, the rate of requests at stage $i$ will be

$$r^i \ = \ r_n + 2^i r_c \ .$$

For any finite value of $r_c$, after several stages, the requests will be arriving at each queue at a greater rate than the queue can forward them. Networks large enough to see this effect will be unstable. For example, consider the case of $r = 0.25$ and $q = 0.01$. Even with $q$ so small, by the ninth stage the arrival rate of the combinables alone will be $r_c = 1.28$, so the queuing delay will be unbounded.

In practice one expects short intensive periods of "hot spot" contention. If there are not too many stages of the fan-in tree in which the rate of requests is greater than one, the system may still provide acceptable performance. Only the combinable requests will suffer extraordinary delays, along with the relatively few noncombinable requests traversing the fan-in tree near its root.

### 3.2. Finite Queues

With finite queues the situation is worse. Pfister and Norton [PfNo85] noticed a very interesting phenomenon they call *tree saturation*. When the queue at the root of fan-in tree becomes full, the two queues feeding it can no longer send requests to it. They too will become full and stop the four queues feeding them from sending requests. Eventually the entire fan-in tree will consist of full queues. All of the queues at the same level of the fan-in tree can together satisfy combinables only at the same rate as the root satisfies them. In other words, at the $i$th level from the root, each queue can satisfy combinables only at a rate $1/2^i$ as fast as the root does. So, although each queue at this level has on average only $1/2^i$ as many combinables as the root, with respect to combinables the queues are not progressing any faster. Thus, progress of the whole system is governed by the service rate at the "hot spot"; noncombinables will suffer delay proportional to the queue size on each stage of the fan-in tree traversed.

Kumar and Pfister [KuPf86] have observed that a relatively short period of hot spot contention will produce tree saturation. Furthermore, after the processors stop issuing hot spot requests, the network takes a long time to return to normal.

## 4. Pairwise Combining

Ideally, one would like to combine all of the combinables that reside concurrently on a queue. This, however, makes the combining process complicated, and also creates congestion at a wait buffer when the response returns from memory. To simplify the combining process and to avoid contention at the wait buffer, the NYU Ultracomputer and IBM's RP3 machine support combining only a pair of requests at a switch. This section studies the effectiveness of such *pairwise* (or *two-way*) combining.

### 4.1. Infinite Queues

We did simulations to check the effectiveness of pairwise combining with infinite queues. Our concern is whether congestion at the hot spot still occurs. (Recall that $r_c^i$ is the traffic load of combinables from each input port of a switch at stage $i$ of the fan-in tree.) In our experiments, $r_c^i$ increased rapidly until $r_n + r_c^i$ reached 1.0 (see Figure 2). This shows that with pairwise combining and infinite queues large networks are unstable.

The reason for congestion even with combining is that a combinable request does not always encounter another combinable request to combine with. Whenever a combinable request does not combine, it will be added to the traffic of the combinables coming out of the queue. Thus, the rate of combinables will necessarily increase towards the root of the fan-in tree. It is conceivable that this rate approaches some limit less than $1\text{-}r_n$, in which case the network would be stable. However, our experiments show this simply does not happen: the rate of combinables increases without bound.

### 4.2. Finite Queues

It may seem *a priori* that finite queues will always provide worse performance than infinite queues, since infinite queues have more storage capacity. However, this is not necessarily so: Suppose at stage $i$ of the fan-in tree, a queue becomes full. Then, the two queues at stage $i$-1 of the fan-in tree feeding this queue will become blocked (at least for requests destined to the full queue). This will increase the chances of these two queues becoming full, thereby blocking the queues at stage $i$-2 that feed them, and so on. Thus, if the rate of requests is large enough to create congestion at the root of the fan-in tree, the whole fan-in tree will tend to become congested. The overall affect on a message traversing the fan-in tree will be that its total delay will be fairly large at every stage, which contrasts with infinite queues where the delay is large only near the root. This means that combinables will spend more time near the leaves of the fan-in tree, and therefore have more chance of combining near the leaves. This will reduce the traffic rate of combinables which in turn will improve the overall performance of the network. (Recall that with pairwise combining, if a combinable traverses a stage without combining, it increases the rate of combinables for all later stages.)

We performed simulations on networks of nine stages with finite queues and pairwise combining. The queue size was assumed to be four. For the traffic of $r = 0.6$ and $q = 0.1$, we observed tree saturation: the average waiting times at each stage of the fan-in tree was approximately equal to the queue size. Waiting times of requests at each processor's queue seem to increase without bound as the number of network cycles simulated increased. Although we did not observe tree saturation for lower traffic loads, we expect that it would occur in larger machines. (See Figure 3.)

Since the probability of combining increases as a combinable request stays in a queue longer, larger sized queues should help combining, which in turn can help avoid tree saturation. One might think that the tree saturation reported here conflicts with the results of Pfister and Norton [PfNo85], where pairwise combining was effective in handling hot spots with queue sizes of only four. Although there were some minor differences in our two models, which could account for the different results, the main difference was that they were simulating a network with only six stages. We believe that adding a few more stages to their network would produce tree saturation and make their network unstable. Minor changes in switch design cannot overcome the inherent weakness of pairwise combining, at least not without making the delays at each stage of the fan-in tree unacceptably long.

## 5. Unbounded Combining

*Unbounded combining* allows any number of combinables to be combined into a single request at a queue. Although the combining of the Columbia CHoPP is very similar to unbounded combining, our study is not directly applicable CHoPP because of its "repetition filter memory", which allows the combining of incoming requests with requests already in the wait buffer.

We have done extensive simulations of networks with infinite queues and unbounded combining. The networks seem to be stable and provide reasonable delay irrespective of the machine size and the traffic load. The traffic of the combinables adds only slightly to the average queuing delay of the noncombinables alone. It seems that unbounded combining eliminates the contention on the fan-in tree because there can be at most only a single combinable request waiting in a queue at any given time.

Simulations show that with unbounded combining, finite queues provide only slightly larger delay than do infinite queues. When compared to infinite queues, delays are just about the same at the first few stages and slightly larger at all the later stages.

## 6. Bounded Combining

We have so far considered two extreme combining schemes: unbounded combining and pairwise combining. Unbounded combining provides good performance, but seems to be expensive (even to approximate); pairwise combining suffers from tree saturation, but is *relatively* easy to implement. We suggest a compromise scheme, *bounded* combining, where more than two, but at most a predetermined constant number of, combinables can be combined into a single request at a queue; in *k-way* combining the bound is $k$. Bounded combining is easier to implement than unbounded combining; the hope is that it will provide approximately the same performance. The question is, how large does $k$ have to be?

In the experiments with unbounded combining, we observed that a combinable request coming out of the switches at the later stages represents on average only slightly more than two combinables. This suggests that *three-way combining,* i.e. at most three combinables can be combined into a single request at a queue, will be effective. Simulations show that three-way combining performs almost as well as unbounded combining for both finite and infinite queues (see Figure 4). This indicates that pairwise combining may be slightly too restrictive with respect to the number of combinables it supports.

## 7. Wait Buffers and Return Queues

Up until now we have considered the delay of a request only from the processors to the memory modules. For the return trip, there must be two *return queues* exiting each switch passing responses from the memory modules towards the processors. The performance of a network will be sensitive to the size of these return queues. We have assumed that the size of wait buffers is infinite. This is unrealistic in practice. The wait buffer size is an important factor for good performance, because combining cannot take place if the wait buffer is full.

A combining of $k$ requests is represented as $k$-1 pairwise combinings, i.e. it uses $k$-1 wait buffer locations. When the response returns from memory, all $k$-1 locations are immediately freed and the $k$ response messages are placed on the return queue.

This section considers the effect of wait buffer size and return queue size on queuing delay. Our main concern is to determine the proper size of wait buffers and return queues for three-way combining to obtain performance close to that of unbounded combining with infinite return queues and infinite wait buffers.

### 7.1. Infinite Queues, Returns Queues, and Wait Buffers

To get an idea of the appropriate size of wait buffers, we measured the average length of the buffers assuming infinite queues, return queues, and wait buffers. Although the unbounded and three-way combining schemes avoid congestion by inserting more combinables into the buffer at a time than pairwise combining, our simulations show that the average length of the buffers with pairwise combining is actually unbounded while it is quite moderate with three-way combining (see Figure 5). The reason for this is that the average length of the buffers is proportional to queuing delays.

Suppose combining takes place in a switch at stage $i$ of an $n$-stage network. Then, a record of the combining will remain in the wait buffer until a response from the memory arrives at the switch some time later. So, the average length of the wait buffer is determined by the average number of combinings and the average number of cycles until the memory responds. Let $c_i$ be the average number of combinings per cycle at the switch and $t_i$ be the average response time from memory (to the switch) for a combinable request. Then, the wait buffer is a queuing system with arrival rate $c_i$ and service rate $1/t_i$. The arrival rate $c_i$ is determined by the traffic load and the position ($i$) of the switch in the network. The service rate is determined by the queuing delays at stage $i$ and later stages. Given fixed traffic load and fixed network size, the average length of the wait buffer will be unbounded if there is severe enough congestion at later stages for $c_i \geq 1/t_i$.

Since the service rate $1/t_i$ is smaller for switches closer to processors, one may worry about the average length of the wait buffers at earlier stages. However, this is counterbalanced to some extent by the fact that there is less contention in the earlier stages so that fewer combinings take place. Notice that the wait buffer lengths become unbounded as the network size increases, for any fixed arrival rate $c_i$ at the wait buffer, irrespective of the combining scheme. The wait buffer size needs to grow with the network size.

### 7.2. Finite Queues, Return Queues, and Wait Buffers

To see the effect of small wait buffer sizes, we did simulations with queues of size four, infinite return queues, and wait buffers of size six. As can be seen in Figure 6, three-way combining with "small" wait buffers performs as badly as pairwise combining does. The reason is that the buffers at the later stages are almost always nearly full, and three-way combining effectively changes to pairwise combining.

To see the effect of small return queue sizes, we did simulations with queues of size four and infinite wait buffers. It turns out that, for three-way combining, return queues of size four are not large enough to provide good performance. This may seem surprising, since (forward) queues of size four are sufficient, and the responses are just returning along the same path that the original request traversed. The reason is that on the return path combinables are returning in bursts, since a combinable response can split into two or three responses. Thus, each return queue in a switch is effectively a queuing system with the same traffic intensity as the (forward) queue in the same switch, but with fewer, larger-sized packets. The former system will provide worse performance and require larger queue sizes (see [KrSW86]). With three-way combining, return queues of size of ten obtained approximately the same performance as infinite return queues.

In our experiments for moderate traffic loads, return queues of size ten and wait buffers of size fifteen seem to be large enough to obtain performance close to that of unbounded combining with infinite sized queues and wait buffers (see Figure 7). Neither return queues of size eight and wait buffers of size fifteen nor return queues of size ten and wait buffers of size ten produced good performance.

## 8. Conclusion

Shared memory machines have the potential of congestion due to concurrent requests to a shared variable. Since hot spot contention becomes more serious as the machine size grows, congestion can severely degrade the performance of "large" machines. To avoid potentially serious congestion, pairwise combining was suggested in the NYU Ultracomputer and IBM RP3 machine as an effective way of eliminating congestion.

We studied the hot spot traffic model, where a fixed fraction of the total memory traffic is for a single shared variable. As observed by Pfister and Norton [PfNo85], large networks with finite queues and no combining suffer from tree saturation. With finite queues, even pairwise combining has the potential of tree saturation creating unbounded delay no matter how "light" the traffic load is, for large enough machines. If hot spots are a real-life phenomenon, pairwise combining as suggested for the NYU Ultracomputer and the IBM RP3 machine is too restrictive. Three-way combining resolves the congestion. It remains to be seen whether three-way combining can be realized efficiently in hardware.

A combining network must be carefully balanced. There are many parameters: the network size, the boundedness of the combining, the queue size, the wait buffer size, the queue size on the return path, etc. It is not obvious how any particular choice of these parameters will behave. For example, we have seen that changing finite queues to infinite queues, which one might expect would improve performance, can actually degrade performance.

One must be very careful in interpreting our results. We do not believe that processors are likely to concurrently access the same shared (synchronization) variable for extended periods of time. If hot spots are only transient, i.e. if there are short, intensive periods of hot spot contention, pairwise combining may very well combine enough to provide acceptable performance. One might consider the (steady state) hot spot model suggested by Pfister and Norton to be a conservative worst case scenario.

We have restricted our attention to square banyan networks composed of 2×2 switches, and to messages of length one. We believe our results generalize to other network topologies, other switch sizes, and other message size distributions. Any interconnection network will have to have a tree, maybe implicitly, leading from every processor to any given memory module. This will create the possibility of tree saturation when there are hot spots, but also the opportunity for combining. With $k \times k$ switches, it seems that $k$-way combining is not enough; some fraction slightly higher than that will be necessary. Longer messages seem to increase the amount of combining, but not enough to avoid tree saturation with only pairwise combining.

### Acknowledgements

The authors thank Alan Norton and Greg Pfister for helpful comments.

### References

[DiJu81] D. M. Dias and J. R. Jump, "Packet Switching Interconnection Networks for Modular Systems", *IEEE Computer,* Vol.14 No.12 1981

[EGKM85] J. Edler, A.Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Willson "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach", *The 12th Annual International Symp. on Computer Architecture,* June, 1985, pp.126-135

[Feng81] T. Y. Feng, "A Survey of Interconnection Networks", *IEEE Computer,* Dec. 1981

[GKLS83] D. D. Gajski, D. J. Kuck, D. Lawrie, and A. Sameh, "Cedar – A Large Scale Multiprocessors", *Proc. of the 1983 International Conf. on Parallel Processing,* Aug. 1983

[GoLi73] G. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems", *The 1st Annual Symposium on Computer Architecture,* 1973

[GGKM83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. M. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - designing an MIMD shared memory parallel computer", *IEEE Trans. on Computers, Vol. C-32, No. 2, 1983*

[KrRS86] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory", *The 5th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing,* Aug. 1986

[KrSn82] C. P. Kruskal and M. Snir, "Some Results on Multistage Interconnection Networks for Multiprocessors", NYU Ultracomputer note 41; also in *Proc. 1982 Conf. Informat. Sci. Syst.,* Princeton Univ., Princeton, NJ, Mar. 1982

[KrSn83] C. P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", *IEEE Trans. on Computers,* Vol. c-32, No. 12, Dec. 1983

[KrSW86] C. P. Kruskal, M. Snir, and A. Weiss, "The Distribution of Waiting Times in Clocked Multistage Interconnection Networks", *Proc. of the 1986 International Conf. on Parallel Processing,* Aug. 1986

[KDLS86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach", *Science,* Vol. 281, Feb. 28, 1986

[KuPf86] M. Kumar and G. Pfister, "The Onset of Hot Spot Contention", *Proc. of the 1986 International Conf. on Parallel Processing,* Aug. 1986

[Lawr75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans. on Computers,* Vol. c-24, 1975

[Pate81] J. A. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors", *IEEE Trans. on Computers,* Vol. c-30, 1981

[PBGH85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. of the 1985 International Conf. on Parallel Processing,* Aug. 1985

[PfNo85] G. F. Pfister and V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks", *IEEE Trans. on Computers,* Vol. c-34, No.10, 1985

[Schw80] J. T. Schwartz, "Ultarcomputers", ACM TOPLAS, 1980, pp. 484-521

[Sieg85] Howard Jay Siegel, *Interconnection Networks for Large-Scale Parallel Processing, Theory and Case Studies,* Lexington Books. 1985

[SuBK77] H. Sullivan, T. Bashkow, and D. Klappholtz, "A Large Scale Homogeneous Fully Distributed Parallel Machine", *Proc. of the Fourth Symp. on Computer Architecture,* 1977

## Figures

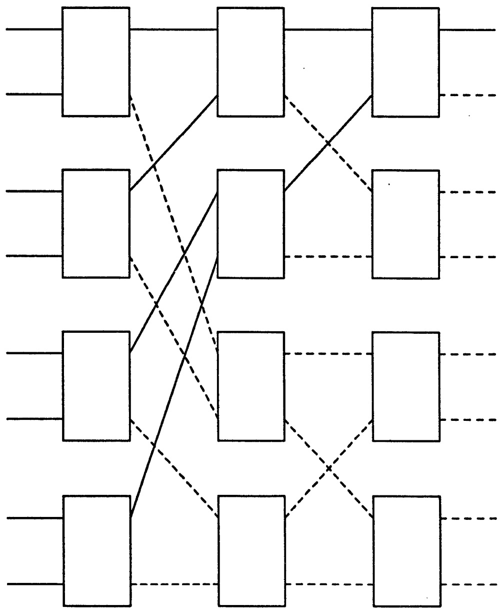


Figure 1. Fan-in Tree on a 3-stage Square Banyan Network

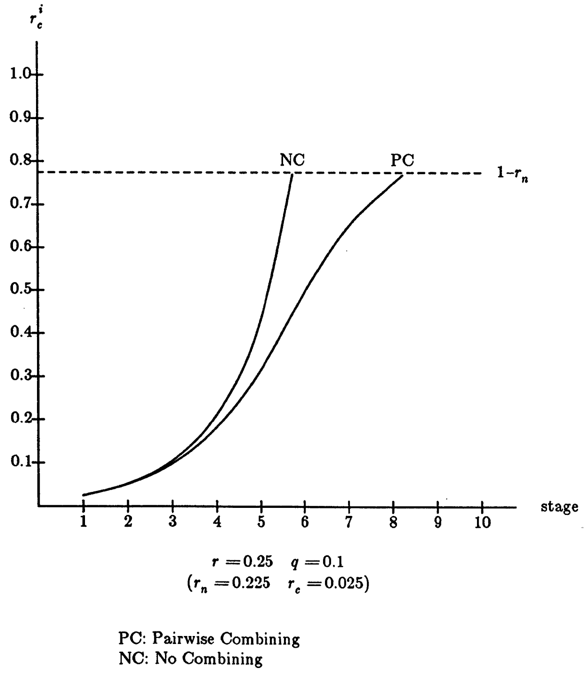


Figure 2. Traffic Load of the Combinables at Later Stages

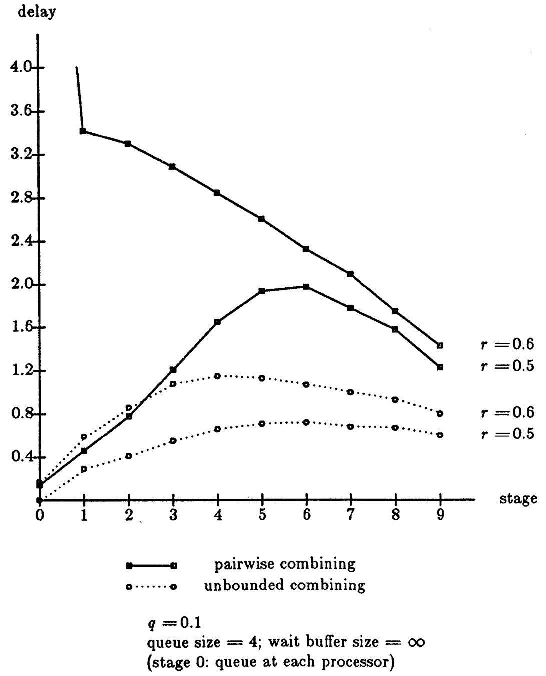
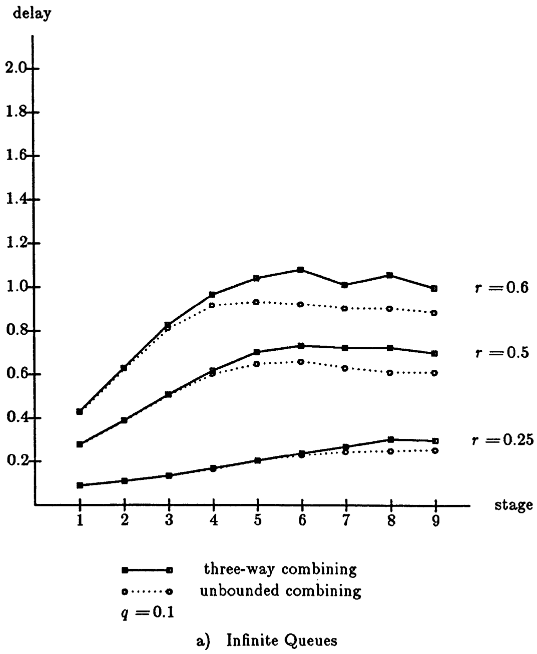
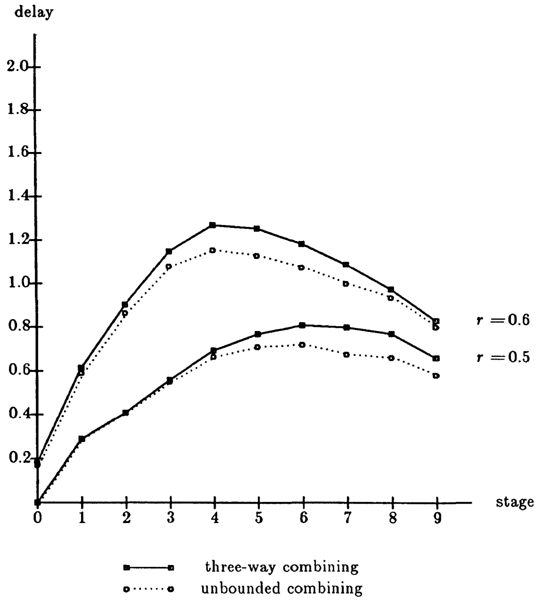


Figure 3. Tree Saturation Effect with Pairwise Combining



a) Infinite Queues

Figure 4. Delays with Bounded Combining



$q$ =0.1
queue size = 4; wait buffer size = ∞
(stage 0 : queue at each processor)

b) Finite Queues

Figure 4. (Continued)

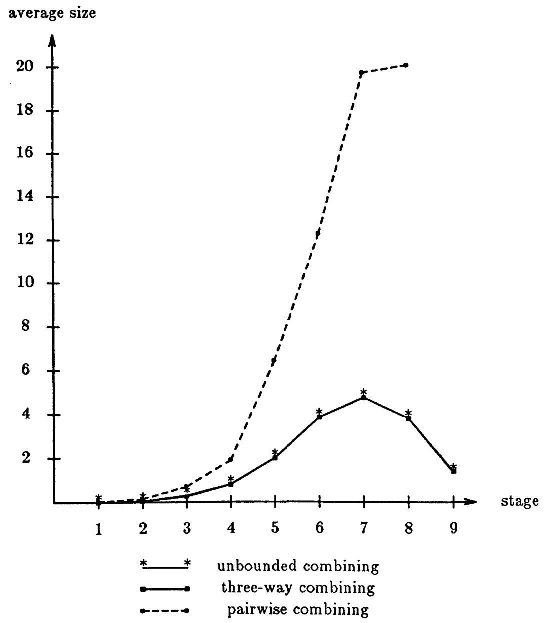
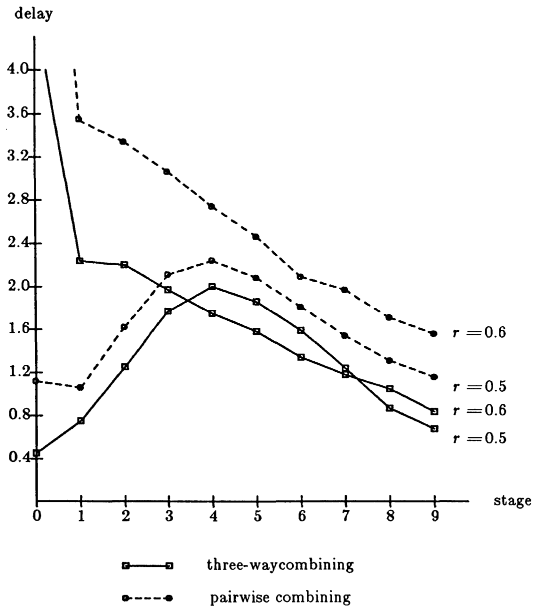
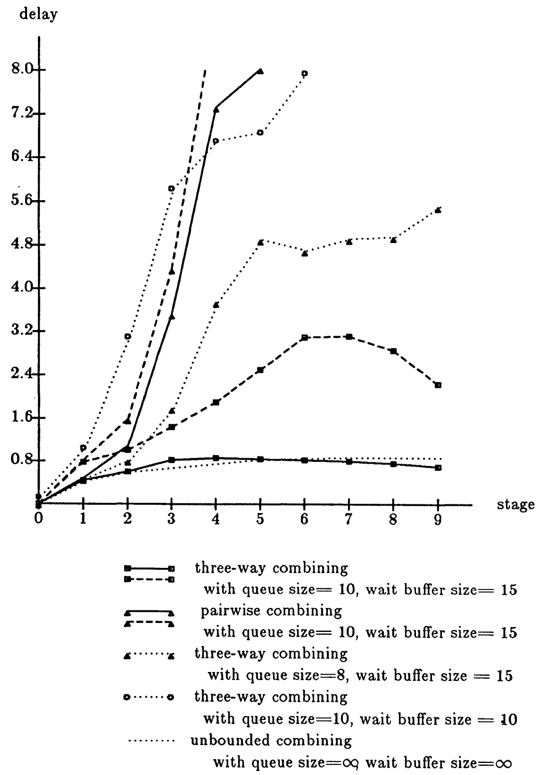


Figure 5. Average Length of Buffers ($r$ =0.25 and $q$ =0.1)



queue size =4; wait buffer size = 6; $q$ =0.1
(stage 0 : queue at each processor)

Figure 6. Effects of Small Sized Buffers



dashed lines represent the sum of delays
on the forward path and on the backward path
$r$ =0.6; $q$ =0.1

Figure 7. Performance of Bounded Combining

# SHARED MEMORY EMULATION ALGORITHM FOR MULTIPROCESSORS

Kyoji Yuyama[a]
Zary Segall
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA, 15213

Research for this paper was sponsored by DARPA (ARPA order 4976), and monitored by the Air Force Avionics Laboratory under contract NOOO39-85-C-0134. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of CMU, ARPA, or the U.S. government.

## Abstract

This paper examines the issue of shared memory emulation for performance prediction of applications running on multiprocessors. It emulates memory contentions of different target multiprocessors on a given multiprocessor. This approach provides not only the average performance measures, but also the instantaneous values of memory contentions which are essential to find bottlenecks of user's application programs.

An algorithm producing an exact emulation result is presented and its implementation trade-off is discussed. To solve these problems, a heuristic approximation approach is introduced. Experimental results on a uniprocessor system show the approach gives a reasonable accuracy. In order to alleviate the emulation overhead, parallel implementation of the algorithms is investigated.

## 1. Introduction

With the advent of commercial parallel processors we enter a new era of proliferation of parallel computation models for problem solving. One of the known characteristics of such endeavors deals with the nonlinear performance behavior of parallel programs with respect to the speed-up ratio as a function of the number of processors. As it is practically and economically not attractive to build a new parallel processor each time we may want to evaluate a parallel application on a new architecture with a different number of processors, emulation techniques are employed. Such techniques are likely to generate a ratio of 50,000 to 100,000 per one multiprocessor instruction step. Hence, the emulation process itself has to be executed in parallel on a multiprocessor.

As we look closer at the nature of this emulation, one can see that multiprocessor systems share some of their resources such as memories, buses and I/O devices to facilitate parallel access. Sometimes, an access to a shared resource has to be serialized because the shared resource may be occupied by another access. This situation is called *contention* and is related to all of the shared resources. The more processors a multiprocessor system has, the more accesses it requires in unit time, so the probability of contentions is increased accordingly. In general, due to the contention phenomenon mentioned above, the performance of an application will not be improved linearly with the number of the processors.

A practical approach to this issue is an emulation facility running on a host multiprocessor and emulating bus and memory contention of another multiprocessor. When contrasted with the performance modeling methods, the emulation method provides, aside from better accuracy, instantaneous performance profiles. The performance modeling provides, in general, statistical average performance profiles which are insufficient to accurately indicate detailed user application performance bottlenecks.

In this paper we are exploring the problem of emulating shared memory multiprocessors on shared memory multiprocessors, as well as indicating and evaluating algorithms for achieving this proposed goal.

## 2. Basic Structures

### 2.1. Memory Access Model

First, we define the memory access mechanism of the *target* multiprocessor. The multiprocessors we want to emulate are shared memory multiprocessors. In this system, global memories and processors are connected by a shared bus. These memories are divided into several banks, called *memory modules*. Access requests are transmitted from each processor to the destination memory module through the shared bus. Then, read/write operation is taking place with results returned from/to the memory module.

Because there are more than one processor and each bus/memory can accept only one request at each cycle, an access request may have to wait if bus or destination memory is servicing another request.

### 2.2. The Emulation Problem

Suppose we have a *host* multiprocessor system with *P* processors. An user of this system may want to know how much performance improvement of the user's program will be gained if there are more processors in this multiprocessor system. The purpose of the emulation facility is to provide such a *virtual* multiprocessor system within the *host* multiprocessor system. More specifically the goal and the problem can be defined as follows;

**The goal:**

> Performance prediction for applications running on a *target* multiprocessor when the number of processors are larger than that of the *host* multiprocessor. We are interested in the instantaneous

[a] Visiting Scientist from Hitachi Ltd., Tokyo, Japan.

values for the performance vector.

**The problem:**

1. Emulation of bus and memory contention for a shared-memory shared-bus multiprocessor.

2. Do step 1 using a multiprocessor with a fixed number of processors.

If we consider $N$ processes ($N > P$), each of which is to be assigned to each of $N$ processors of the *target* multiprocessor system, they can not be executed at once on the $P$ processor multiprocessor system. Therefore we adopt a *divide-composite* approach to this problem. The emulation algorithm proposed here consists of the following three steps;

1. **Divide** $N$ processes $PROC_1 \cdots PROC_N$, which are to be assigned to $N$ processors of emulated multiprocessor, into $K$ groups $Q_1 \cdots Q_K$, each group has $P$ processes. That is:

   $PROC_1 \cdots PROC_P \rightarrow Q_1$
   $PROC_{P+1} \cdots PROC_{2P} \rightarrow Q_2$

   $PROC_{(K-1)P+1} \cdots PROC_N \rightarrow Q_K$

2. **Execute** $P$ processes in each group on the $P$ processors for a certain fixed time span $T$. During execution construct the *memory access profile* consisting of time stamp, process id and memory module id. (See Fig. A) The data collection mechanism will be disussed in Section 3.

3. **Composite** memory access profile for multiprocessor of $N$ processors from $K$ individual profiles of $Q_1 \cdots Q_K$. (Also see Fig. A) This is done by eliminating bus/memory contentions among groups. Details of this part will be explained in Section 4.

Step 2 and 3 are iterated until all processes are terminated.

### 2.3. Related Works

Several performance prediction methods based on statistical models have been discussed elsewhere. [1]- [3] In these methods, distribution of memory requests is assumed to be a statistical distribution and fixed all over the period. Statistical models are quite useful for general case analysis of multiprocessor systems and when performance averages are sufficient, however application programmers may want to know the performance of their own programs rather than of the generalized program, as well as the instantaneous values of the performance measures in order to deal with performance bottlenecks.

The purpose of our research is to provide application programmers a way of performance prediction for their programs. Memory access distribution of such programs will vary time to time and process to process, and these irregularities will cause adaptability problems for statistical models. Alternately, we present an emulation mechanism for memory contention. The novelty of this work is related to the availability of instantaneous values for contentions for a specific user application, as well as the use of a multiprocessor for emulating a multiprocessor.

## 3. Data Collection Mechanism

Since the *memory access profiles* have to be constructed in parallel with the execution of a user's application program, the data collection procedure should be performed without affecting the application program. Toward this end, a hardware sensor which is transparent from the user's program is preferable to its software counterpart. If memory access data is collected by a software sensor, the overhead time for the sensor may disturb the execution of the application program and the emulation result will be different from the exact solution.

## 4. The Composition Algorithm

In this section we will present two versions of algorithms, the first is simple and produces an exact emulation, however it requires an infinite memory space. The second one is a modified version of the first algorithm which can be implemented within a bounded memory space. However, it provides an approximation of the problem.

### 4.1. The First Algorithm

Compared with the bus and the memory access, the bus cycle is the shortest of the two. Accordingly, we will consider the bus cycle as the unit of time; we call it *time slot*. Intuitively, the algorithm goes on as follows (see also Fig. B as an example):

At first, read $K$ profiles into memory. These profiles are constructed during execution phase and stored somewhere.

At time slot *1*:

- Accesses of time *1* (in Fig. B, "a","e" and "i") in those profiles are considered as candidates.
- Select one ("a") from the candidates and have unselected accesses ("e" and "i") wait for the next slot.

In general at time slot $t$:

- Accesses which are not selected in the previous stages and accesses which have time stamp $t$ are considered as candidates (for example, at time *3* "i","f","b" and "j" are candidates).

- For each candidate, check whether it is ready to be requested by the process. This check is done by seeing the time slot when the latest access of the process was accepted.

- Also check whether the destination memory module is ready to accept it by comparing memory cycle and the access interval.

- Select one from the candidates which passes the above feasibility check and have others wait for the next time slot.

- Each candidate has priority for selection. Priority is based on the group to which it belongs and the original access time stamp. Priority on groups is dynamically changed to emulate round robin strategy of multiprocessor system.

After the time slot $T$ (end of cycle):

- We have a composite profile of length $T$ and a set of accesses which are not yet selected, called *overflows*. To proceed to the next time slot ($T+1$), memory access profiles for the next cycle are needed because some accesses in the next cycle can be performed in the next time slot. So, *overflows* are combined with the memory access profiles of the next cycle and considered as the input for the next cycle. In Fig. B, access "l" and "m" are overflow accesses of the first cycle ($t = 1 \cdots T$) and should be considered as candidate accesses at the top of the next cycle ($t = T+1$).

Continue the above procedure for the next cycle ($t = T+1 \cdots 2T$) and so on.

### 4.2. The Second Algorithm

Although the algorithm described in the previous section is straightforward and produces exact emulation result, we should do some modifications when its computer implementation is considered.

At the end of every cycle($t = T, 2T \cdots$), we have a set of *overflows*. The amount of overflows can be estimated by the difference between the number of access requests (*Req*) and the capacity of the shared bus (*Cap*) when $Req > Cap$. The overflows should be added to the requests for the next cycle. When *Req* and *Cap* are constant over cycles and $Req > Cap$, we have the following relation:

$$|Overflow_j| = |Overflow_{j-1}| + Req - Cap$$

where $|Overflow_j|$ denotes the amount of overflows after $j$-th cycle. Thus,

$$|Overflow_j| = j(Req - Cap).$$

Since the above relation shows the amount of overflows grows monotonically and infinitely, the overflows will eventually become too large to keep them in memory.

To overcome this problem, we modified the algorithm so that it suspends reading the next profiles until the amount of overflows becomes smaller than the limit instead of reading them at the end of each cycle.

The modified version of the algorithm may overlook access requests because some of the access requests are not read into memory. This will cause overestimation. However, the degree of overestimation is considered to be fairly small for the following reason. At first, overflow accesses have higher priorities than accesses in the next profiles because of the *first-in first-out* nature of the selection strategy. And, the greater overflows become, the less likely it is that the next access can not be selected from overflows. This means that if we have moderate amounts of overflows, composition procedures can produce good approximation results without reading the next profiles.

## 5. Experimental Results

In order to evaluate both versions of the algorithm, we have experimented with the proposed algorithms. Since the multiprocessor system on which the emulator will run is currently under construction [4], we implemented the algorithms on a VAX 11/780 and simulated the behavior of the emulation using various simulated access profiles at different request rates generated from random distribution. The access rates of profiles are arranged to vary from time to time.

Experiment results for the case of $P = 100$ and $N = 200$ (i.e., emulating 200 processors on a multiprocessor system with 100 processors) are shown in Fig. C. In this figure, the results of the first (exact) version and modified (approximated) algorithm are indicated by circle and box, respectively. In the approximated version of the algorithm, the limit of overflows is set to $2T$ ($T$:time span).

Fig. C shows the relation between required bus access frequency, which is the total of frequencies of all groups, and resulted (composed) bus access frequency. In the ideal case, the relation between them follows the dashed line in Fig. C. However, in general, the actual relation is the dotted line below the ideal line. The maximum difference of composed access frequency between the first and modified algorithm is about 1.5%. This shows that the modified (approximated) algorithm gives a good approximation of the first (exact) algorithm.

## 6. Outline of Parallel Implementation

In this section, we discuss the implementation of the emulation algorithm in a multiprocessor. In Fig. D, the algorithm is described in a pseudo programming language. It has a triple nested loop for time slots, groups and candidate accesses in each group. Within the inner double loop each candidate access is examined as to whether processor and memory are ready for access. If more than one access are feasible at a time slot $t$, one access is selected based on the priority mentioned earlier.

Notice that feasibility check procedures for candidate accesses are mutually independent, allowing these procedures to be invoked simultaneously. The number of accesses to be checked (= the number of procedures invoked) becomes large and increases in proportion to the number of emulated processors. However, if candidates are allocated to the processors in the descending order of their priorities, unnecessary check procedures can be suppressed. Simultaneous feasibility check and priority-based processor allocation will make the check procedure work within a small number of iterations and also will make the algorithm work nearly in proportion to the length of time regardless of the number of emulated processors.

The experimental program on a uniprocessor (VAX 11/780) takes about 20*sec.* to compose two memory access profiles of 1 *ms*; that is, emulation time / execution time ratio is 20 *sec.* / 2x1 *ms* = 10,000. But, from the above discussion, we can expect high performance gain when the algorithm is finally implemented on a multiprocessor. Basically, search processes can run concurrently if they belong to the same time slot, and, even if they belong to a different time slot, they can run in parallel under the control of shared variables. If, for example, we implement the algorithm on a multiprocessor with 100 processors and gain 50 times performance gain, then the emulation overhead will be alleviated to 10,000 /50 = 200.

## 7. Conclusion

In this paper, we have presented two versions of a shared memory emulation algorithm for multiprocessor systems. The first algorithm can emulate exactly but requires an infinite amount of memory. Then, we modified this algorithm to eliminate this implementation problem. The modified version requires a finite amount of memory and experiment results show that the modified algorithm can be a good approximation for the the first algorithm. Also, the algorithm proposed here is suitable for parallel processing.

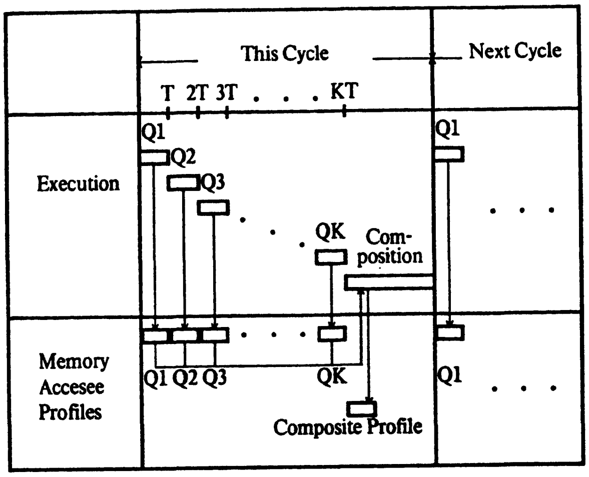


Figure A: Basic Structure of the Emulation

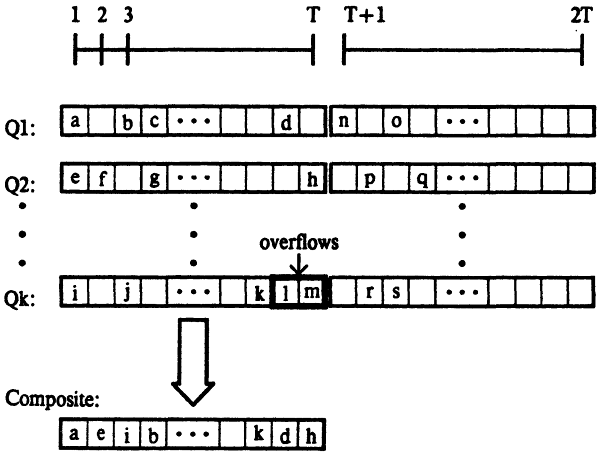


Figure B: Example of Composition

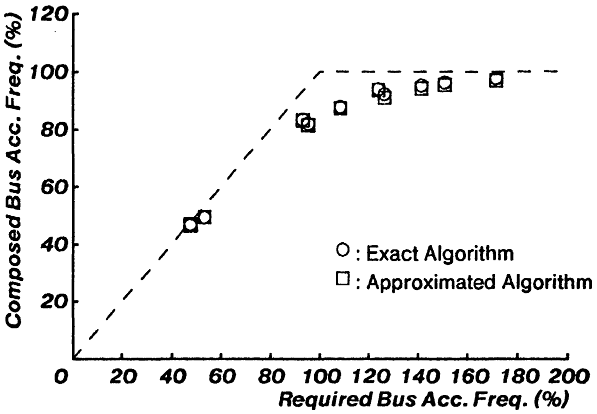


Figure C: Required and Composed Access Frequency

```
/* composition for one cycle */
procedure composition
  if bufferspace is enough then read nextprofile;
    /* loop for time slots */
  for t := 1 to T do begin
      /* loop for groups */
    for all groups i do begin
        /* loop for candidate accesses */
      for all candidate access j do begin
          /* check process and memory */
          /* access intervals       */
        feasiblecheck(access[i,j]);
      end
    end
    if more than one feasible accesses found
      then selectone based on priority;
    remove the selected access from the buffer;
    if bufferspace becomes enough
        /* perform suspended read */
      then read nextprofile;
  end
end composition;
```

Figure D: Description of the Composition Algorithm

## References

1. Baskett, F. and Smith, A.J., "Interference in Multiprocessor Computer Systems and Interleaved Memory", *Comm. ACM*, Vol. 19, No. 6, June 1976, pp. 327-334.

2. Bhandarkar, D.P., "Analysis of Memory Interference in Multiprocessors", *IEEE trans. on Computers*, Vol. C-24, No. 9, September 1975, pp. 897-908.

3. Marsan, M.A. and Gerla, M., "Markov Models for Multiple Bus Multiprocessor Systems", *IEEE Trans. on Computers*, Vol. C-31, No. 3, March 1982, pp. 239-248.

4. Rudolph, L. and Segall, Z., "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors", Tech. report CMU-CS-84-139, Carnegie-Mellon University, 1984.

# BEHAVIOR OF THE BUTTERFLY$^{TM}$ PARALLEL PROCESSOR IN THE PRESENCE OF MEMORY HOT SPOTS

Robert H. Thomas

BBN Laboratories Incorporated
Cambridge, MA 02238

**Abstract** – – This paper describes a set of experiments designed to measure the behavior of the Butterfly Parallel Processor in the presence of memory "hot spots". The experiments were motivated by a paper by Pfister and Norton [3] that reported results from simulation studies on multistage switching networks for shared memory parallel processors. Their results indicated that, for machines with a large number of processors, very slight non–uniformities in memory reference patterns can lead to severely degraded performance for the entire machine, including processors that avoid referencing the hot memories. The results were explained in terms of a phenomenon called "tree saturation" where traffic to the hot memories backs up into the switch and interferes with other switch traffic. The experiments reported here show that those results do not generalize to the Butterfly Parallel Processor. The access time for a memory that contains a hot spot is degraded, but the presence of the hot spot has little effect on the performance of programs that avoid the hot memory. Furthermore, tree saturation does not occur in the Butterfly Switch.

## INTRODUCTION

This paper describes a set of experiments that measure the behavior of the Butterfly Parallel Processor [2] in the presence of memory "hot spots". The experiments were motivated by a paper on memory hot spots by Pfister and Norton [3] that presented results of simulation studies of the switching network for RP3, a research parallel processor being developed at IBM Yorktown Heights.

The simulation results showed that non–uniformities in memory reference patterns, which make certain memories "hot", can have a devastating effect on the performance of an entire machine, including processors that avoid referencing the hot memories. Pfister and Norton explained their results in terms of a phenomenon called "tree saturation", where traffic to the hot memories backs up into the switch and interferes with other traffic, including that to non–hot memories. Their results indicated that for machines with a large number of processors (>=100) even slight non–uniformities in reference patterns can lead to tree saturation and severely degraded performance for the entire machine.

Pfister and Norton claim generality for their results, stating that they apply to all multistage blocking networks. Furthermore, their paper claims that attempts to avoid the problem, such as providing multiple paths through the network, do not really help. Finally, the results are used to motivate the use of a second, combining switch in the RP3 architecture.

The switching networks studied by Pfister and Norton were multistage shuffle exchange switches similar in topology to the switch used in the Butterfly Parallel Processor. However, there is one key difference in switch operation: the switches studied were "blocking", whereas the Butterfly Switch is not. In a blocking switch, when contention for an output port of a switching element occurs, the path from the source to that switching element is held until the desired output port can be obtained. When contention for an output port occurs in a non–blocking switch, the message encountering the contention is rejected (to be retransmitted later) and switch resources associated with it (i.e., the path to the point of contention) are released. When the message is retransmitted, it again competes with other messages for switch resources.

Thus, like the switches studied by Pfister and Norton, the Butterfly Switch is multistage. However, unlike them, it is non–blocking. Because the Butterfly Switch is non–blocking, the behavior of a program on a Butterfly system can be expected to be less severely affected by non–uniformities in memory reference patterns (caused either by the program itself or by other programs on the machine).

Nonetheless, obvious questions to ask are: how does the Butterfly Parallel Processor perform in the presence of memory hot spots? Does it exhibit tree saturation? Does the architecture break down in large configuration for programs whose memory reference patterns exhibit moderate or even very slight non–uniformities?

The experiments described below show that the results presented by Norton and Pfister do not generalize to the Butterfly Parallel Processor. The access time for a memory that contains a hot spot is degraded, but the effect of switch contention is very small, even when severe non–uniformities in memory reference patterns are present. The experiments indicate that tree saturation does not occur in the Butterfly Switch.

## THE BUTTERFLY PARALLEL PROCESSOR

This section presents enough information about the Butterfly Parallel Processor to understand the experiments described in this paper. More information about the Butterfly machine can be found in [2].

The Butterfly Parallel Processor is composed of processors with memory and a multistage switch that interconnects the processors. A Butterfly system can be configured with from 1 to 256 processors. One processor and memory are located on a single board called a Processor Node. All Butterfly Processor Nodes are identical. Collectively, the memory of the Processor Nodes forms the shared memory of the machine. All memory is local to some Processor Node; however each processor can access any of the memory in the machine, using the Butterfly Switch to make remote references. From the point of view of an application program, the only difference between references to memory on its local Processor Node and memory on other Processor Nodes is that remote

0190-3918/86/0000/0046 $01.00 © 1986 IEEE

references take a little longer to complete. (The typical memory referencing instruction takes about 6 microseconds when the data referenced is remote and about 2 microseconds when it is local.) The speeds of the processors, memories, and switch are balanced to permit the system to work efficiently in a wide range of configurations.

Each Butterfly Processor Node contains a Motorola MC68000 microprocessor (or a MC68020 with a MC68881 floating point co-processor), at least 1 MByte of main memory, a co-processor called the Processor Node Controller, memory management hardware, an I/O bus, and an interface to the Butterfly Switch. I/O connections can be made to each Processor Node, making I/O configuration very flexible.

The Butterfly machine supports a very efficient operation for transferring blocks of data from one Processor Node to another. The block transfer operation is implemented by Processor Node Controller microcode. Once initiated, a block transfer occurs at the full 32 MBit/second bandwidth of a path through the Butterfly Switch.

## THE EXPERIMENTS

Two experiments were conducted to measure the performance of the Butterfly Parallel Processor in the presence of hot spots. The objective of the first experiment was to time execution of a typical program, first in an environment without any hot spots, and then in one where N processors were used to generate a hot spot. A matrix multiplication benchmark program [1] was chosen. The objective of the second experiment was to determine the effect hot spots have on typical memory references by systematically measuring the behavior of the machine under non-uniform memory reference patterns. This was done by timing remote read, write, and block transfer operations for various memories, first in an environment without any hot spots, and then in an environment where N processors were used to generate a hot spot.

Hot spots were generated in two different ways:

1. Via read and write references. N processors were used to make a given memory hot by reading and writing the same location in that memory. This was accomplished by having each processor execute the tight loop:

   ```
   for (i = 0; i < count; i++)
       * hotmemp = * hotmemp;
   ```

   where *hotmemp* is a pointer (*short* *) to a location in the hot memory.

2. Via block transfer. N processors were used to make a given memory hot by using the block transfer operation to copy data from that memory to their local memories. This was accomplished by having each processor execute the tight loop:

   ```
   for (i = 0; i < count; i++)
       Do_bt (hotmemp, localp, numbytes);
   ```

   where *Do_bt* initiates a block transfer that moves *numbytes* bytes from the location beginning at *hotmemp* in the hot memory to the location beginning at *localp* in the processor's local memory.

The difference between these two methods is in the duration of the switch messages they generate. Simple read and write references use the switch in 2 microsecond bursts. Each iteration of the loop generates 3 messages, 2 for the read and 1 for the write. Block transfers are broken into 256 byte packets, each of which uses the switch in 64 microsecond bursts. Each iteration of the block transfer loop generates 2 messages for each packet, a short request message and a 64 microsecond response message.

Although all Processor Nodes in a Butterfly system are functionally equivalent, there is a distinguished King Node that is special in two ways: it is the node to which the console terminal is connected; and it controls the machine while the operating system is being booted. Because a terminal handler and window manager run on the King Node, it appears about 8%–10% slower than the other nodes to application programs. To ensure that the measurements were not affected by the processing requirements of the terminal handler and window manager, the King Node was avoided in both experiments.

The experiments were run on a 128 processor Butterfly system. When the experiments were run, 16 processors had been temporarily removed to configure several smaller systems, leaving 112 processors in the system. Since the King Node was not used, 111 processors were available for the experiments. The switch for this system has 4 columns (stages) of 4-input 4-output switching elements, and is configured to contain 2 paths between each pair of Processor Nodes.

## EXPERIMENT #1: MATRIX MULTIPLICATION

The matrix multiplication program was timed in a number of environments:

1. Without any hot spots.

2. With a hot spot generated by read and write references, using only cool memories for the matrices. That is, both the hot memory and the memories of processors used to generate the hot spot were avoided.

3. With a hot spot generated by read and write references, using both the hot memory and the cool memories for the matrices.

4. With a hot spot generated by block transfers, using only cool memories for the matrices. As in (2) above, both the hot memory and the memories of processors used to generate the hot spot were avoided.

5. With a hot spot generated by block transfers, using both the hot memory and the cool memories for the matrices.

### Data

For runs involving a hot spot, 100 processors were used to generate the hot spot. This left 11 processors with cool memories.

Matrix Size = 192x192

| | Time (seconds) | | | | |
|---|---|---|---|---|---|
| | Number processors. | | | | |
| | 1 | 2 | 4 | 8 | 11 |
| No hot memory | 65.73 | 32.73 | 16.37 | 8.22 | — |
| Hot memory - 100 processors doing simple read/write references | | | | | |
| Avoid hot memory (11 cool memories) | 66.02 | 32.97 | 16.67 | 8.47 | 6.27 |
| Use hot memory (11 cool memories + 1 hot memory) | 67.55 | 33.72 | 17.10 | 8.67 | 6.39 |
| Hot memory - 100 processors doing 768 byte block transfers | | | | | |
| Avoid hot memory (11 cool memories) | 66.07 | 33.13 | 16.62 | 8.50 | 6.25 |
| Use hot memory (11 cool memories + 1 hot memory) | 92.01 | 46.51 | 23.42 | 12.05 | 8.90 |

**Table 1: Data from matrix multiplication benchmark program.**

All runs used square matrices of size 192x192. This size was chosen because:

1. The run time for the matrix multiplication is long enough to give statistically interesting results, and short enough to run a series of experiments.
2. The matrix multiplication benchmark is written in a way that makes analysis of the results simpler when the matrix dimensions are multiples of 6 (see below).

The data obtained by timing the matrix multiplication benchmark on successively larger processor configurations for each set of experimental conditions is shown in Table 1.

## Discussion

When the matrix multiplication program avoids the hot memory, the presence of the hot spot has negligible impact on the program's performance: there is less than 1% increase in execution time. When the program uses the hot memory, the impact depends upon the way the hot spot is generated. There is a small increase in run time when the hot spot is generated by read and write references (2.76% in the single processor case) and a substantial increase when the hot spot is generated by block transfers (40% in the single processor case). Since block transfer operations keep the memory busy longer than single read and write references, this result is not surprising.

Switches for larger Butterfly machines are typically configured with alternate paths to make the machine resilient to failures in switching elements (which almost never occur) and to reduce contention within the switch. For example, as mentioned in the previous section, the switch for the 128 processor machine used in these experiments has one alternate path (for a total of two paths) between each pair of nodes. The data presented above was collected with the alternate switch paths enabled. Measurements were also made to determine the sensitivity of the timing data to alternate paths by repeating the experiment with the alternate paths disabled.

Use of alternate paths within the switch makes a small difference. When the hot spot is generated by read and write references and the hot memory is used, the program runs about 1% slower when the alternate paths are disabled. When the hot spot is generated by block transfers and the hot memory is used, the program runs about 2 1/2% slower when the alternate paths are disabled.

The following is an analysis of the program's behavior when running on a single processor in the presence of a hot spot generated by block transfers. It shows that the increase in execution time is due almost entirely to the increase in time required to access data in the hot memory.

The matrix multiplication program uses the block transfer operation to make local copies of matrix rows and columns before accessing the individual elements to multiply and add.

To multiply matrices of size 192x192, 36864 dot products must be computed. The program is written to compute dot products in groups of 36. This involves 12 block transfer operations to obtain 6 rows and 6 columns. Thus, 12 block transfers yield 36 results, each result requiring 1/3 block transfer. Therefore, the program performs 12288 block transfer operations.

Twelve memories were used to hold the matrices, one of which was hot. Therefore, 1/12 of the block transfers can be expected to be delayed due to the hot spot. The block transfer delay from a hot memory was measured separately by timing a 768 byte block transfer from a cool memory, and then timing it again when the memory was made hot by 100 processors doing block transfers from it:

Reference times (microseconds)

| | read | write | bt-from | | bt-to | |
|---|---|---|---|---|---|---|
| | | | 256 bytes | 768 bytes | 256 bytes | 768 bytes |
| No hot memory | | | | | | |
| remote | 15.41 | 7.87 | 111.38 | 317.17 | 112.00 | 339.26 |
| Hot memory - 100 processors doing simple read/write references | | | | | | |
| cool memory | 16.70 | 8.75 | 112.35 | 316.20 | 113.94 | 340.19 |
| hot memory | 701.93 | 306.80 | 473.99 | 1393.59 | 276.61 | 470.09 |
| Hot memory - 100 processors doing 768 byte block transfers | | | | | | |
| cool memory | 15.95 | 9.02 | 112.88 | 315.97 | 113.26 | 335.85 |
| hot memory | 17410.04 | 153.30 | 8178.84 | 25820.95 | 254.55 | 827.14 |

Table 2: Data from remote reference experiment.

| | Time to block transfer 768 bytes (microseconds) |
|---|---|
| No hot memory | 322.18 |
| Hot memory<br>100 processors doing<br>768 byte block transfers. | 25885.81 |

Therefore, the additional time for the matrix multiplication program to perform block transfers from the hot memory should be about:

```
(1/12) * 12888 * (25885.81-322.18) = 26.18 seconds
```

The measured increase in the execution time for the matrix multiplication program for a single processor was

```
92.01 - 65.73 = 26.28 seconds
```

Thus, the performance degradation resulting from the hot memory is due almost entirely to contention at that memory. The effect of switch contention on program performance is negligible, even with severely non-uniform memory reference patterns.

Note that communication (accessing remote memory) accounts for about 6%[1] of the execution time of the matrix multiplication program. Our experience with the Butterfly Parallel Processor is that communication typically accounts for 4%-10% of the execution time for an application. Because a relatively small part of total program execution time is due to communication, remote memory reference times must be severely degraded before memory hot spots can have a signficant effect on overall program performance. The purpose of the second experiment was to measure the effect memory hot spots have on remote memory references as opposed to overall program performance.

## EXPERIMENT #2: REMOTE REFERENCES

The second experiment timed references made from a given processor node to memory on every other processor node. Four types of references were timed:

1. Single word (4 byte) read references;

   ```
   t = * p;
   ```

   where *t* is a variable in local memory and *p* is a pointer (*int* *) to the word to be read.

2. Single word (4 byte) write references;

   ```
   * p = t;
   ```

   where *t* is a variable in local memory and *p* is a pointer (*int* *) to the word to be written.

3. Block transfer of data from the remote memory;

   ```
   Do_bt (remotep, localp, numbytes)
   ```

   where *remotep* is a pointer to a block of data on a remote node to be copied, *localp* is a pointer to an area in local memory, and *numbytes* is the number of bytes to be copied to local memory.

4. Block transfer of data to the remote memory;

   ```
   Do_bt (localp, remotep, numbytes)
   ```

   where *localp* is a pointer to a block of data in local memory to be copied, *remotep* is a pointer to an area on a remote node, and *numbytes* is the number of bytes to be copied from local to remote memory.

The measurements for a given reference type were made by timing a tight loop that included the memory reference:

```
Start_timer;
for (i = 0; i < loopcount: i++)
    Make_reference;
Stop_timer;
```

In addition, the empty loop was timed to measure loop overhead:

[1] = 100% * (12288 blk xfers * 322.18 microsec/blk xfer) / (66.02 sec).

```
Start_timer;
for (i = 0; i < loopcount: i++) ;
Stop_timer;
```

### Data

Runs that involved hot spots used 100 processors to generate the hot spot. Therefore, in those runs there was 1 (remote) hot memory, 10 (remote) cool memories, 1 (local) cool memory, and 99 (remote) memories for processors generating the hot spot.

The timing data in Table 2 shows average times for one iteration of the memory referencing loop for the various memory reference types under the conditions indicated. For the first set of data, which was collected without any hot spots, the "remote" reference times were computed by averaging the loop times measured for each of the 110 remote memories and dividing by *loopcount*. Data from the hot memory measurements was treated similarly. For example, the "hot memory" reference times were computed by dividing the measured times through the reference loop by *loopcount*; and the "cool memory" reference times were computed by averaging the loop times for the 10 cool memories and dividing by *loopcount*. *Loopcount* for this data was 10000. The loop overheads for each of the conditions were measured as described above, and factored out of the data. That is, the times presented exclude the measured loop overheads.

### Discussion

When there is a hot memory, references to cool memory are slowed down slightly. This is probably due to contention within the switch; switch messages used to reference cool memory collide with the switch messages used to make the memory hot.

When there is a hot memory, simple references to cool memory are slowed down about the same amount as block transfer references to cool memory. For example, remote reads from a cool memory when the hot spot is generated by read and write references are slowed by 1.29 microseconds (16.70 versus 15.41), and 256 byte block transfers from a cool remote memory are slowed by .97 microseconds (112.35 versus 111.38)[2]. This is not surprising since the slow down is due to the increased time for initiating successful message transmission through the switch, and the increase is independent of message size.

References to the hot memory are substantially slower. For most types of references a memory made hot by block transfers is slower than one made hot by read and write references. The major exception is that simple writes are slower when the memory is made hot by read and write references than when it is made hot by block transfers (306.80 versus 153.30). This is due to the buffering strategy in the Processor Node switch interface which, in effect, gives preference to simple writes: when the memory is hot due to read and write references, the write being timed must compete with the writes making the memory hot; whereas when the memory is hot due to block transfers, there are no other writes to compete with.

## CONCLUSIONS

The principal conclusion to be drawn from these experiments is that the results reported by Pfister and Norton do not generalize to the Butterfly Parallel Processor. While memory contention has an important effect on program performance in a Butterfly system, switch contention does not.

The matrix multiplication experiment showed that non-uniformities in memory reference patterns have very little effect on the behavior of a program that avoids the hot memory. When the hot memory is avoided, its presence has virtually no effect on a program's performance, even if the non-uniformities are large.

If a program uses a hot memory, the performance degradation due to the hot memory depends on the extent to which the hot memory is used by the program. That is, the program is appreciably slowed only when it references the hot memory. Although the memory reference experiment showed slight slow down in references to the cool memories, the matrix multiplication experiment showed that the slight slow down has negligible impact on overall program performance.

There is no evidence that the tree saturation phenomenon described by Pfister and Norton occurs in the Butterfly Switch. Severe non-uniformities can lead to a small increase in contention within the switch, but the saturation effect simply does not occur.

### Acknowledgements

I would like to thank Will Crowther, John Goodhue, and Walter Milliken, who read several drafts of this paper and provided helpful suggestions for presenting the results of the experiments. The Defense Advanced Research Projects Agency provided support for the development of the Butterfly Parallel Processor.

### References

[1] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, T. Blackadar.
Performance Measurements on a 128-Node Butterfly Parallel Processor.
In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531-540. IEEE Computer Society Press, August, 1985.

[2] *Butterfly Parallel Processor Overview*
BBN Laboratories Incorporated, 1985.

[3] G.F. Pfister and A. Norton.
"Hot Spot" Contention and Combining in Multistate Interconnection Networks.
In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 790-797. IEEE Computer Society Press, August, 1985.

[2] 768 byte block transfers were actually measured to be slightly faster (316.20 versus 317.17).

# DISTRIBUTING HOT-SPOT ADDRESSING IN LARGE-SCALE MULTIPROCESSORS

Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

## ABSTRACT

When a large number of processors try to access a common variable, referred to as *hot-spot* accesses in [6], not only can the resulting memory contention seriously degrade performance, but it may also cause *tree saturation* in the interconnection network which blocks both hot-spot and regular requests alike. It is shown in [6] that even if only a small percentage of all requests are to a hot-spot location, these requests can cause very serious performance problems, and networks that do the necessary combining of requests are suggested to keep the interconnection network and memory contention from becoming a bottleneck.

Instead we propose a *software combining tree* concept and show that it is effective in decreasing memory contention and preventing tree saturation because it distributes hot-spot accesses over a software tree whose nodes can be dispersed among many memory modules. Thus it is an inexpensive alternative to expensive combining networks.

## 1. INTRODUCTION

A large, shared-memory multiprocessor system like Cedar [1], the Ultracomputer of NYU [2], or the RP3 of IBM [3], may contain hundreds or even thousands of processors and memory modules. Multistage interconnection networks such as the Omega network [4] or its variations [5] are usually employed to provide communication between these processors and memory modules.

In these systems, any variable shared by these processors will create memory contention at some memory modules. Those shared variables could be locks for process synchronization [15], loop index variables for parallel loops [12], etc. Even though accesses to these shared variables (called *hot-spot* accesses in [3,6]) may account for a very small percentage of the total data accesses to the shared memory (typically less than 10% are observed in most applications), this memory contention can create a phenomenon called *tree saturation* [6], and can cause severe congestion in the interconnection network. It is shown [6,14] that tree saturation due to hot-spot contention can seriously degrade the effective bandwidth of the shared memory system.

Various schemes like combining networks used in the IBM RP3 [3] and NYU Ultracomputer [2], or the repetition filter memory in the Columbia CHoPP [7] has been proposed to eliminate such memory contention. The basic idea of these schemes is to incorporate some hardware in the interconnection networks to trap and combine data accesses when they are fanning in to the particular memory module that contains the shared variable. Because data accesses can be combined in the interconnection network, it is hoped that memory contention at that memory module can be eliminated.

However, the hardware required for such schemes is extremely expensive. It is estimated [6] that the extra hardware increases the switch size and/or cost by a factor between 6 and 32, and this is only for combining networks consisting of 2×2 switches. With $k \times k$ switches ($k > 2$), the hardware cost will be even greater. The extra hardware also tends to add extra network delay which will penalize most of the regular data accesses that do not need these facilities, unless the combining network is built separately as in RP3 [6].

Furthermore, the effectiveness of the combining network depends very much on the extent to which such combining can be done. If such combining is restricted as described in [8], i.e., if the number of requests that can be combined is restricted to $k$ in a $k \times k$ switch, then the effectiveness of the combining network can be limited. Unless this combining is unrestricted, tree saturation can still occur even in a combining network [8].

In this paper, we are studying this problem from a different perspective. We assume a shared memory multiprocessor system like Cedar [1] with a standard, buffered Omega network providing interconnection [9], and without expensive combining hardware. In addition we use a hardware facility in the shared memory modules to handle necessary indivisible synchronization operations for the shared variables [10]. Regular memory

This work was supported in part by the **National Science Foundation** under Grants No. US NSF DCR84-06916 and US NSF DCR84-10110 and by the **US Department of Energy** under Grant No. US DOE DE FG02-85ER25001.

accesses bypass this hardware without delay and, hence, will not be penalized. Each memory module will handle memory accesses, including those memory accesses to shared variables, one at a time.

To eliminate memory contention due to the hot-spot variable, a software tree is used to do the combining. This idea is similar to the concept of a combining network, but it is implemented in software instead of hardware. We will show that this scheme can achieve quite satisfactory results as compared to more expensive hardware combining.

## 2. HOT-SPOTS AND TREE SATURATION

The phenomenon of how hot-spot accesses can cause tree saturation is briefly described here. For a more detailed analysis and discussion, please refer to [6].

Assume $N$ is the number of processors in the system, and there are also $N$ memory modules in the shared memory system. Each processor issues $r$ requests to the shared memory per network cycle ($0 \leq r \leq 1$). Among those requests, $h$ percent of the requests are hot-spot requests. Thus, in each network cycle, there are $Nrh$ hot-spot requests and $r(1\text{-}h)$ normal requests directed to the "hot" memory module for a total of $Nrh + r(1\text{-}h)$. If each memory module can accept 1 request per network cycle (i.e., the maximum rate), the maximum network throughput per processor is

$$H=1/(1+h(N\text{-}1)) \qquad (1)$$

and the total effective memory bandwidth for the shared memory system is

$$B=N/(1+h(N\text{-}1)). \qquad (2)$$

Fig. 1 shows B as a function of $N$ for various $h$. This clearly shows that in a system with 1000 processors, hot-spot traffic of only 1% can limit the total memory bandwidth B to less than 10%.

Notice that this discussion assumes that hot-spot requests can continue to be issued from a processor even if that processor still has an unsatisfied hot-spot request pending in the network. In many applications this is not true, because hot-spot requests are usually related to some kind of synchronization operation: A processor usually has to wait for the outcome of the synchronization operation before it can issue another request to the synchronization variable. So, the issuing rate from a processor is inherently limited. We will address these issues in more detail in later sections.

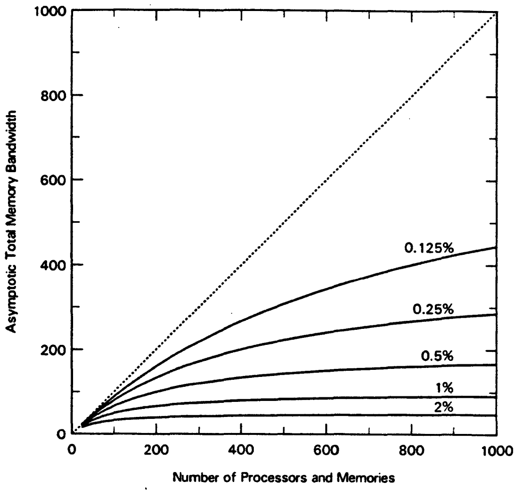


Fig. 1. Asymptotically maximum total network bandwidth as a function of the number of processors for various fractions of the network traffic aimed at a single hot spot.

(results from [6])

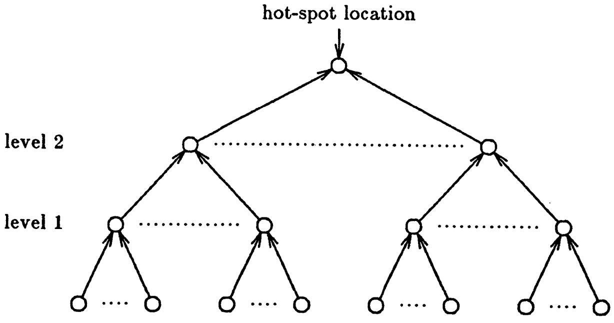


Fig. 2. A tree with fan-in = 10 to the hot-spot location.

## 3. SOFTWARE COMBINING TREES

To illustrate the principle of a software combining tree, let us assume that we have a variable whose value is N and that we want each processor to decrement this variable so that when all processors are finished, the value will be zero. This is a common way of making sure all processors are finished with a given task before proceeding with a new task, for example, and is one cause of hot-spot accesses. Now suppose that instead of one single variable, we build a tree of variables, assigning each to a different memory module, as shown in Fig. 2. If N = 1000 and assuming a fan-in of 10, we have 111 variables each with value 10. We partition the processors into 100 groups of ten, with each group sharing one of the variables at the bottom of the tree. When the last processor in each group decrements its variable to zero, it then decrements the value in the parent node. Thus, we have increased the total number of accesses from 1000 to 1110, but instead of having one hot spot with 1000 accesses, we have 111 hot spots with only 10 accesses each. It should be clear that this will result in a significant improvement in throughput rate and bandwidth, and the simulations we describe later verify that even if we account for the increase in total accesses, the improvement is still quite significant. It should also be clear that a three-level tree with fan-in equal to 10 is not necessarily optimal, but that the optimal point depends on access times and on other factors.

Another basic operation that can be implemented with a software combining tree is *busy-wait.* Here it is assumed that processors are waiting for a shared variable to change in some way. Presumably some other processor will cause this change. We build a combining tree as before, this time assigning one processor to each node in the tree. Each processor monitors the state of its node by continually reading the node. When the processor monitoring the root node detects the change in its node, it in turn changes the state of its children's nodes, and so on until all processors have detected the change and are able to proceed with the next task.

This idea, in a sense, is not very different from a hardware combining tree built from 10X10 switches, except that the combining buffer that would be inside each switch now resides in a shared memory module in a software combining tree. One distinct advantage for a software combining tree is that we can tune our performance by changing the fan-in of each node without incurring any hardware cost.

### 3.1. Modeling of Software Combining Tree

We will classify hot-spot accessing in two ways. First, accesses will be *limited* or *unlimited* depending on whether a given processor can have only one or more than one hot spot request outstanding. We let $\eta$ denote the number of outstanding hot-spot requests. Second, the number of accesses will be *fixed* or *variable* depending on whether the total number of accesses is fixed, or whether the total number varies depending on the number of conflicts or some other factor. For example, assume we are adding a vector of numbers to form a sum. Then each processor can have more than one outstanding request to add an element to the shared sum, but since we assume the addition is done indivisibly by logic in the memory, the total number of requests generated by all the processors is fixed. This case is *unlimited-fixed.* A case like that described earlier where processors are decrementing a counter to see who is the last processor is *limited-fixed.* A third example is illustrated by *busy waiting* where the processors may all be waiting for one processor to complete some task. Each processor continually reads the value of a shared variable until the value changes, for example from zero to one. Thus the number of requests to the hot spot depends on how soon the variable gets reset, and this case is *limited-variable.* Notice that a barrier synchronization [11] can be implemented by a counter decrement (*limited-fixed*), followed by a busy wait (*limited-variable*) triggered by the final processor which decrements the counter.

When we implement combining trees for hot-spot accesses, it is important to minimize the possible memory contention, and so it is preferable that all shared variables in a software combining tree (i.e., the nodes of the tree) reside in separate memory modules. The largest combining tree we can construct for a hot spot is a tree with minimum fan-in, i.e., a fan-in of 2. The total number of nodes in a combining tree with N leaves is N/2 + N/4 + $\cdots$ + 2 + 1 = N-1. Hence, it is always possible to spread those nodes across N separate memory modules. Our simulations in this study assume all of the nodes in a software tree to be in separate memory modules.

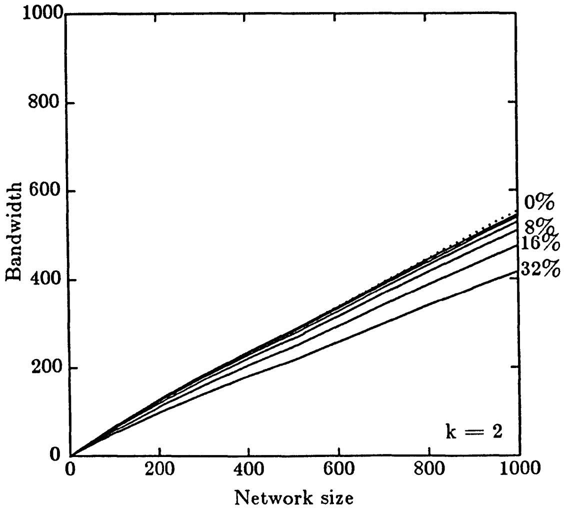


Fig. 3. Lower bound of network bandwidth under various hot-spot rates ($\eta$ = unlimit).

We also assume the following system configuration:

(1) There are two identical, back-to-back, uni-directional Omega networks: one is for traffic from processors to the shared memory; the other is for traffic from memory returning to the processors. Both networks are packet-switching, pipelined networks.

(2) Each network consists of 2X2 switching elements with an output buffer of finite size at each output port of a switching element. The fan-in capability of each output port is 2, i.e., it can accept two simultaneous requests from its two input ports. One request is forwarded to the next stage and the other is stored in the output buffer. If the output buffer is full, no more requests are accepted by the output port. In our simulations, we assume the size of the output buffer to be 4.

(3) When a software combining tree is used, accesses to a shared variable are distributed over the nodes of a tree instead of a single shared variable, and additional memory accesses are needed to access these nodes in the tree. This occurs both in a counting operation where the last processor to decrement a node must also decrement the parent node, and in the busy-waiting case where each processor except at the leaves must propagate the state change to the node's children. These extra accesses are accounted for in our simulations.

(4) All requests are of the same length. In our simulations, we assume each request consists of only one packet.

(5) The access time of a memory module is 1 network cycle, i.e., the time for a request to go through a switching element when no conflict exists.

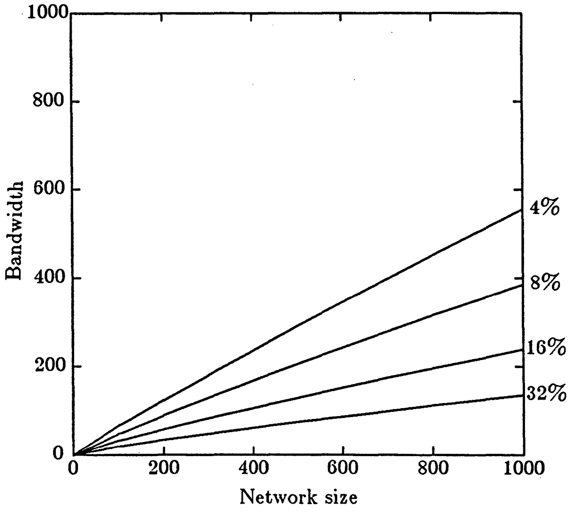


Fig. 4. Upper bound of network bandwidth under various hot-spot rates ($\eta = 1$).

### 3.2. Possible Overhead in a Software Combining Tree

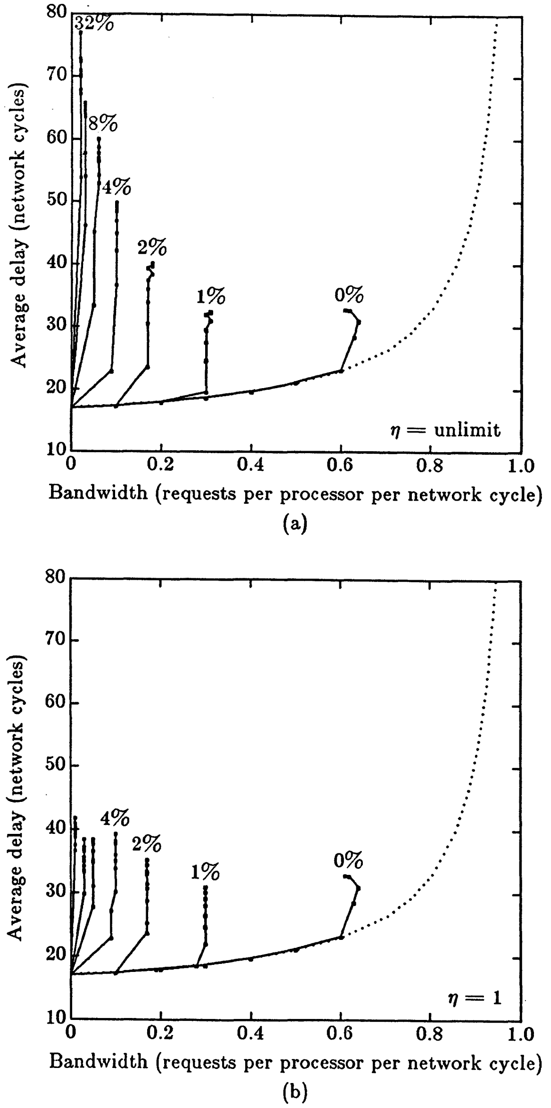
As mentioned earlier, constructing a software combining tree creates many shared variables. Therefore, more hot-spot traffic is created even though that traffic generates less memory contention.

As before, let us assume that the hot-spot rate from a processor is $r \times h$, and the software combining tree has a fan-in of $k$ for each node. For *fixed*-type access patterns, the fractional increase in hot-spot traffic will be

$$\sum_{l=1}^{\log_k N-1} rh/k^l = rh(\frac{1-(k/N)}{k-1}).$$

When $k = 2$, the increased hot-spot traffic is $rh(1-2/N)$, which approximates the original hot-spot traffic for large $N$. This means that the hot-spot traffic can not be more than doubled after all of the extra hot-spot traffic is included. As we will see later in our simulations, the decreased memory contention will more than offset the increased hot-spot traffic if $h$ is less than 30%.

For *variable* access patterns, the additional accesses caused by the combining tree are difficult to quantify because the number of accesses is not fixed to begin with. In practice, since busy-waiting is often the cause of *variable* access patterns (with $\eta = 1$), and the number of accesses for a busy-wait operation depends on how quickly the state change is propagated to the children in the tree, the total number of accesses could even be less than that required by a single shared variable because the state change can be propagated more quickly by the combining tree than by N accesses to a single shared variable.



(a)

(b)

Fig. 5. Average delay versus bandwidth for a size 256 network. ($h$ varies from 0 to 32%)

## 4. BOUNDS ON BANDWIDTH

### 4.1. Unlimited Hot-Spot Requests per Processor

In a packet-switching Omega network, with finite buffers in each switching element and with hot-spot rate $h = 0$, we still cannot achieve 100% memory bandwidth because of conflicts in the network [9]. These conflicts are also possible if a crossbar switch is used. If we assume R to be the maximum request rate reaching a memory module when no hot-spot exists, then in a steady state, R is also the maximum request rate allowed for a processor. Therefore, we can consider R to be an absolute upper bound on the bandwidth per processor.

The value of R depends on the network buffer size, the length of a request, and the network switch size, etc.

[13]. However, as $h$ increases, the request rate to the hot memory module, i.e., $r(1\text{-}h)+rhk$, will increase from R to 1. Tree saturation will occur when the request rate to the hot memory module approaches 1, and the maximum processor request rate $r$ will decrease. Hence, we have

$$R \leq r(1\text{-}h)+rhk \leq 1.$$

By rearranging the above equation, we have the following:

$$R/(1+h(k\text{-}1)) \leq r \leq 1/(1+h(k\text{-}1))$$

$1/1+h(k\text{-}1))$ is equal to 1 when $h$ is 0. Since the absolute upper bound is $R$ ($R \leq 1$) as discussed before, we can have a tighter upper bound by using $R$, i.e.,

$$R/(1+h(k\text{-}1)) \leq r \leq R. \qquad (3)$$

Notice that Eq. (3) also shows a lower bound for the maximum processor request rate $r$ when a software combining tree is used with a fan-in of $k$, and $\eta$ is unlimited, i.e., even when $\eta$ is unlimited, the maximum bandwidth cannot be worse than $NR/(1+h(k\text{-}1))$.

We obtained $R$ from simulations, and in Fig. 3 we plot lower bounds for various system sizes with $h$ varying from 0% to 32%. Notice that those curves are in a very narrow range, i.e., the lower bound in Eq. (3) seems to be tight at least for systems up to size 1024. The top dotted line in Fig. 3 shows R, the maximum bandwidth we can get when there are no hot-spot requests.

The degradation factor in Eq. (3) is $1+h(k\text{-}1)$. This degradation factor is independent of the system size and reaches a minimum when $k = 2$. Given unlimited hot-spot requests, i.e., $\eta \geq 1$, the optimal software combining tree for maximum memory bandwidth has a minimum fan-in of 2.

### 4.2. Single Hot-Spot Request per Processor

If the hot-spot request rate is limited ($\eta = 1$), then there cannot be more than N hot-spot requests in the system at any time. For systems with instruction look-ahead or with data prefetching capability, regular requests still may be issued while a hot-spot request is pending. However, this case is not different from that of unlimited hot spot requests with a very small $h$: when $h$ is very small, it is unlikely that there will be more than one hot-spot request pending at any time.

Hence, when $\eta = 1$, we will only consider the case where no additional requests, hot or regular, are issued by the processor when there is a pending hot-spot request. Thus, the bandwidth depends on the delay of the hot-spot requests. The request rate from the processor is further restricted by any increased delay. If a software combining tree is used to eliminate the memory contention caused by the hot-spot requests, the limiting factor for the memory bandwidth will only be $\eta$; the inherent nature of the hot spot that prohibits further processor requests.

During a long period of time $T$, there will be $rT$ requests generated from a processor, among which $rhT$ requests are hot-spot requests. The processor will be barred from issuing any request for a total period of $rhTC$, where $C$ is the average round-trip delay for a hot-spot request. The processor can issue a request only for a total period of $T-rhTC$. Within that period, $r(1\text{-}h)T$ regular requests are issued. Hence, the real issuing rate for regular requests is $r(1\text{-}h)T/(T-rhTC)$. This rate can not be greater than 1, i.e.,

$$r(1\text{-}h)T/(T-rhTC) \leq 1.$$

This equation can be rearranged to obtain an upper bound for $r$:

$$r \leq 1/(1\text{-}h+hC) \qquad (4)$$

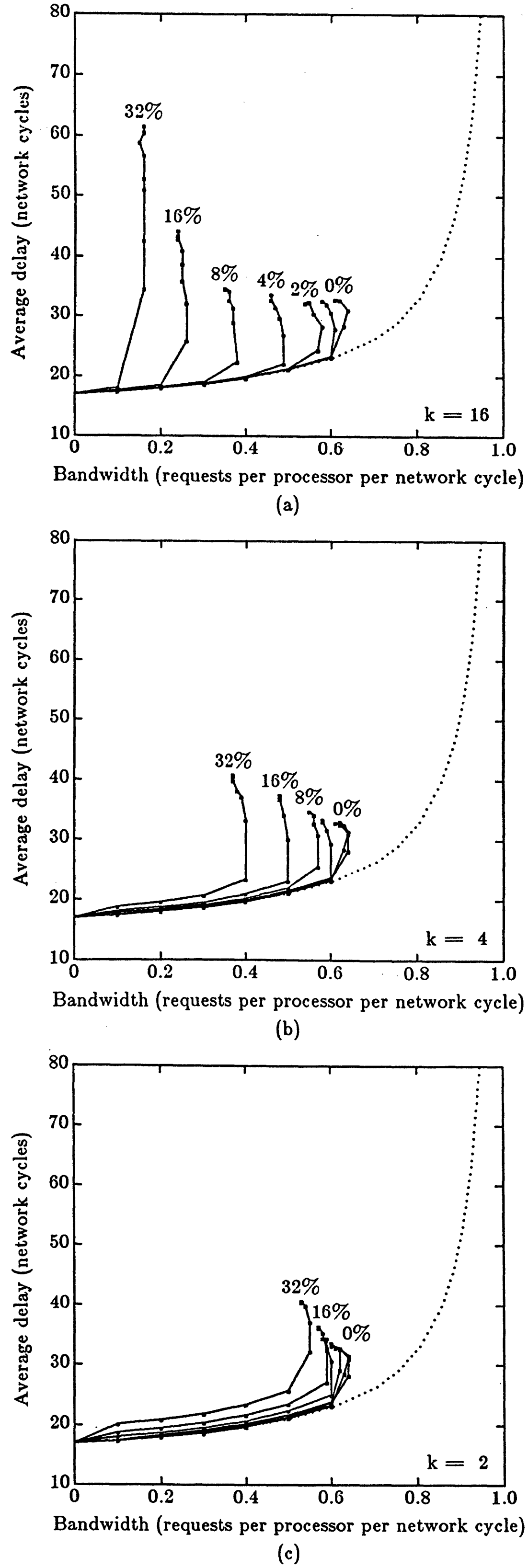
As expected, the maximum rate of $r$ is greatly dependent on the hot-spot delay $C$. This bound gets tighter as the hot-spot rate $h$ gets larger. When $h = 1$, the equality in Eq. (4) will hold. Fig. 4 shows this upper bound for various hot-spot rates $h$ with minimum hot-spot delay of $C = 2\log_2 N$. For N = 1000 and $h = 8\%$, the upper bound will be around 40% of the total bandwidth. Notice that *the upper bound in Eq. (4) is valid even for a hardware combining network* because it is a bound imposed by the inherent nature of the hot-spot request (i.e., $\eta = 1$).

## 5. SIMULATION RESULTS

To study the effectiveness of a software combining tree, we performed several simulations for N = 256, with $h$ varying from 0% to 32%. Fig. 5 shows the delay and maximum bandwidth when neither a software combining tree nor a hardware combining network is used. Following each curve from left to right, each point represents a larger value of r. As shown in [6], while r increases, bandwidth increases while delay stays relatively constant up to a point of saturation. After the saturation point, bandwidth ceases to increase while delay gets worse. This clearly shows low bandwidth and increased average network delay results. The maximum bandwidth of 0.63N is achieved when $h = 0$.

Fig. 6 represents *fixed*-type access patterns with unlimited $\eta$, and shows the use of a software combining tree to reduce hot-spot contention. The fan-in's for the software combining trees are varied from $k = 16$ to 2. The improvement is quite significant compared to the result in Fig. 5(a). According to Eq. (3), the minimum degradation factor for the bandwidth can be obtained when the software combining tree has the minimum fan-in. In Fig. 6 we can see that when $k = 2$, the degradation is indeed the smallest.

As presented in section 3.2, the hot-spot traffic can be nearly doubled by the extra hot-spot traffic created by the software combining tree with the minimum fan-in $k = 2$. In Fig. 6 $h$ is indicated as the original hot-spot request rate; the results shown there already include all

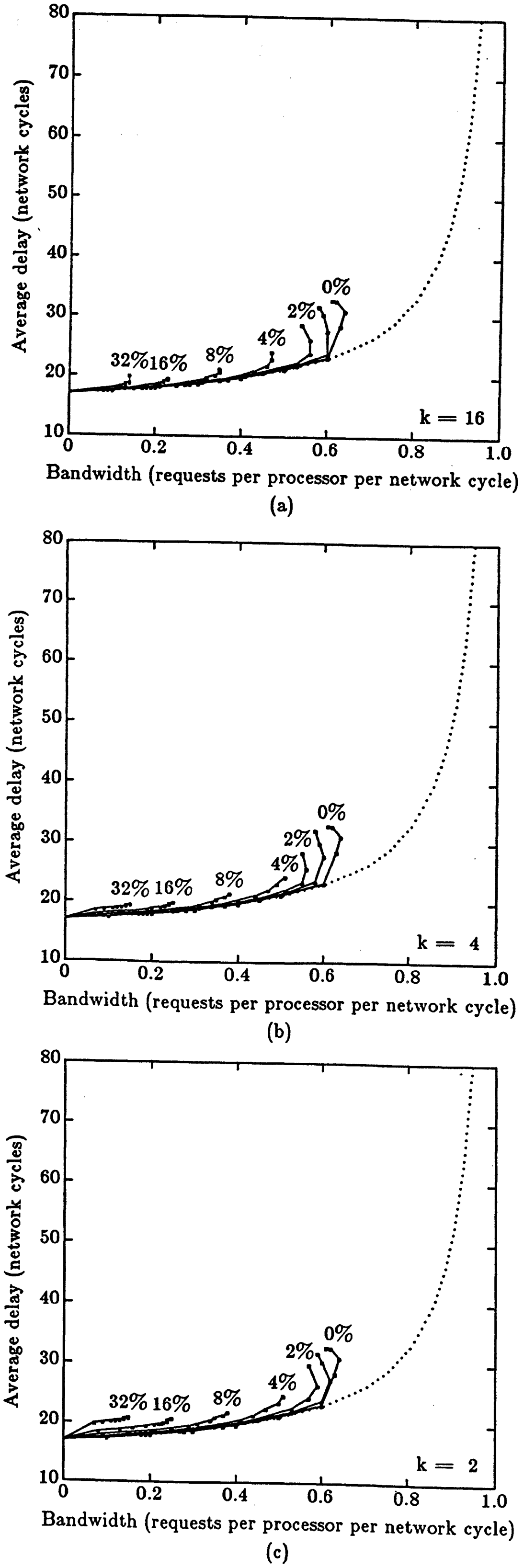


(a)

(b)

(c)

Fig. 6. Average delay versus bandwidth for unlimited-fixed access patterns. (N = 256, $h$ varies from 0 to 32%)



(a)

(b)

(c)

Fig. 7. Average delay versus bandwidth for limited-fixed access patterns. (N = 256, $h$ varies from 0 to 32%)

extra hot-spot traffic. This shows that with an original hot-spot request rate of 16%, the degradation remains small. The elimination of the hot-spot contention, indeed, more than offsets the results of increased traffic.

Fig. 8 represents *limited-variable* access patterns, wherein no additional requests are issued by a processor while it has a hot-spot request pending, but the total number of requests allowed over time is not fixed. The upper bound on the bandwidth given in Eq. (4) will depend on $C$, the average delay of the hot-spot requests. The value of delay $C$ includes the overhead from traversing the software tree, busy waiting in the intermediate nodes, and the possible memory contention. From these figures, we can see that the optimal fan-in $k$ for the software tree is no longer $k = 2$, but rather at around $k = 4$. The increased fan-in $k$ allows for a lesser number of levels of nodes in the tree, thus reducing the time required for requests to traverse the tree.

Furthermore, when $h$ increases, the upper bound in Eq. (4) becomes tighter. There is less traffic in the network due to the restriction that no more requests will be issued when a hot-spot request is pending. In this case, the turnaround time for a request can actually be improved as Fig. 9 shows.

We also simulate some cases for fixed-type access patterns with $\eta = 1$ (Fig. 7). If we take into account the fact that busy waiting is not required in this kind of access pattern, we can see that the results are quite similar to those from our simulations of variable-type access patterns discussed above. In fact, the average hot-spot request delay, i.e., $C$ in Eq. (4), is smaller in this case. Also, as shown in Eq. (4), we can expect an improved maximum rate $r$.

The lower dotted lines in Fig. 5 through Fig. 9 are the average delay of a request through the network assuming the buffer size in each switching element is infinite. These values are calculated based on the analytical model in [16].

## 6. DISCUSSION

Our simulations show that the software combining tree effectively eliminates tree saturation caused by hot-spot contention. However, the main purpose of the software combining tree differs slightly from the original purpose of the hardware combining networks [2,6].

Hardware combining networks were originally proposed to speed up hot-spot requests by combining those requests in the interconnection network and in this way eliminate memory contention at the hot memory module. Because such memory contention creates the serious side effect of tree saturation that can adversely affect even regular requests [6], such requests must also be processed through the hardware combining network. Although it alleviates the problem of tree saturation, hardware combining can cause extra delay in processing regular requests.

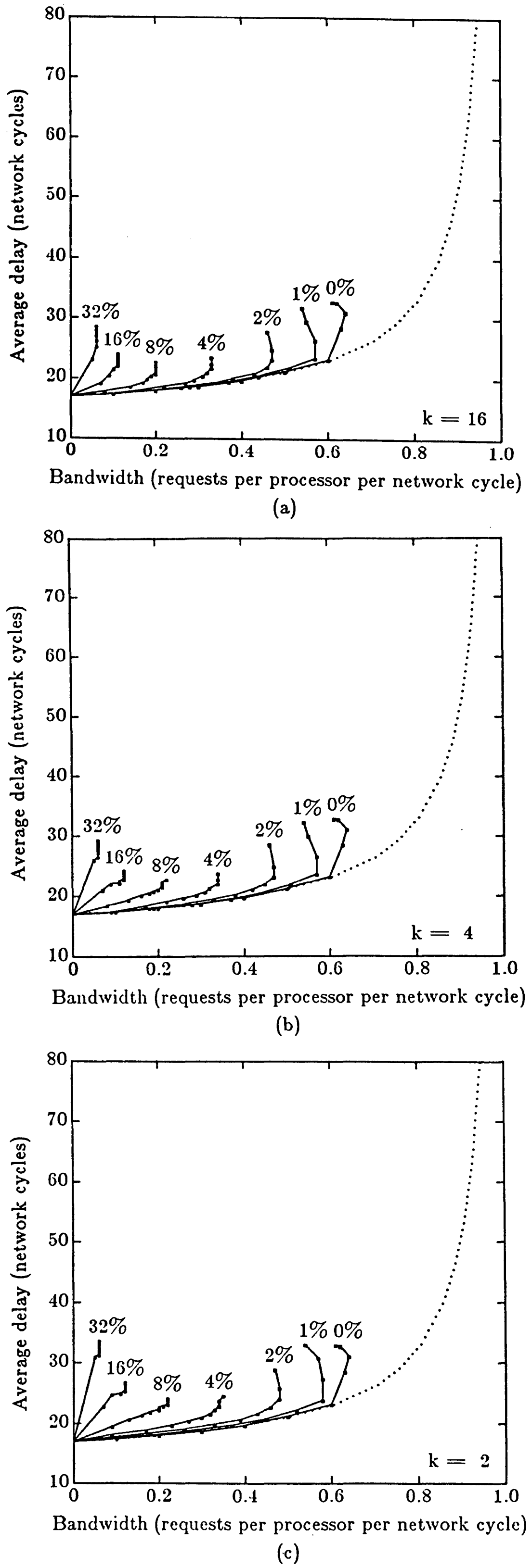


Fig. 8. Average delay versus bandwidth for limited-variable access patterns. (N = 256, $h$ varies from 0 to 32%)

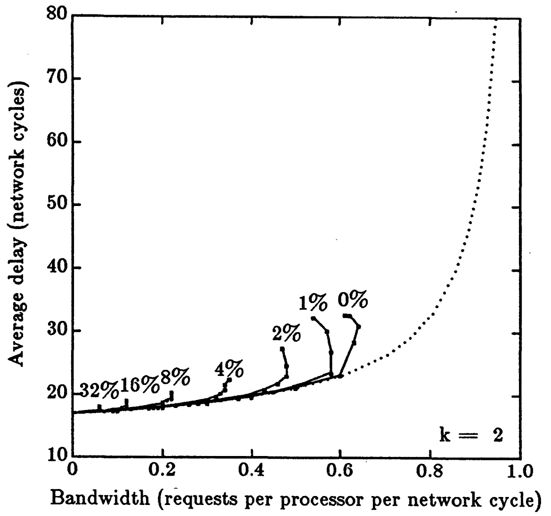


Fig. 9(a). Average delay of regular requests versus bandwidth for limited-variable access patterns. (N = 256, *h* varies from 0 to 32%)

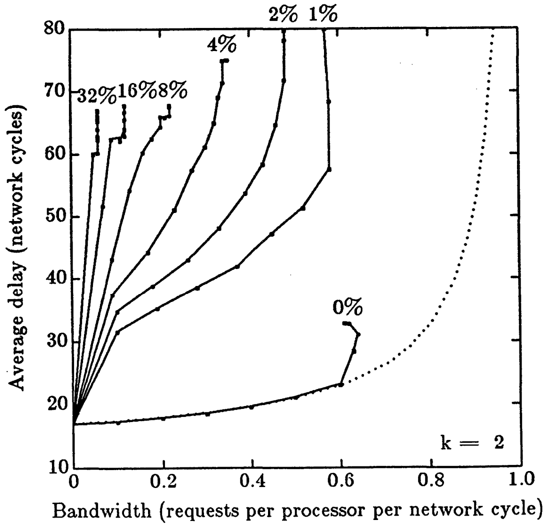


Fig. 9(b). Average delay of hot-spot requests versus bandwidth for limited-variable access patterns. (N = 256, *h* varies from 0 to 32%)

Software combining trees seem to effectively relieve regular requests from the side effect of tree saturation, without the expense of hardware combining networks. The beneficial side effect from this scheme is that the service time of hot-spot requests decreases. Theoretically, this improvement cannot be as good as a hardware combining network with unrestricted combining capability: in a software combining tree, a hot-spot request must traverse the interconnection network $\log_k N$ times, whereas in a hardware combining network the request must traverse the network only once. However, in a real implementation, unrestricted combining in a switch is impossible due to the complexity of the switches in a hardware combining network. This will inevitably hamper the effectiveness of a combining network [8], and also introduces increased delay due to the extra hardware. It is difficult to determine at what system size requirements necessary to prove the hardware combining network to be the optimal method of speeding up hot-spot requests. The effect of the somewhat slower hot-spot requests will have on total system performance, if the rate is very small, also remains to be seen.

## REFERENCES

[1] D.D. Gajski, D.J. Kuck, D.H. Lawrie, and A.H. Sameh, "Cedar - A Large Scale Multiprocessor," *Proc. 1983 Int'l Conf. Parallel Processing,* Aug. 1983, pp. 524-529.

[2] A. Gottlieb, et al., "The NYU Ultracomputer - Designing a MIMD, Shared Memory Parallel Machine," *IEEE Trans. Computers,* Vol. C-32, pp. 175-189, Feb. 1983.

[3] G. F. Pfister, et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 Int'l Conf. Parallel Processing,* Aug. 1985, pp. 764-771.

[4] D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers,* Vol. C-24, pp. 1145-1155, Dec. 1975.

[5] C.L. Wu and T.Y. Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. Computers, Vol. C-29, pp. 694-702, Aug. 1980.

[6] G.F. Pfister and V.A. Norton, "'Hot-Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Trans. Computers,* Vol. C-34, pp. 943-948, Oct. 1985.

[7] H. Sullivan, T. Bashkow, and D. Klappholtz, "A Large Scale Homogeneous, Fully Distributed Parallel Machine," *Proc. Fourth Ann. Symp. Computer Architecture,* June 1977, pp. 105-124.

[8] G. Lee, C.P. Kruskal, and D.J. Kuck, "The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot Spot'," to appear in *Proc. 1986 Int'l Conf. Parallel Processing.*

[9] D.M. Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Networks," *IEEE Trans. Computers,* Vol. C-30, pp. 273-282, Apr. 1981.

[10] C.Q. Zhu and P.C. Yew, "A Synchronization Scheme and Its Applications for Larger Multiprocessor Systems," *Proc. 4th Int'l Conf. Distributed Computing Systems,* May 1984, pp. 486-493.

[11] P. Tang, P.C. Yew, and C.Q. Zhu, "Processor Self-Scheduling in Large Multiprocessor Systems," University of Illinois at Urbana-Champaign, Center for Supercomputing R&D, Cedar Doc. 536, Dec. 1985.

[12] E.L. Lusk and R.A. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Argonne National Laboratory, Argonne, Illinois, Technical Report ANL-83-97, Dec. 1983.

[13] P.Y. Chen, "Multiprocessor Systems: Interconnection Networks, memory Hierarchy, Modeling and Simulations," University of Illinois at Urbana-Champaign, Department of Computer Science, Report No. UIUCDCS-R-82-1083, Jan. 1982.

[14] R. Lee, "On Hot Spot Contention," *ACM SIG Computer Architecture,* Vol. 13, pp. 15-20, Dec. 1985.

[15] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Comm. ACM 8,* pp. 569-569, Sept. 1965.

[16] C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", *IEEE Trans. Computers,* Vol. C-32, pp. 1091-1098, Dec. 1983.

# SIMULTANEOUS ITERATIONS ALGORITHM FOR GENERAL EIGENVALUE PROBLEMS ON PARALLEL PROCESSORS+

S. Utku* and Y. Chang*
Duke University, Durham, N.C. 27706

M. Salama** and D. Rapp**
Jet Propulsion Laboratory
California Institute of Technology, Pasadena, CA 91109

## Abstract

The method of simultaneous iteration with shift is extended to extraction of m-eigenpairs of a general eigenvalue problem of large order n, n>>m, in a parallel processing environment. The algorithm combines the power method and the Jacobi technique and reduces to performing four basic operations: decomposing a banded matrix of order n into upper and lower triangular factors; forming a product of matrices that may be banded, rectangular or square; extracting all eigenpairs of m-order problems by a generalized Jacobi technique; and forward/backward substitution.

Parallel implementation of the algorithm is discussed in detail. The analysis accounts for computation and communication costs, and utilizes a parallel processing architecture of the ensemble type. Expressions for the computational efficiency and speedup are defined as a function of the problem and hardware parameters. Selected representative problems exhibit efficiencies ranging from 60% to 98%.

## 1. Introduction

Parallel processing is particularly attractive for large computationally intensive problems where hours of sequential computing might be reduced to minutes of parallel computing. Unlike vector computers such as the Cray or Cyber 205, parallel computers of the ensemble type, such as the Cosmic Cube[1] and the CHIP[2], require the use of computational strategies that permit decomposing the solution of a problem into concurrently executable computational tasks (or processes). Each task is executed on one processor which is a member of a network of communicating processors that operate in parallel. Communication between tasks is inevitable, since all tasks contribute to the solution of a single problem. However, such communications must be minimized in order to achieve the full potential of these machines. This offers the challenge of restructuring existing algorithms and discovering others so that they are best suited for the parallel processing environment.

To this end, the present work addresses a strategy for the parallel extraction of the eigenpairs of the general eigenvalue problem:

$$\underset{\sim}{A}\,\underset{\sim}{x} = \lambda\,\underset{\sim}{B}\,\underset{\sim}{x} \tag{1}$$

where $\underset{\sim}{A}$ and $\underset{\sim}{B}$ are Hermitian banded matrices of large order n and half bandwidth b<<n. As is the case in many technologically important problems, the interest here is in computing only the set of eigenpairs $(\lambda_\ell, x_\ell)$, $\ell$=1,...,m residing within a prescribed range $\lambda_L \leq \lambda_\ell \leq \lambda_U$, where m<<n. For this class of problems, a mixture of power methods and the Jacobi technique appears to be most suitable, and is therefore explored herein.

## 2. Power Methods

When $\underset{\sim}{B}$ is positive definite, the general eigenvalue problem in (1) may be reduced to the standard form:

$$\underset{\sim}{\mathcal{A}}\;\underset{\sim}{x} = \lambda\,\underset{\sim}{x} \quad ; \quad \underset{\sim}{\mathcal{A}} = \underset{\sim}{B}^{-1}\,\underset{\sim}{A} \tag{2}$$

Furthermore, if $\underset{\sim}{Q}(\underset{\sim}{\mathcal{A}})$ and $\underset{\sim}{R}(\underset{\sim}{\mathcal{A}})$ denote any polynomials of $\underset{\sim}{\mathcal{A}}$, it can be shown[3] that $(\underset{\sim}{Q}(\underset{\sim}{\mathcal{A}})\;\underset{\sim}{R}(\underset{\sim}{\mathcal{A}})^{-1})^r$, r = integer, has the same eigenvectors as $\underset{\sim}{\mathcal{A}}$, with eigenvalues $(Q(\lambda_i)R(\lambda_i)^{-1})^r$, provided that det $\underset{\sim}{R}(\underset{\sim}{\mathcal{A}}) \neq 0$. By starting with any vector $\underset{\sim}{v}$, iterations of the type

$$\begin{aligned} &\underset{\sim}{Q}(\underset{\sim}{\mathcal{A}})\;\underset{\sim}{R}(\underset{\sim}{\mathcal{A}})^{-1}\;\underset{\sim}{u}_{k-1} = \underset{\sim}{v}_k \\ &\underset{\sim}{u}_k = \underset{\sim}{v}_k / \max\,(\underset{\sim}{v}_k) \qquad k = 1,2,\ldots \end{aligned} \tag{3}$$

lead to

$$\lim_{k \Rightarrow \infty} \underset{\sim}{u}_k = \underset{\sim}{x}_p \tag{4}$$

where $\underset{\sim}{x}_p$ is the p<u>th</u> eigenvector of (1) such that

$$\|Q(\lambda_i)/R(\lambda_i)\| \max_{i=1,2,\ldots} = \|Q(\lambda_p/R(\lambda_p)\| \tag{5}$$

The algorithm implied by (3) is a power method which enables one to extract the eigenpairs of (1) selectively. For example, by taking

$$\underset{\sim}{Q}(\underset{\sim}{\mathcal{A}}) = \underset{\sim}{I} \quad ; \quad \underset{\sim}{R}(\underset{\sim}{\mathcal{A}}) = (\underset{\sim}{\mathcal{A}} - \lambda_o \underset{\sim}{I}) \tag{6}$$

where $\lambda_o$ is a prescribed shift, (3) and (5) reduce to the method of inverse iteration with shift:

[+] The paper presents research results carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA).

[*] Department of Civil Engineering and Computer Science

[**]Mechanical and Chemical Systems Division

0190-3918/86/0000/0059 $01.00 © 1986 IEEE

$$(A - \lambda_o B)\, v_k = B\, u_{k-1}$$

$$u_k = v_k/\max(v_k) \qquad k = 1,2,\ldots \tag{7}$$

and $$\|1/(\lambda_i - \lambda_o)\|\max_{i=1,2,\ldots} = \|1/(\lambda_p - \lambda_o)\| \tag{8}$$

The iterations will converge to the eigenvectors with eigenvalues closest to the shift $\lambda_o$.

It is known that when $A$ and $B$ are Hermitian (i.e., $A=A^H$, $B=B^H$), all eigenvalues $\lambda_i=\bar{\lambda}_i$, $i = 1,2,\ldots,n$ are real; and if $A$ and $B$ are also positive definite then $\lambda_i > 0$ for $i = 1,2,\ldots,n$. When $A$ and $B$ are Hermitian, if the labels of the eigenvalues are such that $\lambda_1 < \lambda_2 < \ldots < \lambda_n$, then by Sylvester's law of inertia of quadratic forms[3], it may be shown that the index p of the converged eigenpair is such that:

$$p = \eta' \quad , \quad \text{if} \quad \lambda_p < \lambda_o \tag{9}$$

$$p = \eta' + 1 \; , \quad \text{if} \quad \lambda_p > \lambda_o$$

where $\eta'$is the number of negative diagonals $d_i$ in the factorization:

$$(A - \lambda_o B) = U'^H \operatorname{diag}(d_i)\, U' \tag{10}$$

Equivalently, $\eta'$is the number of pure imaginary diagonals of the upper triangular matrix $U$ in the Cholesky factorization of:

$$(A - \lambda_o B) = U^H U \tag{11}$$

The algorithm given by (6), (7) and (8) can also be carried out with m iteration vectors, $V_k$, simultaneously. In this case, instead of the normalization in (7) one may orthonormalize the set of m vectors, $V_k$, with respect to $A$ and $B$. This can be done either by Gram-Schmidt, Givens or Householder orthogonalization; or by replacing $V_k$ with an orthogonal set of m-basis vectors $U_k$ which span the same subspace, $S_k$, as $V_k$. In the latter scheme, the orthogonality property is derived from satisfaction of the relation between the projected operators $A$ and $B$ onto the $S_k$ subspace. This is accomplished by the following recursive relationship:

$$(A - \lambda_o B)\, V_k = B\, U_{k-1} \tag{12}$$

and the transformation

$$U_k = V_k\, Q_k \qquad k = 0,1,2,\ldots,K_{\max} \tag{13}$$

Initially, for $k = 0$, $V_k$ is any set of m linearly independent vectors. For succeeding steps, $Q_k$ is determined so as to satisfy the eigenvalue problem of order m<<n:

$$A_k\, Q_k = B_k\, Q_k \operatorname{diag}(\Omega_k) \tag{14}$$

in which,

$$A_k = V_k^H (A - \lambda_o B)\, V_k \quad ; \qquad B_k = V_k^H\, B\, V_k$$

Normalization of $Q_k$ with respect to $B_k$, i.e., ($Q_k B_k Q_k = I$) is enforced such that

$$U_k^H\, B\, U_k = I \tag{15}$$

With Equations (12) to (15), the resulting eigenpairs are assured to converge to:

$$\lim_{k \Rightarrow \infty} U_k = [x_{1\ell}, x_{2\ell}, \ldots, x_{m\ell}]_{K_{\max}} \tag{16}$$

$$\lim_{k \Rightarrow \infty} \operatorname{diag}(\Omega_k) = \operatorname{diag}(\lambda_{1\ell} - \lambda_o, \lambda_{2\ell} - \lambda_o, \ldots, \lambda_{m\ell} - \lambda_o)_{K_{\max}} \tag{17}$$

where the indices $1\ell, 2\ell, \ldots, m\ell$ are those of $\lambda_i$ of (1) which reside nearest the shift $\lambda_o$.

## 3. Generalized Jacobi Technique

The generalized Jacobi technique[4] is most suitable for obtaining all eigenpairs of the reduced general eigenvalue problem in (14). In the generalized Jacobi, both $A_k$ and $B_k$ are diagonalized simultaneously by a sequence of orthogonal transformations of the form:

$$A_{k,r+1} = P_r^T \cdots P_2^T P_1^T A_k P_1 P_2 \cdots P_r$$

$$B_{k,r+1} = P_r^T \cdots P_2^T P_1^T B_k P_1 P_2 \cdots P_r \tag{18}$$

where $P_r$ is a rotation matrix:

$$P_r = \begin{array}{c} \left[\begin{array}{cccccccc} 1 & & \cdot & & & & \cdot & \\ & 1 & \cdot & & & & \cdot & \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \alpha_{ij} & \cdot \\ & & \cdot & 1 & & & \cdot & \\ & & \cdot & & 1 & & \cdot & \\ & & \cdot & & & 1 & \cdot & \\ \cdot & \cdot & \beta_{ij} & \cdot & \cdot & \cdot & 1 & \cdot \\ & & \cdot & & & & \cdot & 1 \end{array}\right] \begin{array}{c} \\ \\ i \\ \\ \\ \\ j \\ \\ \end{array} \\ \begin{array}{cccccccc} \quad & \quad & i & \quad & \quad & \quad & j & \quad \end{array} \end{array} \tag{19}$$

Each transformation in (18) involves one pre- and post- multiplication, and will simultaneously reduce to zero the symmetric off-diagonal terms (ij) and (ji) of $A_{k,r+1}$ and $B_{k,r+1}$ if:

$$\alpha_{ij} = \omega_1/\theta \quad ; \quad \beta_{ij} = -\omega_2/\theta$$

$$\text{where} \quad \theta = \frac{\omega_3}{2} + (\operatorname{sign}\omega_3)\left[\left(\frac{\omega_3}{2}\right)^2 + \omega_1\,\omega_2\right]^{1/2}$$

$$\omega_1 = (A_{jj}\, B_{ij} - B_{jj}\, A_{ij})_{k,r}$$

$$\omega_2 = (A_{ii}\, B_{ij} - B_{ii}\, A_{ij})_{k,r} \tag{20}$$

$$\omega_3 = (A_{ii}\, B_{jj} - B_{ii}\, A_{jj})_{k,r}$$

A single transformation as defined above will result in changing only the row and column pairs of $A_k$ and $B_k$ corresponding to the ij pair of Equation (19) in the manner listed in Table (1). All remaining terms of $A_{k,r}$ and $B_{k,r}$ remain unchanged, until subsequently operated upon by a transformation involving their row or column pair. In the "cyclic" generalized Jacobi technique used here, no attempt is made to determine which off-diagonal term is largest so

TABLE (1)

Changes in the i,j rows and columns of $A_{k,r+1}$ and $B_{k,r+1}$ due to a single transformation using $P_r$ of Equation (19)

| Element | $P_r^T A_{k,r} P_r$ | $P_r^T B_{k,r} P_r$ |
|---|---|---|
| ii | $(A_{ii} + \beta_{ji}^2 A_{jj} + 2\,\beta_{ji} A_{ij})_{k,r}$ | $(B_{ii} + \beta_{ji}^2 B_{jj} + 2\,\beta_{ji} B_{ij})_{k,r}$ |
| jj | $(A_{jj} + \alpha_{ij}^2 A_{ii} + 2\,\alpha_{ij} A_{ij})_{k,r}$ | $(B_{jj} + \alpha_{ij}^2 B_{ii} + 2\,\alpha_{ij} B_{ij})_{k,r}$ |
| ij = ji | 0 | 0 |
| $\delta = 1,2,\ldots,m$ ; $\delta \neq i$ , $\delta \neq j$ | | |
| $i\delta = \delta i$ | $(A_{i\delta} + \beta_{ji} A_{j\delta})_{k,r}$ | $(B_{i\delta} + \beta_{ji} B_{j\delta})_{k,r}$ |
| $j\delta = \delta j$ | $(A_{j\delta} + \alpha_{ij} A_{i\delta})_{k,r}$ | $(B_{j\delta} + \alpha_{ij} B_{i\delta})_{k,r}$ |

as to reduce it to zero first. Instead, all off-diagonal terms are operated upon in row major order, sweep after sweep. As such, diagonalizing the m x m matrices $A_k$ and $B_k$ of (14) will require several sweeps, $J_{max}$; each sweep consisting of m(m-1)/2 orthogonal transformations of the type in (19). As the number of sweeps increases, both $A_{k,r+1}$ and $B_{k,r+1}$ will approach their diagonal form, in which case the eigenpairs are simply:

$$\mathrm{diag}\,(\Omega_k) \simeq \mathrm{diag}\,(A_{ii}/B_{ii})_{k,r+1} \qquad (21)$$
$$Q_k \simeq P_1\, P_2 \,\cdots\, P_r\ \mathrm{diag}\,(1/\sqrt{B_{ii}})_{k,r+1}$$

## 4. Parallel Implementation of Simultaneous Iteration

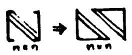
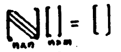
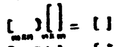
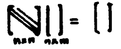
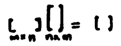
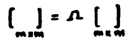
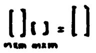
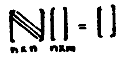
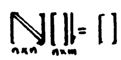
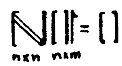
Preliminaries:

The steps of the algorithm described in Section 2 are summarized in Table (2). The algorithm consists of four essential operations: (i) decomposing a banded Hermitian matrix of large order n into upper and lower triangular factors; (ii) forming the product of matrices that may be banded, rectangular or square; (iii) extracting all eigenpairs of the m-order problem by a generalized Jacobi and (iv) solving for the unknowns of an algebraic set of equa-tions by a forward pass, and then by a backward pass.

The objective of this section is to construct parallel implementation strategies for the above steps, and to derive expressions for the speedup and efficiency of the individual steps and of the complete algorithm. The speedup and efficiency are the most common measures for evaluating the performance of algorithms on N-identical processors operating in parallel. Both measures are related. The speedup is the ratio of the time required to execute a given algorithm on a single processor computer to the time required to execute the same algorithm on a parallel processing computer composed of N processors of the same type. The efficiency is defined as the speedup per processor. Equivalently, the efficiency is the ratio of time spent in computation to the total time required for computation, interprocessor communication and idling, if any.

TABLE (2)

Simultaneous Iteration Algorithm

| Solution Step | Comments |
|---|---|
| 1. Obtain the triangular factors of $(A-\lambda_o B) = U^H U$ |  |
| 2. Choose m-independent vectors $V_o = [v_1, v_2, \ldots, v_m]_o$ and set diag $(\Omega)_{-1} = 0$ | |
| 3. Construct the product $Z_{k-1} = (A - \lambda_o B) V_k$, k=0 |  |
| 4. Compute $A_k = V_k^H Z_{k-1}$ |  |
| 5. $W_k = B V_k$ |  |
| 6. $B_k = V_k^H W_k$ |  |
| 7. Extract m orthonormal eigenpairs of $A_k Q_k = B_k Q_k$ diag $(\Omega_k)$ |  |
| 8. Compute $U_k = V_k Q_k$ |  |
| 9. Convergence check: if $\|(\Omega_k - \Omega_{k-1})/\Omega_k\|_{ii} < \epsilon$, set $U_k = [x_1, x_2, \ldots, x_m]_k$<br>diag$(\Omega_k + \lambda_o)$ = diag$(\lambda_1, \ldots, \lambda_m)_k$, exit otherwise continues | |
| 10. Form $Z_k = B U_k$ |  |
| 11. With a forward pass compute $Y_{k+1}$ from $U^H Y_{k+1} = Z_k$ |  |
| 12. With a backward pass compute $V_{k+1}$ from $U V_{k+1} = Y_{k+1}$ |  |
| 13. Set k <-- k+1, and go to 4 | |

To quantify the complete processing costs, two architectural parameters, q and $\mu$, are used. The computational cost is characterized by q, which is defined as the time required to perform a floating point operation. One floating point operation is assumed, on the average, to consist of a multiplication followed by an addition. The cost of interprocessor communication is characterized by $\mu$, which is defined as the ratio of the time required to transmit one floating point number from one processor to its nearest neighbor, to the time required to perform a floating point operation. In the current state-of-the-art computers, $\mu$ is of order unity. The cost of propagating data through a network of N-processors depends on the transmission speed $\mu q$, communication topology among processors, the number of data items to be transmitted, and whether all processors should receive the same data or each should receive different data. In computers such as the Cosmic Cube, the time required for transmitting $\nu$ floating point numbers to N-processors is[5]:

$$T_1 = \nu(\log_2 N)\,\mu q \qquad (22)$$

if the same $\nu$ data items are sent to all N processors, and

$$T_2 = (\nu + N - 2)\,\mu q \qquad (23)$$

if different processors are sent different sets of $\nu$ data items each.

In the following, operations on matrices of large order are reduced to operations on smaller partitions of size s. It will be assumed that 1<s<<b. Further, by proper padding with zeros:

$$\mathrm{Mod}(n,s) = 0$$
$$\mathrm{Mod}(b,N) = 0$$
$$\mathrm{Mod}(b,s) = 0$$

and

$$\begin{aligned} b &= 2Ns \ll n \\ n_s &= n/s \\ b_s &= b/s = 2N \\ n_N &= 1 + \mathrm{Int}\,((n_s-1)/N) \end{aligned} \tag{24}$$

i. Parallel Matrix Decomposition

Two strategies were given in Reference [6] for the parallel implementation of decomposition of a banded Hermitian matrix of large order, n, into its triangular upper and lower factors $\underset{\sim}{U},\underset{\sim}{U}^H$. These stratagems use two different interpretations of the Cholesky factorization. Their performance is nearly equal. With the definitions and notations outlined above, the time required to factor an n x n matrix with half bandwidth b using N-processors is [6]:

$$T_i \approx qns^2N\,[2(1 + \frac{1}{N}) + \frac{\mu}{s}\,(10 + \frac{1}{N})] \tag{25}$$

where the coefficient of $\mu$ is due to interprocessor transmission of data. Expression (25) as well as others to be developed in subsequent sections are required for estimating the efficiency of the entire algorithm.

ii. Matrix Product

Three schemes for performing the matrix product $\underset{\sim}{C} = \underset{\sim}{A}^H\underset{\sim}{B}$ are given in this section, depending on whether $\underset{\sim}{A}$ and $\underset{\sim}{B}$ are banded, square or rectangular.

a. Inner Product of Two Rectangular Matrices

Let $\underset{\sim}{A}$ and $\underset{\sim}{B}$ be n x m matrices, each of which is partitioned into blocks $\underset{\sim}{A}_i$, $\underset{\sim}{B}_i$; $i = 1,\ldots,n_s$ with block size s x m. Assuming that each of the N-processors can store ($2ms+m^2$) floating point numbers, the inner product may be expressed by:

$$\underset{\sim}{C} = \sum_{\ell=1}^{N} \underset{\sim}{C}_\ell \tag{26}$$

where

$$\underset{\sim}{C}_\ell = \sum_{k=1}^{n_N} \underset{\sim}{A}^H_{(k-1)N+\ell}\;\underset{\sim}{B}_{(k-1)N+\ell} \tag{27}$$

or

$$\underset{\sim}{C}_\ell = \sum_{k=1}^{n_N} \underset{\sim}{A}^H_{(\ell-1)n_N+k}\;\underset{\sim}{B}_{(\ell-1)n_N+k} \tag{28}$$

The choice between placing the operands $\underset{\sim}{A}_i$ and $\underset{\sim}{B}_i$ in the N processors according to $i = (k-1)N + \ell$ of Equation (27), or $i = (\ell-1)n_N + k$ of Equation (28) may be dictated by the step previous to the matrix multiplication. With N processors, N values of $\underset{\sim}{C}_\ell$, $\ell=1,\ldots,N$ can be computed in parallel. Each processor performs $n_N$ serial floating point operations on their operands. No interprocessor communication is needed, and all $\underset{\sim}{C}_\ell$ are completed in a time of order $n_Nm^2q(1+s)$. The cascade sum of all $\underset{\sim}{C}_\ell$ is performed next according to Equation (26) in a time proportional to $m^2q(1+\mu)\log_2N$ during which the final results are collected in the host computer. The total cost of the inner matrix product is, therefore:

$$T^a_{ii} \approx m^2q\;[n_N\;(1 + s) + (1 + \mu)\;\log_2 N] \tag{29}$$

b. Product of a Rectangular Matrix by Square Matrix

Let $\underset{\sim}{A}$ and $\underset{\sim}{B}$ denote n x m and m x m matrices, respectively. Again, assuming that each processor can hold ($2ms+m^2$) floating point numbers, and that only $\underset{\sim}{A}$ is partitioned into s x m blocks $\underset{\sim}{A}_i$; $i=1,\ldots,n_s$, the product may be written as:

$$\underset{\sim}{C}_i = \underset{\sim}{A}_i\underset{\sim}{B} \tag{30}$$

where the resulting product $\underset{\sim}{C}$ is conformably partitioned as $\underset{\sim}{A}$ into $\underset{\sim}{C}_i$ blocks. Since $n_s>N$, the parallel implementation of Equation (30) can be accomplished by assigning each processor the task of computing $n_N$ of the $\underset{\sim}{C}_i$ partitions according to either

$$\underset{\sim}{C}_\ell = \underset{\sim}{A}_{(k-1)N+\ell}\;\underset{\sim}{B} \quad \text{or} \quad \underset{\sim}{C}_\ell = \underset{\sim}{A}_{(\ell-1)n_N+k}\;\underset{\sim}{B} \tag{31}$$
$$k = 1,2,\ldots,n_N$$

To begin with, the same $\underset{\sim}{B}$ and the appropriate partition of $\underset{\sim}{A}$ as defined by Equation (31) must reside in each processor. Then all processors perform their products simultaneously in a time of order ($n_Nm^2sq$). The final results, now available in N partitions in the N-processors may be transmitted to the host computer in a time proportional to $\mu q(sm+N-2)$. Therefore, the total cost is:

$$T^b_{ii} \approx q\;[n_N\;m^2s + \mu\;(sm + N - 2)] \tag{32}$$

c. Product of Banded by Rectangular Matrix

Here $\underset{\sim}{A}$ is n x n banded matrix with half bandwidth, b, partitioned into $n_s$ x $n_s$ square blocks according to Equation (24). On the other hand, $\underset{\sim}{B}$ is rectangular n x m partitioned row-wise into $n_s$ rectangular blocks, each s x m. For parallel implementation, the product $\underset{\sim}{C} = \underset{\sim}{A}\underset{\sim}{B}$ may be written as:

$$\underset{\sim}{C}_i = \sum_{\ell=1}^{N} \underset{\sim}{C}_{i\ell} \quad ; \quad \underset{\sim}{C}_{i\ell} = \sum_{k=1}^{4} \underset{\sim}{A}_{ij}\;\underset{\sim}{B}_j \tag{33}$$
$$i=1,\ldots,n_s;\; j=i-2N+4(\ell-1)+k;\; \text{and } \ell=1,\ldots,N$$

For a given row partition i, the appropriate s x s partition $\underset{\sim}{A}_{ij}$ and s x m partition of $\underset{\sim}{B}_j$ corresponding to k=1 must reside in the appropriate processor $\ell=1,\ldots,N$ according to the indices above. The product and partial sum defined by Equation (33) are then computed in parallel using N processors. When this is repeated sequentially for k=2, 3 and 4, one obtains the results of the ith row partition of $\underset{\sim}{C}$. Other row partitions are processed in sequence for all $i=1,\ldots,n_s$ and the final sum is continually accumulated in one processor. The completed results are performed in time

$$T^{c}_{ii} = nmq\ (4s + \mu \log_2 N) \qquad (34)$$

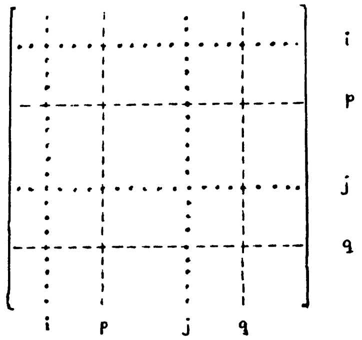
iii. Parallel Generalized Jacobi for Extracting the Eigenpairs

The fact that each of the (m/2)(m-1) transformations of (18) and (19) systematically annihilates the ij,ji terms of $\underset{\sim}{A}_{k,r+1}$ and $\underset{\sim}{B}_{k,r+1}$, and thereby causes only their respective ith and jth rows and columns to change, suggests considerable concurrency in a parallel implementation of the generalized Jacobi. By selecting a set of m/2 non-overlapping pairs of i,j indices, it is possible to perform m/2 parallel transformations on $\underset{\sim}{A}_{k,r}$ and m/2 similar transformations on $\underset{\sim}{B}_{k,r}$. The process is repeated (m-1) times, each time using a different set of non-overlapping indices. Completion of all (m/2)(m-1) transformations constitutes a typical Jacobi sweep that may be repeated until diagonalization is achieved.

For example, the first set of m/2 transformations may be $P_1(1,2)$, $P_2(3,4)$,..., $P_{m/2}(m-1,m)$. Their execution will result in:

$$\underset{\sim}{A}_{k,r+m/2} =$$
$$\underset{\sim}{P}^T_{m/2}(m-1,m) \ldots \underset{\sim}{P}^T_2(3,4)\ \underset{\sim}{P}^T_1(1,2)\ \underset{\sim}{A}_{k,r}\ \underset{\sim}{P}_1(1,2)\ \underset{\sim}{P}_2(3,4) \ldots \underset{\sim}{P}_{m/2}(m-1,m)$$
$$\underset{\sim}{B}_{k,r+m/2} = \qquad (35)$$
$$\underset{\sim}{P}^T_{m/2}(m-1,m) \ldots \underset{\sim}{P}^T_2(3,4)\ \underset{\sim}{P}^T_1(1,2)\ \underset{\sim}{B}_{k,r}\ \underset{\sim}{P}_1(1,2)\ \underset{\sim}{P}_2(3,4) \ldots \underset{\sim}{P}_{m/2}(m-1,m)$$

Suppose that N=m/2 processors are used to perform Equation (35). Each processor may be assigned one pair of the (1,2),(3,4),... indices, for which it needs to store only the corresponding row and column pairs of $\underset{\sim}{A}_{k,r}$ and $\underset{\sim}{B}_{k,r}$; i.e., a total of 4m values. Considering any two processors $\eta$ and $\xi$, the interaction between any two transformations $\underset{\sim}{P}_\eta(i,j)$ and $\underset{\sim}{P}_\xi(p,q)$ of the m/2 transformations in (35) is visualized schematically by the dotted (i,j) and dashed (p,q) rows and columns in (36):



(36)

As indicated by Table (1), the ii,jj,ij, and ji terms of rows i and j residing in processor $\eta$ are affected only by transformation with $P_\eta(i,j)$. However, all other i$\delta$ and j$\delta$ terms ($\delta$=1,...,m; $\delta \neq i \neq j$) are affected by column changes stemming from the remaining [(m/2)-1] transformations. For example, the ip, iq, jp, jq terms are changed not only by $P_\eta(i,j)$, but also by $P_\xi(p,q)$, thus becoming:

$$(A_{ip})_{k,r+m/2} = A'_{ip} + \beta_{qp} A'_{iq}$$
$$(A_{iq})_{k,r+m/2} = A'_{iq} + \alpha_{pq} A'_{ip}$$
$$(A_{jp})_{k,r+m/2} = A'_{jp} + \beta_{qp} A'_{jq} \qquad (37)$$
$$(A_{jq})_{k,r+m/2} = A'_{jq} + \alpha_{pq} A'_{jp}$$

where from Table (1), $\delta$ assumes p or q:

$$A'_{ip} = (A_{ip} + \beta_{ji} A_{jp})_{k,r}\ ; A'_{jp} = (A_{jp} + \alpha_{ij} A_{ip})_{k,r}$$
$$A'_{iq} = (A_{iq} + \beta_{ji} A_{jp})_{k,r}\ ; A'_{jq} = (A_{jq} + \alpha_{ij} A_{iq})_{k,r}$$

However, because the m/2 indices do not overlap, each of the [(m/2)-1] transformations affect different terms of rows i and j.

Instead of computing the eigenvectors at the end of the last sweep according to (21), data transmission can be eliminated if the eigenvectors are continuously updated at the same time that $\underset{\sim}{A}_k$ and $\underset{\sim}{B}_k$ are transformed by the P's. Only the simple scaling by diag $(1/\sqrt{B_{ii}})_{k,r+1}$ is deferred to the end. Similar to (35), an intermediate matrix of eigenvectors $\underset{\sim}{Q}_k$ is updated by m/2 transformations $\underset{\sim}{P}_1,\ldots,\underset{\sim}{P}_{m/2}$:

$$\underset{\sim}{Q}_{k,r+m/2} = \underset{\sim}{Q}_{k,r}\ \underset{\sim}{P}_1\ \underset{\sim}{P}_2 \cdots \underset{\sim}{P}_{m/2} \qquad (38)$$

In a parallel implementation of Equation (38), using m/2 processors each storing a row pair $\underset{\sim}{Q}_{i\delta}$, $\underset{\sim}{Q}_{j\delta}$, $\delta$=1,...,m with non-overlapping i and j indices, each processor updates four terms associated with each of the m/2 transformations. For example, the following four terms of the i,j row pair residing in any of the m/2 processors will be changed by a typical transformation $\underset{\sim}{P}_\xi(p,q)$:

$$(Q_{ip})_{k,r+m/2} = (Q_{ip} + \beta_{qp} Q_{iq})_{k,r}$$
$$(Q_{iq})_{k,r+m/2} = (Q_{iq} + \alpha_{pq} Q_{ip})_{k,r}$$
$$(Q_{jp})_{k,r+m/2} = (Q_{jp} + \beta_{qp} Q_{jq})_{k,r} \qquad (39)$$
$$(Q_{jq})_{k,r+m/2} = (Q_{jq} + \alpha_{pq} Q_{jp})_{k,r}$$

Again, since the m/2 indices do not overlap, the transformations in (39) affect different terms of rows i and j of $\underset{\sim}{Q}$. All processors can perform Equation (39) simultaneously on different rows. Thus, with N=m/2 processors, each storing two rows of each of $\underset{\sim}{A}_k$, $\underset{\sim}{B}_k$ and $\underset{\sim}{Q}_k$ corresponding to a pair of non-overlapping indices, the steps of the parallel generalized Jacobi technique are summarized as follows:

1. Each of the m/2 processors computes one pair of rotation coefficients $\alpha_{ij}$ and $\beta_{ji}$ according to (20). This requires 12q computational resources.

2. Each processor sends the two values of $\alpha_{ij}$, $\beta_{ji}$ of Step (1) to all other [(m/2)-1] processors in the network. In doing so, (m/2)$\mu$q resources are consumed.

3. Each processor updates: (a) its ii and jj values of the current rows of $\underset{\sim}{A}_k$ and $\underset{\sim}{B}_k$ according to the first two expressions of Table (1) using its own $\alpha_{ij}$ and $\beta_{ji}$; (b) the remaining terms of its current rows of $\underset{\sim}{A}_k$ and $\underset{\sim}{B}_k$ having column indices pq that match the rotation coefficients supplied by other processors in Step (2). Equations (37) are used for this purpose; (c) all terms of the stored pair of rows of $\underset{\sim}{Q}_k$ having column indices which match the m/2 pair of rotation coefficients. This is done according to Equation (39).

This 3-part step requires (12+12m+2m)q resources.

4. A new set of m/2 pairs of rows with unique non-overlapping indices is acquired by the m/2 processors. This may be done by a systematic row exchange according to the scheme described in Appendix (A). According to this scheme, corresponding rows of $\underset{\sim}{A}_k$ and $\underset{\sim}{B}_k$ are exchanged among processors, while the $\underset{\sim}{Q}_k$ rows remain stationary in their original processors. This requires 2m$\mu$q resources.

5. One Jacobi sweep is completed when Steps (1) to (4) are repeated (m-1) times, at which time the current values of off-diagonals in the current rows of $\underset{\sim}{A}_k$ and $\underset{\sim}{B}_k$ are tested for smallness according to:

$$(A^2_{i\delta}/A_{ii}\ A_{jj})_k \le 10^{-\varepsilon} \quad ; \quad (A^2_{j\delta}/A_{ii}\ A_{jj})_k \le 10^{-\varepsilon}$$

$$(B^2_{i\delta}/B_{ii}\ B_{jj})_k \le 10^{-\varepsilon} \quad ; \quad (B^2_{j\delta}/B_{ii}\ B_{jj})_k \le 10^{-\varepsilon}$$

$$\delta = 1,2,\ldots,m \quad ; \quad \delta \ne i \ne j \tag{40}$$

A "pass" is reported to halt the sweeps when all off- diagonals are small enough, and when all eigenvalues computed according to Equation (21) pass the convergence test

$$\|\Omega^J_{ii} - \Omega^{J-1}_{ii}/\Omega^{J-1}_{ii}\|_k \le 10^{-\varepsilon} \tag{41}$$

A total of 12mq resources are required each time the above convergence tests are performed.

6. Final collection of the converged eigenvalues and vectors from all N=m/2 processors requires 2.5 m$\mu$q resources.

Assuming that on the average $J_{max}$ - sweeps are needed for convergence, computation of the m eigenvalues and vectors by the parallel generalized Jacobi requires total computer time of the order:

$$T_{iii} \simeq m^2q\ (14 + 2.5\ \mu)\ J_{max} \tag{42}$$

iv. <u>Forward and Backward Passes</u>

When solving for the n x m unknowns $\underset{\sim}{X}$ in

$$\underset{\sim}{U}^H\underset{\sim}{U}\underset{\sim}{X}=\underset{\sim}{C} \tag{43}$$

a forward pass is first required to find $\underset{\sim}{Y}$, then a backward pass is performed to compute $\underset{\sim}{X}$;

$$\underset{\sim}{U}^H\underset{\sim}{Y}=\underset{\sim}{C} \quad ; \quad \underset{\sim}{U}\underset{\sim}{X}=\underset{\sim}{Y} \tag{44}$$

All $\underset{\sim}{A}$, $\underset{\sim}{U}$,$\underset{\sim}{U}^H$ are n x n with half bandwidth $b_s$ partitioned into $n_s$ rows. Each row is partitioned into $b_s$ column blocks of size s x s. Similarly, $\underset{\sim}{C}$, $\underset{\sim}{Y}$ and $\underset{\sim}{X}$ are n x m rectangular matrices, also partitioned into $n_s$ rows $\underset{\sim}{C}_i$, $\underset{\sim}{Y}_i$ and $\underset{\sim}{X}_i$; i=1,...,$n_s$, each of size s x m. From Equations (44):

$$\underset{\sim}{Y}_i = \underset{\sim}{U}^{-1}_{ii}\left[\underset{\sim}{C}_i - \sum_{k=i-b_s+1}^{i-1} \underset{\sim}{U}_{ik}\ \underset{\sim}{Y}_k\right] \tag{45}$$

and

$$\underset{\sim}{X}_i = \underset{\sim}{U}^{-1}_{ii}\left[\underset{\sim}{Y}_i - \sum_{k=i+1}^{i+b_s-1} \underset{\sim}{U}_{ik}\ \underset{\sim}{X}_k\right] \tag{46}$$

With N=$b_s$ /2 processors, Equation (45) may be executed in parallel for a typical row partition i=1,...,$n_s$ in five substeps. To begin with, as a result of performing the forward pass of Equation (45) on the (i-1)<u>th</u> row partition, two sets of $\underset{\sim}{Y}$ and $\underset{\sim}{U}$ partitions will have been stored in each processor (except the first). For example, processor #1 will have stored $\underset{\sim}{C}_i$, $\underset{\sim}{U}_{ii}$, $\underset{\sim}{Y}_{i-1}$, and $\underset{\sim}{U}_{i,i-1}$; processor #2 will have stored $\underset{\sim}{Y}_{i-2}$, $\underset{\sim}{U}_{i,i-2}$, $\underset{\sim}{Y}_{i-3}$ and $\underset{\sim}{U}_{i,i-3}$; and processor #N will have stored $\underset{\sim}{Y}_{i-bs}$, $\underset{\sim}{U}_{i,i-bs}$, $\underset{\sim}{Y}_{i-bs+1}$, and $\underset{\sim}{U}_{i,i-bs+1}$. During the first substep, all processors simultaneously multiply and add their contents. Thus, a typical processor such as #2 computes $-(\underset{\sim}{U}_{i,i-2}\ \underset{\sim}{Y}_{i-2} + \underset{\sim}{U}_{i,i-3}\ \underset{\sim}{Y}_{i-3})$, and #N computes $-(\underset{\sim}{U}_{i,i-bs}\ \underset{\sim}{Y}_{i-bs} + \underset{\sim}{U}_{i,i-bs+1}\ \underset{\sim}{Y}_{i-bs+1})$. The exception is processor #1 which performs $-(-\underset{\sim}{C}_i + \underset{\sim}{U}_{i,i-1}\ \underset{\sim}{Y}_{i-1})$. The time required for the first substep is of order 2ms$^2$, and the result is a partial sum residing in each processor. In the second substep, the cascade sum of the partial sums in all N processors is formed by adding the m x s contents of pairs of processors in the lower-numbered processor of each pair. This requires simultaneous passing of one m x s data block between the next neighbor processors of each pair. As a result, the full cascade sum of the quantity in parenthesis in Equation (45) is readily accumulated in processor #1 in total time of the order (1+$\mu$(msq)log$_2$N.

In the third substep, processor #1 computes $\underset{\sim}{Y}_i$ by multiplying $\underset{\sim}{U}_{ii}$ by the total sum obtained in the second substep in time of order ms$^2$q/2. All other processors remain idle during this substep. The purpose of the last two substeps is to store the needed data in the appropriate processors in preparation for performing the forward pass on the (i+1)<u>th</u> row partition.

Thus in the fourth substep, all $\underset{\sim}{Y}_i$ blocks are shifted one storage location toward the higher-numbered neighboring processor. For example, $\underset{\sim}{Y}_{i-1}$ is transferred from processor #1 to processor #2; $\underset{\sim}{Y}_{i-2}$ replaces $\underset{\sim}{Y}_{i-3}$ in the same #2 processor, while $\underset{\sim}{Y}_{i-3}$ is transferred to processor #3 to replace $\underset{\sim}{Y}_{i-4}$; simultaneously $\underset{\sim}{Y}_{i-4}$ replaces $\underset{\sim}{Y}_{i-5}$ in processor #3 while $\underset{\sim}{Y}_{i-5}$ is transferred to processor #4 to replace

$\underset{\sim}{Y}_{i-6},\ldots$,etc. This is done in parallel by all processors on s x m data blocks in time proportional to msq$\mu$. In the last substep, two s x s blocks ($\underset{\sim}{U}_{ik}$ and $\underset{\sim}{U}_{i,k-1}$) are downloaded from the host computer to each of the N-processors, where $k=i,\ldots,i-b_s+1$. This requires communication time proportional to $(2s^2+N-2)q\mu$. The total time required for performing the forward pass of (45) on $n_s$ row partitions consists of the above five substeps repeated $n_s$ times, and is denoted by $T_{iv}^{F}$.

$$T_{iv}^{F} \simeq mnq\ [\frac{5}{2}\ s + \log_2 N + \mu\ (2 + \log_2 N)] \qquad (47)$$

As can be seen from Equations (45) and (46), the time required for the backward pass $T_{iv}^{B}$ is essentially the same as (47), i.e., $T_{iv}^{B}=T_{iv}^{F}$.

## 5. Algorithm Efficiency and Speedup

By summing the results of Equations (25, 29, 32, 34, 42, and 47), the total time, T, required to complete the algorithm is found to be of order:

$$T \simeq mnq\ [(\alpha_0 + \alpha_1\ K_{max}) + \mu\ (\beta_0 + \beta_1\ K_{max})] \quad (48)$$

where $\alpha_0 = 2s\ (2 + s/m)$

$$\alpha_1 = [13s + 2\ \log_2 N + \frac{m}{n}\ (\frac{3n}{N} + 14\ J_{max} + 2\ \log_2 N)]$$

$$\beta_0 = (\log_2 N + 10s/m)$$

$$\beta_1 = [4(1 + \log_2 N) + \frac{m}{n}\ (2.5\ J_{max} + 2\ \log_2 N)]$$

The coefficients $\alpha_o$ and $\beta_o$, respectively, are computation and communication costs associated with Steps 1 through 3 of Table (2). These are fixed and do not depend on the convergence rate. On the other hand, $\alpha_1$ and $\beta_1$ are, respectively, computation and communication cost per simultaneous iteration $k=1,2,\ldots,K_{max}$. Both $\alpha_1$ and $\beta_1$ are functions of the parameters b,m,n,s,N as well as the maximum number of Jacobi sweeps, $J_{max}$.

According to our previous definitions, the algorithm efficiency $\epsilon^*$ and speedup $g^*$, are:

$$\epsilon^* = \frac{\alpha_0 + \alpha_1\ K_{max}}{(\alpha_0 + \alpha_1\ K_{max}) + \mu\ (\beta_0 + \beta_1\ K_{max})}, \qquad (49)$$

$$\text{and } g^* = \epsilon^* N$$

To illustrate the behavior of the efficiency $\epsilon^*$ in Equation (49), the representative values of problem parameters in Table (3) are considered. Because contributions of the maximum number of required Jacobi sweeps, $J_{max}$, to the coefficients $\alpha_1$ and $\alpha_2$ (and consequently to $\epsilon^*$) is made negligibly small by the small ratio of m/n, $J_{max}$ is arbitrarily taken equal to ten in subsequent discussions of $\epsilon^*$ and $g^*$. However, contributions of the maximum number of simultaneous iterations, $K_{max}$, to $\epsilon^*$ is not as obvious. In Figure (1), the efficiency $\epsilon^*$ is depicted for the three sets of parameters labled "A" in Table (3) for the range of $K_{max}$=0,1,2,5,10. As is observed from Figure (1), the efficiency of the parallel algorithm becomes rather insensitive to values of $K_{max}$ in their upper range. The total time, T, of Equation (48) behaves linearly with $K_{max}$ beyond the initial Steps 1 through 3 of Table (2).

TABLE (3)

Selected Problem Parameters

| | A | | | B | | | C | | |
|---|---|---|---|---|---|---|---|---|---|
| n | 16384 | | | 4096 | | | 4096 | | |
| b | 2048 | | | 512 | | | 256 | | |
| N | 128 | 64 | 32 | 64 | 32 | 16 | 64 | 32 | 16 |
| s | 8 | 16 | 32 | 4 | 8 | 16 | 2 | 4 | 8 |
| m | 256 | 128 | 64 | 128 | 64 | 32 | 128 | 64 | 32 |

In Figure (2), $K_{max}$=10 is used to construct Equation (49) graphically for cases A, B and C of Table (3). As illustrated, the algorithm is more efficient for larger problem order n and bandwidth b. The speedup can readily be obtained from Figure (3) by the simple multiplication $g^*=N\epsilon^*$. For example, using 128 processors in parallel, Case A can be solved more than 100-times faster than with a single processor.

## 6. Conclusions

A parallel algorithm is described for extracting m eigenpairs of generalized eigenvalue problems of large order n>>m. The algorithm combines power methods with a generalized Jacobi technique, both of which are known to have good convergence properties. Concurrency is introduced by partitioning the computational work load among N-processors during an initial one-time factorization step and during successive iterations. As Equation (48) indicates, the first is functionally dependent upon $\alpha_o$ and $\beta_o$ while the second is dependent upon $\alpha_1$ and $\beta_1$. As the number of required iterations $K_{max}$ increase, the iterative part of the calculations dominates the required computational resources as well as the efficiency and speedup of the algorithm. This is illustrated by Figure (1). In terms of efficiency and speedup, the algorithm performs best for problems with larger order n and bandwidth b.

## References

1. Seitz, C.L., "The Cosmic Cube", Communications of the ACM, Vol. 28, No. 1, 1985, pp. 22-33.

2. Snyder, L., "Introduction to the Configurable Highly Parallel Computer", ONR Report No. CSD-TR-351, May 1981.

3. Wilkinson, J.H., "The Algebraic Eigenvalue Problems", London: Oxford University Press, 1965.

4. Bathe, K.J., "Solution Methods for Large Generalized Eigenvalue Problems in Structural Engineering", Report UC SESM 71-20, Univ. of California, Berkeley, CA.

5. Tuazon, J., Peterson, J., Pniel, M. and Liberman, D., "CALTECH/JPL MARK II Hypercube Concurrent Processor", Proceedings of the 1985 International Conference on Parallel Processing, August 1985, pp. 666-673.

6. Utku, S., Salama, M., and Melosh, R., "Concurrent Cholesky Factorization of Positive Definite Banded Hermitian Matrices", to appear in the Int'l Journal for Numerical Methods in Engineering.

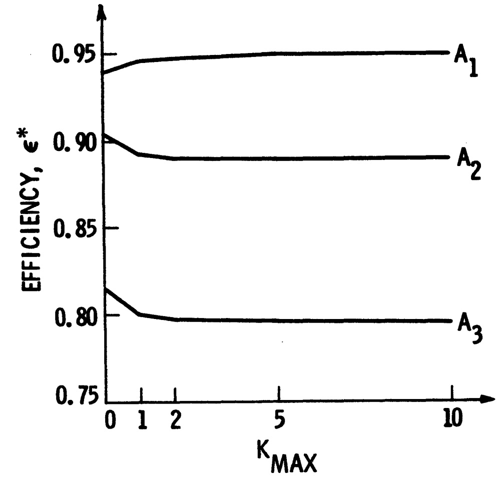


FIG 1. EFFICIENCY DEPENDENCE ON $K_{MAX}$

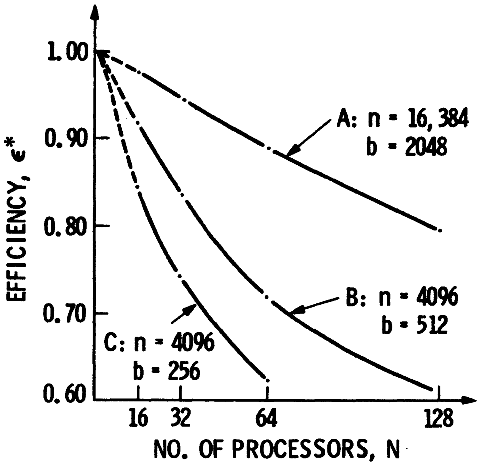


FIG 2. EFFICIENCY FOR SELECTED PROBLEM PARAMETERS IN TABLE (3)

### APPENDIX A

Five basic operations are defined in the following table:

| Oper # | Description | Example: $m=8$, $r^2=m/2$, $x=1$ |
|---|---|---|
| 1 | shift j indices to the right<br>$P_{\lambda,\nu} \Rightarrow P_{\lambda,\nu+x}$ if $\nu+x \le r$<br>$\Rightarrow P_{\lambda,\nu+x-r}$ if $\nu+x > r$ | 11: $1,\overrightarrow{2}$ 12: $3,\overrightarrow{4}$<br>21: $5,\overrightarrow{6}$ 22: $7,\overrightarrow{8}$<br>⇒<br>11: 1,4 12: 3,2<br>21: 5,8 22: 7,6 |
| 2 | shift j indices up<br>$P_{\lambda,\nu} \Rightarrow P_{\lambda-x,\nu}$ if $\lambda-x \ge 1$<br>$\Rightarrow P_{r+\lambda-x,\nu}$ if $\lambda-x < 1$ | 11 ↑ 1,2 12 ↑ 3,4<br>21 ↑ 5,6 22 ↑ 7,8<br>⇒<br>11: 1,6 12: 3,8<br>21: 5,2 22: 7,4 |
| 3 | shift j indices down<br>$P_{\lambda,\nu} \Rightarrow P_{\lambda+x,\nu}$ if $\lambda+x \le r$<br>$\Rightarrow \lambda+x-r,\nu$ if $\lambda+x > r$ | 11 ↓ 1,2 12 ↓ 3,4<br>21 ↓ 5,6 22 ↓ 7,8<br>⇒<br>11: 1,6 12: 3,8<br>21: 5,2 22: 7,4 |
| 4 | exchange j of $P_{\lambda,\nu}$ with i of $P_{\lambda,\nu+x}$ | 11: $1,\overrightarrow{2}$ 12: $\overleftarrow{3},4$<br>21: $5,\overrightarrow{6}$ 22: $\overleftarrow{7},8$<br>⇒<br>11: 1,3 12: 2,4<br>21: 5,7 22: 6,8 |
| 5 | diagonally exchange i of $P_{\lambda,\nu}$ with j of $P_{\lambda+x,\nu}$ | 11: 1,2 12: 3,4<br>21: 5,6 22: 7,8<br>⇒<br>11: 6,2 12: 8,4<br>21: 5,1 22: 7,3 |

I. Start with a set of m/2 non-overlapping set of i,j indices in each processor $P_{\lambda,\nu}$, e.g., $m=32$, $N=m/2=r^2$, $r=4$, $\lambda=1,2,\ldots,r$, $\nu=1,2,\ldots,r$, so that $P_{11}(1,2)$, $P_{12}(3,4)$,... $P_{44}(31,32)$. This is the first of the (m-1) combinations.

II. Generate $(r^2-1)$ combinations of (i,j) as follows:
   1. Do (r-1) times
      a. apply operation #1, (r-1) times with x=1
      b. apply operation #3, once with x=1
   2. Repeat operation #1, (r-1) times with x=1

III. Generate $(r^2-1)$ combinations as follows.
   1. Apply operation #4, once with x=1.
   2. Do (r-2) times
      a. apply operation #2, (r-1) times
      b. apply operation #4 with x=(r-2), or #1 with x=(r-2) (alternate between the two).
   3. Apply operation #2, (r-1) times with x=1.
   4. Apply operation #5 with x=1,2,..., (r-2).
   5. Apply operation #2 with x=(r-2).

# MATHEMATICAL MODEL PARTITIONING AND PACKING FOR PARALLEL COMPUTER CALCULATION

Dale J. Arpasi and Edward J. Milner

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Abstract -- This paper deals with the development of multiprocessor simulations from a serial set of ordinary differential equations describing a physical system. The identification of computational parallelism within the model equations is discussed. A technique is presented for identifying this parallelism and for partitioning the equations for parallel solution on a multiprocessor. Next, an algorithm which packs the equations into a minimum number of processors is described. The results of applying the packing algorithm to a turboshaft engine model are presented.

## Introduction

Multiple processors, operating together to solve a single problem, can, in many cases, decrease the time of calculation. This is important in time-critical applications, such as real-time simulation, where this technique can provide computational rates unachievable on a single processor or allow the use of lower cost hardware to provide the necessary computational capabilities. For certain classes of problems it is possible to configure a network of microcomputers to achieve the same throughput rate as a large mainframe computer at a lower initial and ongoing maintenance cost.

The parallel processing concept has opened new areas of research and development in hardware, software and theory. Some efforts sponsored by NASA Lewis are described in Refs. [1] to [6]. Techniques for developing mathematical models that can be solved efficiently on parallel processors is a key area of study. The first step in developing these multiprocessor models is to identify parallelism within the mathematical formulation of the problem. This requires a data flow analysis of the problem's equations and will identify the "critical path" and the minimum achievable calculation time. The next step is to arrange, or "pack" the noncritical path computations on the minimum number of processors so as to make maximum use of the available computing resources.

This paper presents a method of partitioning and packing equations for multiprocessor solution. Reference [6] gives a more detailed discussion of these techniques, including a comprehensive example of applying them to a turboshaft engine model.

## Computational Parallelism

A mathematical model of a physical system consists of a set of equations which describe, to some degree of accuracy, the response of that system to external influences (driving functions) over a limited range of operation. This range is defined in terms of the maximum and minimum values of the driving functions and, if time dependent, the maximum frequency or maximum rate of change of these functions. Generally, the object of this modeling effort is to provide a simulation of the physical system.

The prerequisite to developing parallel processor simulations is to be able to identify the parallel computational paths contained in the model. In general, a dynamic model can be programmed on a digital computer as a set of N computationally sequential equations of the form

$$X_K(ih) = f_K[X_m(ih),\ X_m((i - 1)h),\ \ldots,\ u(ih)]$$

where $X_K(ih)$ is the result of the $K^{th}$ equation at time ih. Here, h denotes the simulation time step or update interval of the model calculations. The arguments $X_m(ih)$ are the current values of the results of preceding equations in the model (i.e., m = 1 to K - 1), and $X_m((i - 1)h)$ are the past values of the results of all equations in the model. The argument u represents values obtained from sources external to the model which are always available at the start of the model computation sequence. The functional relationship between $X_K$ and its arguments is represented by $f_K$. Assuming an equation is an indivisible computational unit, then the parallelism in the model is determined by the arguments of each equation. That is, two equations, or sets of equations, can be computed in parallel within an update interval if their arguments are independent of the results of the others computed in that interval.

For example, a model of the form

$$X_1(ih) = f_1[X3((i - 1)h)]$$

$$X_2(ih) = f_2[X_1(ih)]$$

$$X_3(ih) = f_3[X_2(ih),\ X_3((i - 1)h)]$$

contains no parallelism since $X_3(ih)$ requires $X_2(ih)$ and $X_2(ih)$ requires $X_1(ih)$. These calculations must be done serially. However, the model

$$X_1(ih) = f_1[X_3((i - 1)h)]$$

$$X_2(ih) = f_2[u(ih)]$$

$$X_3(ih) = f_3[X_1(ih),\ X_2(ih),\ X_3((i - 1)h)]$$

does contain parallelism since $X_1$ can be computed at the same time as $X_2$.

Parallelism due to decoupled or loosely coupled equation sets is easily identified from the physical nature of the model. A more difficult task is the identification of parallelism in a set of closely coupled equations, where the process dynamics dictate the use of current arguments in solving for equation results. For instance, suppose a model contains the following set of equations:

$$X_1(ih) = f_1[X_5((i - 1)h), u(ih)]$$

$$X_2(ih) = f_2[X_5((i - 1)h)]$$

$$X_3(ih) = f_3[X_2(ih)]$$

$$X_4(ih) = f_4[X_1(ih), X_2(ih)]$$

$$X_5(ih) = f_5[X_3(ih), X_4(ih), X_5((i - 1)h)]$$

The variable $X_1$ can be computed at the start of the calculation interval, since it is a function of the past value of $X_5$ and the external variable u. $X_2$ may also be computed at the start of the interval. However, the calculation of $X_3$ must be delayed until $X_2$ has been determined, the calculation of $X_4$ must be delayed until both $X_1$ and $X_2$ are determined, and the calculation of $X_5$ must be delayed until both $X_3$ and $X_4$ have been determined. As shown in Fig. 1(a), three computational paths can be identified which can be assigned to three different computers in the simulation. Note that "wait states" have been inserted to insure the currency of the arguments. That is, equation calculation is delayed until current argument values become available. The $X_3$ calculation is shown to be delayed slightly for the transfer of $X_2$. The shaded areas (time slots) in Fig. 1(b) indicate the time available for result transfer to computer number 1. The calculation of $X_1$ and $X_3$ can take place anywhere in the time slot.

The detection of this type of computational parallelism can become burdensome when the equation set becomes large. The technique, however, can be automated. Related to this problem of partitioning is the problem of allocation (i.e., packing these paths into a minimum number of computers without extending the update time). Figure 1(b) demonstrates packing of the paths defined in Fig. 1(a). Arbitrary calculation times of $TX_1$, $TX_2$, $TX_3$, $TX_4$, and $TX_5$ have been assigned to the equations producing results $X_1$ through $X_5$, respectively. The time $TX_1$ includes the time required to obtain the value of u. Note in Fig. 1(b), that, because of the calculation times, the $X_2$ - $X_4$ - $X_5$ path is critical in that it contains no idle states. This path, therefore, dictates the minimum possible update time ($\Delta T = TX_2 + TX_4 + TX_5$). The paths $X_1$ and $X_3$ are assigned to separate computers. Packing in this example is a trivial task, since the $X_3$ calculation can be moved onto computer number 2 to be calculated during the idle period, as shown in the figure.

In many cases, efficient packing requires shifting equations in their time slots. This causes a ripple effect on the time slots of other equations which can complicate the packing problem. Because of the nature of the packing problem, a unique solution to the development of a packing algorithm does not exist. There are many ways to pack most parallel models.

In the following sections partitioning and packing algorithms that have been developed at NASA Lewis are discussed. These algorithms were tested with a model of a turboshaft engine and detailed results are presented in Ref. [6].

## Partitioning

To begin the discussion of the partitioning algorithm, certain terms should be defined. A mathematical model is a set of equations, written to define the characteristics of a physical system to some desired degree of accuracy. A program is a sequential set of digital equations and supporting information (e.g., variable and constant definition) which define the mathematical model within the constructs of a programming language. A path is a subset of these equations which, because of interrelationships between arguments and results, contains no parallelism. Partitioning is the transformation of the program equations into a number of paths which may be calculated in parallel. Packing is the combination of paths into a minimum number of processors (computers) which provide computation of the model within a prescribed update interval. The critical path is the longest path and the prescribed update interval must be greater than or equal to the calculation time of the critical path.

In this discussion of partitioning it is assumed that a program is given. That is, these equations, when executed serially, provide the required results. No assumptions are made concerning the parallelism of computational operations contained in the program equations.

$$x = a*y + b*z$$

contains parallelism (i.e., a*y can be calculated in parallel with b*z) which will be ignored since we are concerned with partitioning at the equation level and not parsing. For purposes of this discussion, the above equation will be considered as

$$x = f(a, y, b, z)$$

where f is some single operation. Therefore, equations will be assigned to paths in their entirely and not broken up into more primitive result-argument relationships.

As indicated in the last section, partitioning requires the establishment of result-argument relationships for the serial set of equations in order to develop computational paths. It is also necessary to know the calculation time of each equation. The program must be processed to provide this information. For this effort, the result-argument relationships and the calculation time information are outputs of the multiprocessor programming utility RTMPL [2], [3]. The primary function of this utility is to translate a structured program of the mathematical model into assembly language for the simulation processor(s). As an option, the utility also provides information on the result and arguments of each equation, the processor operations necessary to obtain the result, and the processor calculation time for each operation. For the equation

$$X = y + z$$

The processor operations to compute the equation are: load register R1 with z (requires 8 time units), add variable y to R1 (16 time units) and store R1 as the value of variable X (8 time units). This type of information is generated for each equation in the program.

Consider the close-coupled example in the previous section. The first step in the partitioning process is to convert utility generated information into the form needed for partitioning. Dependent arguments are those which are the results of previous program equations calculations in the update interval (e.g., $X_1$ is a dependent argument of $X_4$). These are the drivers for partitioning since their current values are required before the computation sequence can continue. The independent arguments u and $X_5$ do not affect partitioning since only past values are used. The calculation time for each equation is determined by adding the calculation times of the given processor operations.

The time at which an equation can start is determined by the arguments and calculation time of each equation. The first equation of a set only has independent arguments and thus, can always start at time 0 (measured from the beginning of the calculation update interval). It can never require results from calculations in the current update interval since none are yet available. An equation can end at the time obtained by adding its calculation time to the time it can start. The general formula for obtaining this time is

$$\text{CANSTART(RESULT)} = \text{MAX(CANEND(ARG 1), CANEND(ARG 2), ..., 0)}$$

$$\text{CANEND(RESULT)} = \text{CANSTART(RESULT)} + \text{CALCTIME(RESULT)}$$

where ARG1 is the first dependent argument, etc. This formula is applied sequentially to each equation in the program.

Once these attributes have been established for each program equation, the identification of computational paths contained in the program can begin. The algorithm used for path identification is shown in Fig. 2. Its purpose is to identify all sequences of equations which contain no parallelism and which must be computed serially. These paths are organized into a linked list called PATHLIST. The paths in PATHLIST are ordered in terms of decreasing path calculation time. Therefore, the first path in PATHLIST is the critical path. To form a path, the algorithm selects the program equation, having the maximum CANEND time, and which has not already been assigned to a path. This is the result equation of the path. The next equation selected is the one which produces a result used as a dependent argument of the result equation. If more than one equation result is used as a dependent argument, then the one with maximum CANEND time is selected. The selected equation then is inserted in front of the result equation in the path. The path formation process continues until an equation is inserted which has no dependent argument equations which are not already assigned to a path. Paths are formed until all program equations have been assigned.

Partitioning has been discussed in terms of equations that produce values of variables. Often, mathematical models contain statements that do not produce values. Two common examples are conditional statements (e.g., IF ... THEN ... ELSE) and command statements (e.g., I/O operations). The calculation time of such a statement must be combined with a preceding or following equation. This could impose limitations on program structure and is a subject for future study.

## Packing

The partitioning process produces a number of paths consisting of equations which must be computed serially and a table of information on each equation. The final task in the process of formulating a multiprocessor model is to pack the paths for assignment to a minimum number of processors. The first path in our list has the largest calculation time due to the partitioning algorithm. This is called the critical path and its calculation time is the minimum time in which the model can be computed no matter how many processors are used. The number of paths identified through partitioning is usually greater than the minimum number of processors necessary.

The minimum number of processors necessary to implement a multiprocessor simulation depends on how fast the simulation must be computed. This update time must be specified prior to packing. The simulation time step h is usually based on stability and dynamic accuracy requirements. For real-time applications, the update time $\Delta T$ must be equal to h. The update time also specifies when the computations must end. The first step in packing is to determine when each equation must end using the specified update time.

To determine when an equation must end, we begin with the state variables (defined here as those variables whose current values are not used as arguments in the model, but appear as results of model equations). The state variable computations will be the last computations performed, and thus must end at the prescribed update intervals. The calculation of equations which are dependent arguments of these variables must end no later than the time at which the state variable calculations must start. The times when subsequent equations in the result/argument string must end are similarly determined. Since a variable can be used as a dependent argument in more than one equation, care must be taken that the earliest time, arrived at after all paths are analyzed, is used to specify when that equation must end.

We now have determined when an equation can start, can end, and must end. These are termed equations attributes. Since the paths are serial they can also be assigned these attributes: A path can start when its first equation can start, a path can end when its last equation can end, a path must end when its last equation must end, and additionally, the calculation time of a path is the summation of the calculation times of its equations. This is sufficient information to pack the paths.

The solution to the packing problem is not unique in that many arrangements of paths in processors can result in a satisfactory solution.

The packing algorithm, shown in Fig. 3, was designed to achieve the minimum number of processors. Other requirements which may be imposed, such as memory size limitations and interprocessor data transfer limitations were not imposed on the algorithm.

As input, the algorithm requires: (1) that all paths be specified in a linked list called PATHLIST in order of decreasing calculation time; (2) that the required update time of the simulation, ΔT, is specified and that the attributes of each equation and path (CANSTART, CANEND, MUSTEND, CALCTIME) have been determined as described above.

The packing algorithm creates processors as needed and inserts paths from PATHLIST according to a hierarchy of relationships between existing equations in a processor and the equations in the unpacked paths. When a processor is created, the path with the longest calculation time in PATHLIST is inserted. Next, the paths which are related to paths already in a processor are tested to see if they fit (see discussion of TESTFIT algorithm below). If so, they are inserted, if not, they are placed in a carry-over list.

Then, paths in pathlist which are unrelated to the equations in the processor are tested. If one of these is inserted, unrelational testing is ended and relational testing begins again. When no other paths can be inserted into a processor, another processor is created. This process continues until all paths in PATHLIST are inserted into a processor.

Relational testing is prioritized. All unpacked paths which provide critical arguments are tested first. (A path is considered to provide a critical argument if the result of the last equation in the path (EL) is an argument of a processor equation (EP) and

MUSTEND (EL) = MUSTEND (EP) - CALCTIME (EP)

Next, other related paths are tested. Then paths in the carry over list (which was formed from paths which were related to equations packed into previously formed processors, but not yet packed) are tested.

Paths are tested for insertion on an equation by equation basis using the test fit algorithm shown in Fig. 4. First the attributes (CANSTART, CANEND, MUSTEND, AND, CALCTIME) of all program equations are saved. This is necessary because inserting an equation into a processor can cause a ripple effect on the attributes of other equations. If the whole path does not fit, any equation of the path, inserted into the processor, must be removed and the attributes of affected equations restored.

The ripple effect is illustrated in Fig. 5. Assume a processor contains two equations (A and B) and that it has been determined that equation (C) can be inserted between them. The calculation time of each equation is shown as the shaded areas. For packing purposes, the calculation of each equation can take place anytime between its CANSTART time and its MUSTEND time.

The space available for C is the difference between the time at which B must end and A can end minus the calculation time of B. Equation C will be inserted to start directly after A can end. The calculation of B will be delayed until C can end. Note that the difference between the MUSTEND and CANEND times of A and C have been reduced to zero by the positioning of C, and that the time difference for B has been reduced. The primary impact of these changes is to reduce the space in the processor available for packing other paths. There is also a secondary impact of equal importance. By increasing the times at which C and B can end, any unpacked equations which use these equations as arguments have their starting times delayed. Similarly, by reducing the MUSTEND times of A and C, the MUSTEND times of any unpacked arguments of these equations are moved up. These effects tend to reduce the slot sizes of unpacked equations restricting the range of time into which they can be packed into a processor. Also, these ripple effects may introduce computational gaps within unpacked paths.

After the attributes are saved (Fig. 4), the equations within a path are ordered in terms of decreasing CANEND times for insertion testing. That is the latest equation will be tested first and the earliest last.

The processor equations are arranged in sequential order where EP(1) is the earliest equation and EP(n) is the latest equation. Testing to determine if a path equation (E(i)) can be inserted into the processor involves the identification of all slots between any two processor equations (EP(j - 1), EP(j)) where the equation fits. The processor end points (i.e., EP(j) = EP(1) and EP(j - 1) = EP(n)) must also be considered. Because of the argument and result relationships between E(i) and the processor equations it is required that the range of processor equations be limited for testing purposes. Let the end points of the range be designated by EPE and EPL (the earliest and latest processor equations respectively, before which E(i) may be inserted). This range is established as follows: If E(i) is an argument of a processor equation, then EPL is the earliest processor equation of which E(i) is an argument (EPE = EP(1)); if any processor equation is an argument of E(i), then EPE is the one following the latest of these and EPL is the last processor equation plus one (end point); if E(i) is unrelated to any processor equation, then EPE = EP(1) and EPL is the last processor equation plus one.

Once the range of testing has been established, all slots within that range are tested to determine if E(i) fits. The fit criterion is as follows:

1. If EP(j) = EP(1) then CANEND* (E(i)) = CALCTIME (E(i)) else CANEND* (E(i)) = CANEND (EP(j - 1)) + CALCTIME (E(i));

2. If CANEND* (E(i)) < CANEND (E(i)) then CANEND* (E(i)) = CANEND (E(i));

3. If EP(j - 1) = EP(n) then MUSTEND* (E(i)) = ΔT else MUSTEND* (E(i)) = MUSTEND (EP(j)) - CALCTIME (EP(j));

4. If MUSTEND* (E(i)) > MUSTEND (E(i)) then MUSTEND* (E(i)) = MUSTEND (E(i));

5. If [[CANEND* (E(i)) < = MUSTEND (E(i))] and [CANEND* (E(i)) < = MUSTEND* (E(i))]], then E(i) fits.

The asterisk indicates the attributes of E(i) if it were inserted into the slot between EP(j - 1) and EP(j).

If it is established that E(i) can fit in more than one slot, the testfit algorithm proceeds to select the slot into which E(i) best fits. The best fit criterion is as follows:

If a slot exists such that

$$\text{CANEND* (E(i)) - CALCTIME (E(i)) - CANEND (EP(j - 1)) = 0}$$

then this slot is selected. Otherwise, the latest slot which maximizes

$$\text{[MUSTEND* (E(i)) - CANEND* (E(i))]}.$$

is selected.

This criterion provides for efficient packing by eliminating processor idle time if possible, and if not, then the ripple effect from insertion is minimized.

Once a slot is selected, the equation is inserted and the attributes of all program equations are updated to reflect the insertion. If any path equation cannot be inserted into the processor, path equations which have already been inserted are removed from the processor, the original attributes are restored to the program equations and the Test Fit algorithm ends.

## Results

The packing algorithm was programmed in Pascal, along with the partitioning algorithm. It was then tested on a turboshaft engine model. The results, in terms of percent processor utilization, are presented in Table I. The first column is the update time ΔT specified prior to packing. It is given in terms of machine cycles. In this case, four processors were required. The second column gives the number of processors required for packing. The remaining columns show the percent utilization of the update time in calculating each processor's assigned equations. The first specified update time (5666 cycles) was the minimum possible time and corresponds to the critical path calculation time. The second update time selection (10 000 cycles) required two processors. Note that the percent utilization of the last processor in each case exceeds the summation of time available on the other processor(s). The algorithm, therefore, functioned satisfactorily.

Since the packing algorithm does not account for data transfer times, it is possible that the available time between when a variable is computed on one processor and when its value is required for computation on another processor will be less than the time required to transfer the variable between the processors. The effect of this will be to increase the effective calculation time of the packed simulation, and therefore, to increase the minimum achievable update time. Data transfer effects may be significant for multiprocessor systems with inefficient data transfer mechanisms or for simulations that require large volumes of data transfer between processors. Future work in the development of a packing algorithm should include a study of the effects of data transfer. While these effects will increase the critical path time, and therefore, the minimum update time, proper consideration of data transfer will minimize this increase and provide far more efficient packing.

## Concluding Remarks

The algorithms and considerations presented for partitioning and packing mathematical models for calculation on parallel processors have simplified the development of multiprocessor simulations at Lewis. Evaluation of the packing algorithm will continue as other multiprocessor simulations are developed. Work on a completely automated programming package is in progress, which will take a structured serial statement of a mathematical model, detect this parallelism, and provide load modules for the required number of processors.

The authors welcome discussions of the techniques presented in the paper, related techniques, and developments in the many other aspects of multiprocessor simulation.

## References

[1] Blech, R.A.; and Arpasi, D.J.: Hardware for a Real-Time Multiprocessor Simulator. NASA TM-83805, 1984.

[2] Arpasi, D.J.: RTMPL - A Structured Programming and Documentation Utility for Real-Time Multiprocessor Simulations. NASA TM-83606, 1985.

[3] Arpasi, Dale J.: Real-Time Multiprocessor Programming Language, (RTMPL) - Users Manual. NASA TP-2422, 1985.

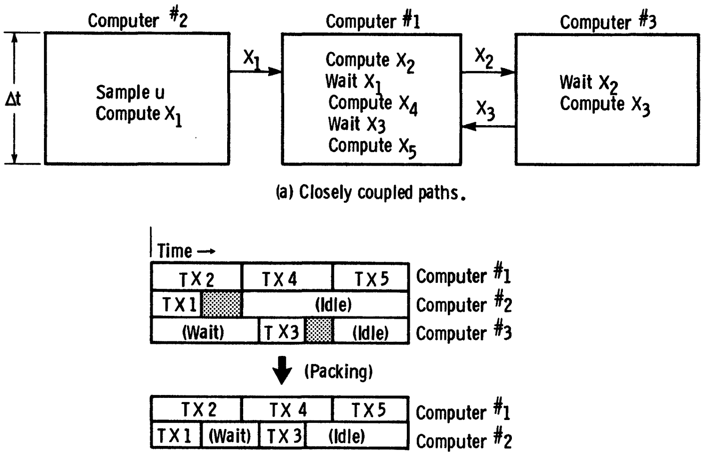
[4] Cole, G.L.: Operating System for a Real-Time Multiprocessor Propulsion System Simulator. NASA TM-83605, 1985.

[5] Milner, E.J.; and Arpasi, D.J.: Simulating a Small Turboshaft Engine in a Real-Time Multiprocessor Simulator (RTMPS) Environment. NASA TM-87216, 1986.

[6] Aprasi, D.J.; and Milner, E.J.: Partitioning and Packing Mathematical Simulation Models for Calculation on Parallel Computers. NASA TM-87170, 1986.

TABLE I. - PACKING ALGORITHM RESULTS FOR TURBOSHAFT ENGINE MODEL

| Update time | Processors required | Processor percent utilization | | | |
|---|---|---|---|---|---|
| | | P number 1 | P number 2 | P number 3 | P number 4 |
| 5 666 | 4 | 100 | 98 | 97 | 53 |
| 10 000 | 2 | 99 | 98 | -- | -- |
| 19 568 | 1 | 100 | -- | -- | -- |



(a) Closely coupled paths.

(b) Packing.

Figure 1. - Partitioning and packing closely coupled equations.

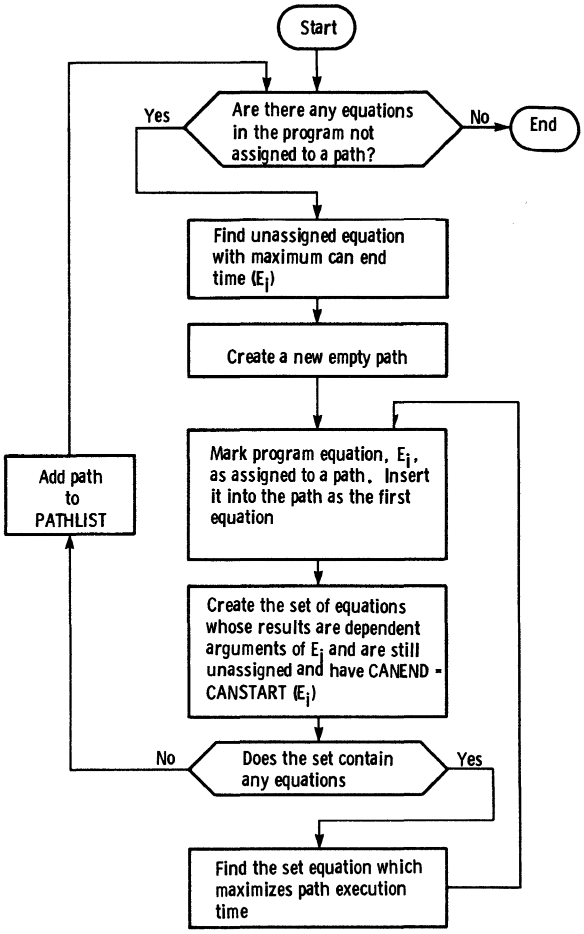
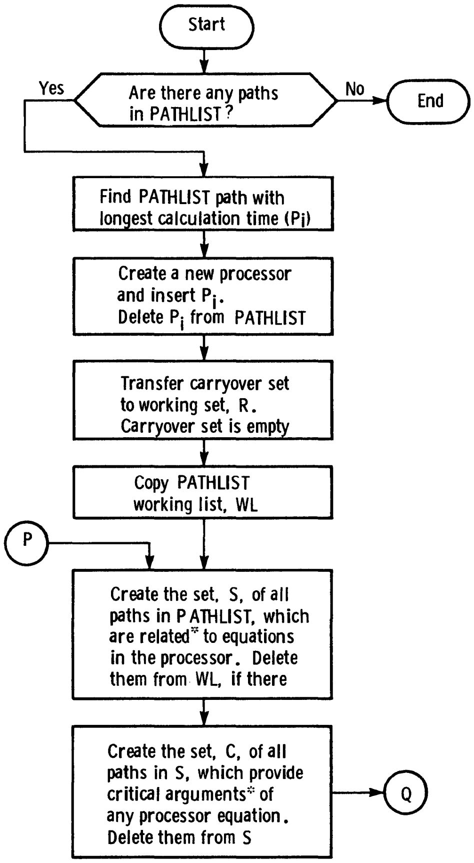

Start
Are there any equations in the program not assigned to a path?
Yes
No
End
Find unassigned equation with maximum can end time (Ei)
Create a new empty path
Mark program equation, Ei, as assigned to a path. Insert it into the path as the first equation
Create the set of equations whose results are dependent arguments of Ei and are still unassigned and have CANEND - CANSTART (Ei)
Does the set contain any equations
No
Yes
Find the set equation which maximizes path execution time
Add path to PATHLIST

Figure 2. - Path identification Algorithm.


Start
Yes
Are there any paths in PATHLIST?
No
End
Find PATHLIST path with longest calculation time (Pi)
Create a new processor and insert Pi. Delete Pi from PATHLIST
Transfer carryover set to working set, R. Carryover set is empty
Copy PATHLIST working list, WL
P
Create the set, S, of all paths in PATHLIST, which are related* to equations in the processor. Delete them from WL, if there
Create the set, C, of all paths in S, which provide critical arguments* of any processor equation. Delete them from S
Q

*See explanation in test.

Figure 3. - Packing Algorithm.

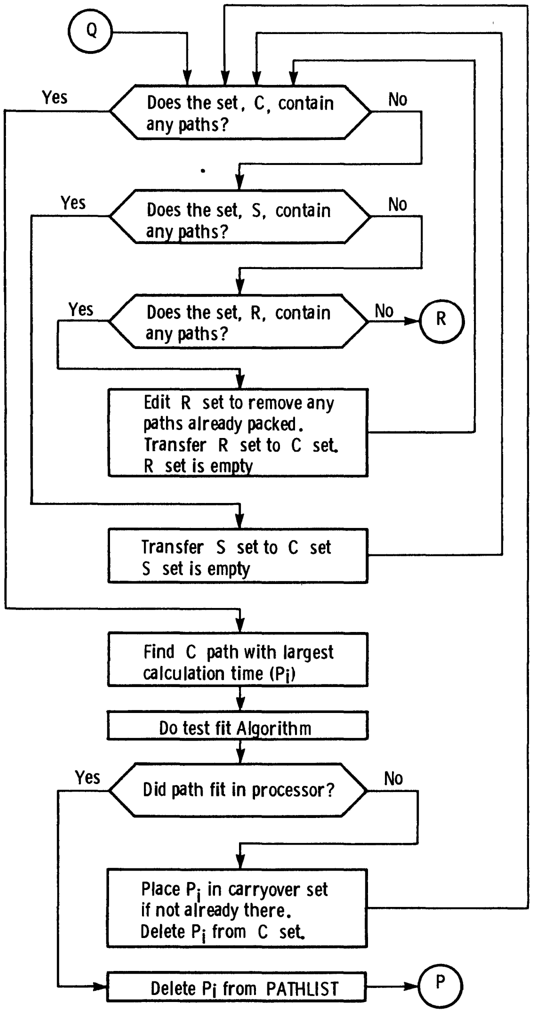

Q
Yes
Does the set, C, contain any paths?
No
Yes
Does the set, S, contain any paths?
No
Yes
Does the set, R, contain any paths?
No
R
Edit R set to remove any paths already packed. Transfer R set to C set. R set is empty
Transfer S set to C set S set is empty
Find C path with largest calculation time (Pi)
Do test fit Algorithm
Yes
Did path fit in processor?
No
Place Pi in carryover set if not already there. Delete Pi from C set.
Delete Pi from PATHLIST
P

Figure 3. - Continued.

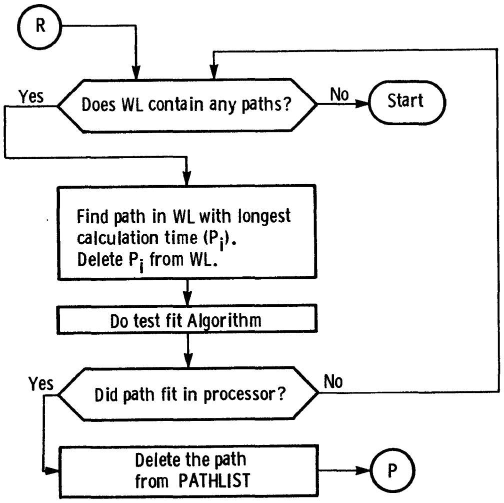
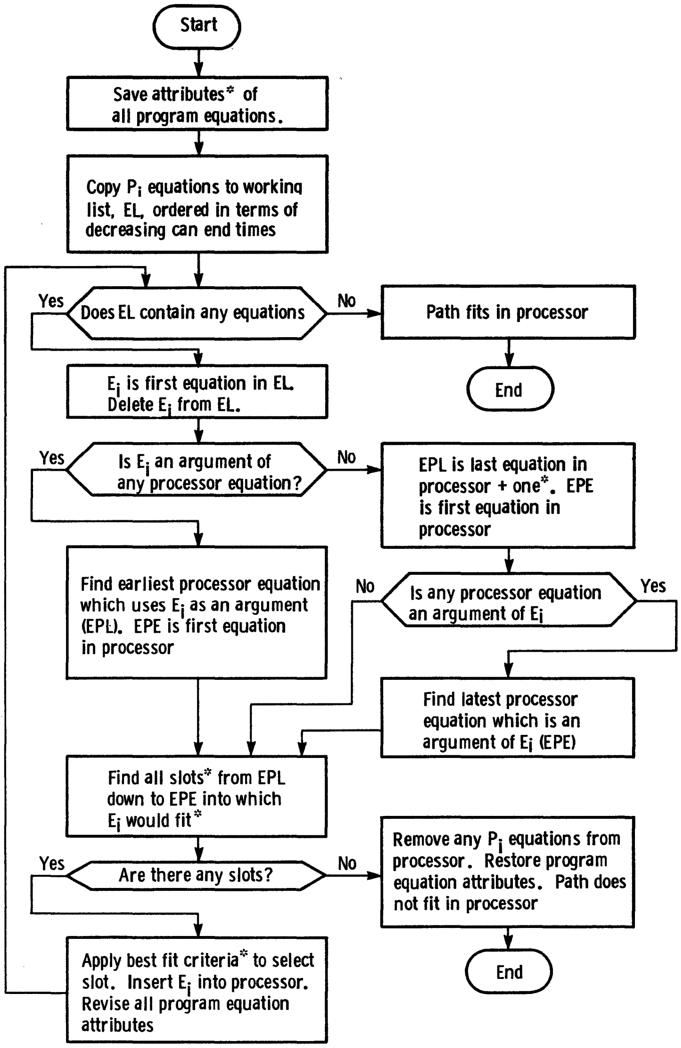

R
Yes
Does WL contain any paths?
No
Start
Find path in WL with longest calculation time (Pi). Delete Pi from WL.
Do test fit Algorithm
Yes
Did path fit in processor?
No
Delete the path from PATHLIST
P

Figure 3. - Concluded.


Start
Save attributes* of all program equations.
Copy $P_i$ equations to workinq list, EL, ordered in terms of decreasing can end times
Yes
Does EL contain any equations
No
Path fits in processor
End
$E_i$ is first equation in EL. Delete $E_i$ from EL.
Yes
Is $E_i$ an argument of any processor equation?
No
EPL is last equation in processor + one*. EPE is first equation in processor
Find earliest processor equation which uses $E_i$ as an argument (EPL). EPE is first equation in processor
No
Is any processor equation an argument of $E_i$
Yes
Find latest processor equation which is an argument of $E_i$ (EPE)
Find all slots* from EPL down to EPE into which $E_i$ would fit*
Yes
Are there any slots?
No
Remove any $P_i$ equations from processor. Restore program equation attributes. Path does not fit in processor
End
Apply best fit criteria* to select slot. Insert $E_i$ into processor. Revise all program equation attributes

* See explanation in text

Figure 4. - Test fit Algorithm.

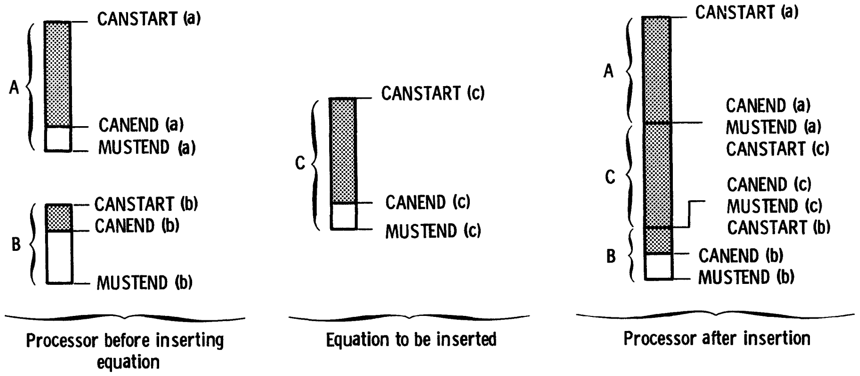


Figure 5. - The affect of insertion on equation attributes.

# A NEW GRAMMAR FOR ARITHMETIC EXPRESSIONS IN A PARALLEL PROCESSING ENVIRONMENT

Fatemeh Abdollahzadeh*
Medhi Badii#
D. J. Cooke+

*Dept. of Computer Science
and Engineering
The University of Toledo
Toledo, Ohio 43606

#Dept. of Computer Science
and Mathematics
Isfahan Univesity
Isfahan, Iran

+Dept. of Computer Studies
Loughborough University of Tech.
Leics, United Kingdom

## Abstract

The specification of arithmetic expressions by means of (i) ambiguous context free grammars and (ii) context sensitive grammars is considered. A CSG is developed which selects the balanced tree most suitable for parallel execution. The possibilities of parsing an arithmetic expression by using the grammar and constructing a replacement syntaxanalyser module for a parallel compiler are discussed.

## Introduction

In recent years much work has been done on the problem of "balancing arithmetic trees, Ashoke [4] Evans and Williams [8] and Evans and Abdollazadeh [7]. The principle behind this process being to divide an arithmetic expression into two parts of comparable computational 'size' which can then be evaluated at the same time on different processors and thus reduce the time required to evaluate the entire expression. The smaller subexpressions formed as a result of this dissection can then be treated in a similar fashion until we reach a level at which such processing is uneconomical or unnecessary.

Until now the algorithms have relied on explicit knowledge of the special (algebraic) properties of the arithmetic operators involved. Because we would like to be able to apply our tree manipulations to parts of a program other than arithmetic expressions we restrict the properties used to (i) associativity of certain like operators and (ii) precendence of multiplicative operations over additive ones. These properties not only prove to be the most useful, but also appear in numerous other constructions within high-level languages and lend themselves to specification by formal grammars.

After preliminary discussion of relevent segments of language theory and of the structure of arithmetic expressions we derive suitable grammars for two classes of arithmetic expressions. We conclude the paper with a discussion of how this work may be extended to more general situations and the problem assoicated with parsing the languages generated by these grammars.

## Some Elements of Language Theory

The grammars presented later in this paper are related to a new class of operator precedence grammars. In order to facilitate their description some terminology is required. This is fairly standard and can be found elsewhere (eg Aho and Ullman [3])) but is included here for completeness. In these definitions we assume familiarity with basic set theory.

A phrase-structure grammar (PSG) is an algebraic structure consisting of an ordered 4-tuple (N,T,P,S) where N and T are non-empty finite alphabets of non-terminal symbols and terminal symbols respectively such that $N \cap T = 0$ and $N \cup T = V$ where V is called the vocabulary of G. P is a set of productions, $P \subseteq V^+ x V^*$ and, assuming the symbol → not to be in V, $(\alpha, \beta) \epsilon P$ is usually written as

$$\alpha \rightarrow \beta$$

Finally $S \epsilon N$ and S is called the start symbol.

Given $w,y \epsilon V^*$, then w directly derives y if $w=z_1uz_2$ and $y=z_1vz_2$ where $z_1,z_2,v \epsilon V^*$, $u \epsilon V^+$ and $u \rightarrow v \epsilon P$ and is written as $w \Rightarrow y$.

If now w and y are words over V and there is a finite sequence $w_0,w_1,w_2, \ldots w_r$ where $w_0=w$, $w_r=y$ and $w_{i-1} \Rightarrow w_i$ $(i=1,\ldots,r)$ then we say that w derives y, written as $w \overset{*}{\Rightarrow} y$. Moreover if $\alpha \epsilon T^*$ and $S \overset{*}{\Rightarrow} \alpha$ then $\alpha$ is a sentence generated by G. Thus the language, L(G) generated by G is $\{x : x \epsilon T^* \alpha \nu \phi\ S \overset{*}{\Rightarrow} x\}$. Where G is understood we can define $L(X) = \{x : x \epsilon T^*, X \epsilon N \text{ and } X \overset{*}{\Rightarrow} X\}$.

Let G=(N,T,P,S) be a PSG as described above. such a grammar is called Chomsky Type 0. If each element of P is of the form $z_1uz_2 \rightarrow z_1vz_2$ where $z_1,z_2\ \epsilon V^*$, $u \epsilon N$ and $v \epsilon V^+$ then G is said to be context sensitive (a CSG), or Chomsky Type 1. An alternative restriction is that if $w \rightarrow y\ \epsilon P$ then w and y should be such that $1 \leq |w| \leq |y|$.

If the replacements may be carried out regardless of context then we may replace 'contexts' $z_1$ and $z_2$ by the empty string, $\Lambda$, and obtain the weaker restriction that if $w \rightarrow y \epsilon p$ then $w \epsilon N$ and $y \epsilon V+$. This restriction is satisfied by context-free or Chomsky Type 2 grammars, (CFG'S).

An important subset of CFG's is the so called operator grammars. These are grammars in which all productions are such that no two non-terminals are adjacent in any right-hand side of a production. For the definition of precedence relations and operator precedence grammars, see [3]. The symbols <•, = and •> denote precedence relations and provided that at most one such relation holds between any two elements of $T \cup \{\$\}$ then the associated operator grammar is called an operator precedence grammar.

0190-3918/86/0000/0075 $01.00 © 1986 IEEE

## Structure of Arithmetic Expressions

For any expression we need to consider all admissible tree structures and from these select one most suitable for parallel execution. We say a tree is admissible if, by evaluating the subexpressions in the order defined by corresponding subtreees, evaluation of the expression associated with the entire tree is arithmetically correct for all legal assignments of numerical values within the expression.

Example: For the expressions 7 - 3 + 2 the tree in Figure 1 (a) corresponds to the valuation (7 - 3) + 2 = 6 and is admissible, but the tree in 1 (b) infers the calculation 7 - 3(+2) = 2 and is inadmissible.

Eventually it will be necessary to specify the processing of arbitrary well-formed arithmetic expressions consisting of identifiers a, b, c, ... etc., each associated with a numeric quantity, the operators +,-,* and /, and the usual parentheses.

However, mindful of the desire to be able to extend this methodology to manipulation of other aspects of programming languages we shall concentrate on expressions most amenable to the process being developed here. Identification of the relevent algebraic properties is readily achieved by examination of the rules of elementary arithmetic.

Our concern is with expressions involving quantities of type real. Mathematically, the reals are an infinite set of numbers on which the two operations of addition (+) and multiplication (*) are defined. Computationally, any implementation of real arithmetic uses only a (comparatively, very small) finite set of numbers and often encounters situations where the result of an addition or multiplication is simply not defined; resulting in overflow conditions[a] and hopefully a halt in the proceedings.

Using parentheses to indicate the desired order of evaluation, the field of real numbers R is subject to the following 9 axioms:

1. Addition is commutative
   a+b = b+a for all a,b $\epsilon$R
2. Addition is associative
   (a+b)+c = a+(b+c) for all a,b,c $\epsilon$R
3. There is a specific x$\epsilon$R such that
   x+a = a for all a$\epsilon$R
   x is called zero and usually written 0.
4. For each a$\epsilon$R there corresponds an element b$\epsilon$R such that
   a+b = 0
   b is the additive inverse of a, usually written -a or (-a).
5. Multiplication is commutative
   a*b = b*a for all a,b$\epsilon$R
6. Multiplication is associative
   (a*b)*c = a*(b*c) for all a,b,c$\epsilon$R
7. There is a specific y$\epsilon$R\{0} such that
   y*a = a for all a$\epsilon$R
   y is the multiplicative identity and is usually denoted by 1.
8. For each a$\epsilon$R\{0} there is an element c$\epsilon$R such that a*c = 1
9. Multiplication distributes over addition
   a*(b+c) = (a*b)+(a*c) for all a,b,c$\epsilon$R.

Subject to the calculations being carried out within the limits determined by an implementation, the arithmetic is fully and strictly as defined by these nine rules. However, in order to make the expressions more easily understood various conventions are adopted. First, subtraction and division are introduced and defined in terms of additive and multiplicative inverses viz.

$$a-b \overset{def}{=} a+(-b) \qquad a/b \overset{def}{=} a*(b^{-1})$$

Second, in order to reduce the necessity for some parentheses, it is assumed that multiplication (and hence division) take precedence over addition (and subtraction)
ie a*b+c means (a*b)+c and a+b*c means a+(b*c)

Following the concepts of admissibility and generalization we choose not to allow tree transformations based on commutativity or distributivity. This is because these properties are very special and hence do not easily generalize. They are also of questionable benefit in reducing execution time of expressions, (see Abdollahzadeh [1] for details). In terms of tree transformations the manipulations are as in Figures 2 and 3 respectively; commutativity requires reordering of leaves in the trees and distributivity involves deletion/creation of terminal and non-terminal nodes.

Axioms 3,4,7 and 8, involving identities and inverses, serve to define the auxiliary operations of subtraction and division. If these operations are not implemented directly but via the process of constructing inverses then they need not be considered further. Notwithstanding this fact most computers have explicit subtraction (and, getting away from most micro-processors, division) instructions which are more awkward to manipulate. The subtraction operation being subject to the derived rules as follows. By definition

$$x-y-z = x+(-y)+(-z)$$

This second expression can then be evaluated either as

$$(x+(-y))+(-z) = (x-y)-z$$

or, as

$$x+((-y)+(-z)) = x+((-1)*y+(-1)*z)$$
$$= x+(-1)*(y+z) = x-(y+z)$$

It therefore follows that both the trees in Figure 4 are admissible for this expression and to be consistent with our earlier remarks on tree tramsformations we reject the form in Figure 4 (b). This emphasises the left associativity of subtraction. Similar arguments apply to division.

[a] This is in contrast to calculations suffering loss of accuracy due to round-off error. When specific orders of evaluation are required in order to minimize this effect, parentheses can be used as noted below.

This leaves us with the associativity of addition and multiplication, and the precedence of multiplication over addition.

We now turn our attention to the grammatical specification of admissible trees for arithmetic expressions that allow associative transformations and assist in the selection of the admissible tree most suitable for parallel execution.

### Expressions Involving a Single Associative Operator

The classical grammar central to our considerations is:

$E \to E+T \mid T$ , $T \to T*P \mid P$ , $P \to (E) \mid I$ (Grammar $G_1$)

$G_1$ involves two operators and hence we shall initially only use one layer of it, namely the sub-grammer $G_2$.

$E \to E+T \mid T$ (Grammar $G_2$)

The sentences derived from E in $G_2$ are of the form
T, T+T, T+T+T, ... etc.

Moreover a typical derivation in $G_2$ is

$$E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow E+T+T+T \Rightarrow E+T+T+T+T \Rightarrow T+T+T+T$$

with the derivation tree in Figure 5 (a).

The corresponding arithmetic tree is given in Figure 5 (b).

Symbolically (ie syntactically rather than arithmetically). $L(G_2)=\{T(+T)^n : n \geq 0\}$. Now this language can also be written as $\{(T+)^n T : n \geq 0\}$ and this gives rise to a syntactically equivalent grammar $G_3$.

$E \to T+E \mid T$ (Grammar $G_3$)

However the underlying syntactic structure, and therefore any 'natural' semantic ordering, associated with $G_3$ is as shown in Figure 6. Figure 6 depicts the $G_3$ derivation of the same sentence as in Figure 5.

In grammar $G_2$ the left recursion [3] indicates the left associativity of addition and gives rise to the precedence relation + •> +. Similarly the right recursion in $G_3$ yields right associativity and + <• +. Notice also that both $G_2$ and $G_3$ are unambiguous and hence there is only one admissible semantic structure for any sentence in each of the grammars. However, since addition is associative (rather than just left- or right-associative), there are, in non-trivial examples, very many admissible trees. To enable syntactic generation of these trees we combine $G_2$ and $G_3$ into $G_4$

$E \to E+T \mid T+E \mid T$ (Grammar $G_4$)

Now $G_4$ is ambiguous and gives rise to, amongst others, the five semantic trees depicted,in Figure 7.

$G_4$ incorporates the precedence relations + <• + and + •> + and leads to the definition of an extended precedence relation, namely $^{\bullet>}$ . Formally: within an operator grammar, given terminal (operator) symbols a and b, we say that a and b are of comparable precedence (written a $^{\bullet>}$ b) iff a <• b and a •> b. Thus, from $G_4$ we may obtain the precedence table in Figure 8.

In fact $G_4$ can generate all admissible trees for sentences in $L(G_2)$ but by the deterministic nature of context-free parsing algorithms, even when acting on ambiguous grammars, such algorithms will always give a tree similar to those in Figure 5 (b) or 6 (b). It is argued elsewhere, Evan and Adbollazadeh [7] that the most suitable tree for parallel execution of the expression T+T+T+T+T is that given in Figure 7 (b) and we now set about the problem of isolating this particular tree from all admissible trees for the expression.

Limiting consideration to expressions built from the addition of terms (T) we can always overcome the selection problem by the inclusion of syntactic sugar in the form of parentheses. Of course, this is cheating, expecting the programmer to analyse the expression for himself, however it helps to motivate the grammatic constructions that will eventually lead to automatic balancing.

Explicitly, using parentheses to indicate the tree structure, the expressions $E_n$ involving n terms ($T_i$) and n-1 operators are:

$$E_1 = T_1,\quad E_2 = (T_1+T_2),\quad E_3 = (T_1+T_2)+T_3,$$
$$E_4 = ((T_1+T_2)+(T_3+T_4))$$
$$\ldots\ E_n = ((\ldots(T_1+T_2)+(T_3+T_4))+\ldots\ldots+T_n)\ldots)$$

Obviously, when the value of n is not a power of 2, we cannot represent the structure of the expression $E_n$ by a tree which is balanced and in which all subtrees are balanced. In such cases we balance the subtrees from left to right, (the rationale for this is given in Evans and Abdollazadeh [7]). To particularize our use of the word 'balanced' we define its use in two technical terms. Both of these are recursive and they recurse with respect to the height of a tree.

The height, h(t) of the binary tree t is such that,

(i) $h(t) = 0$ if t is a tree associated with an expression involving no operators,

and (ii) $h(t) = \max(h(t_1), h(t_2))+1$ if t is composed of two subrees $t_1$ and $t_2$, joined by the binary operator op, ie $t=t_1$ op $t_2$.

A complete balanced binary tree (CBBT) of height n is (i) any tree of height n=0, or (ii) a tree corresponding to the expression $t_1$ op $t_2$ where $t_1$ and $t_2$ are CBBT's of height n-1.

It follows immediately that the CBBTs associated with the language $\{T(+T)^n : n \geq 0\}$ are those derived from expressions $E_n$ where $n=2^m$ for some m, ie trees of height m involving n operands.

A (left-to-right) balanced binary tree (BBT) is either (i) a trivial tree of height 0 or (ii) $t_1$ op $t_2$ where $t_1$ is a CBBT and $t_2$ is a BBT and $h(tl) \geq h(t_2)$.

A consequence of the definition of BBTs is that the BBT involving n operands is of height p where $2^{p-1}<n\leq 2^p$; moreover, by isolating non-trivial CBBTs in parentheses as soon as such a structure is recognized, we can deduce the form of any BBt while processing it from left to right. Using multiplicative notation to indicate duplicity of structure the process can be written explicitly as:

$E_1 = T,\ E_2 = T+T=(E_1+E_1)=2E_1,\ E_3 = 2E_1+E_1=(E_2)+E_1$

$E_4 = (E_2)+E_1+E_1=(E_2)+(E_1+E_1)=E_2+E_2)=2E_2$

$E_5 = (E_4)+E_1,\ E_6 = (E_4)+(E_1+E_1)=(E_4)+2E_1=(E_4)+(E_2)$

$E_7 = (E_4)+(E_2)+E_1\{=(E_4)+E_3\},\ E_8 = (E_4)+(E_2)+(E_1+E_1$

$=(E_4)+(E_2)+(E_2)=(E_4)+2E_2=(E_4)+(E_4)=2E_4$

$$\text{and } E_n = \{ \begin{array}{ll} 2E_{2^{p-1}} & \text{when } n=2^p \\ E_{2^{p-1}}+E_q & \text{where } 2^{p-1}<n<2^p \\ & \text{and } q<2^{p-1} \end{array}$$

Diagrammatically the balanced structure of $E_6$, $E_7$ and $E_8$ is indicated in Figure 9.

As will be seen from the right subtrees of the diagrams in Figure 9, if the tree is complete ($E_2$ in Figure 9) but of a smaller height then the left subtree ($E_4$ in Figure 9 (a), then the height of the right tree is increased by one ($E_3$ in Figure 9 (b)) and the previous subtree inserted as the left part ($E_2$ in Figure 9 (b)). Otherwise we step down the right subtree to find its largest complete component and perform the same process at a lower level. It therefore follows that a 'balancing' grammar needs to keep track of the heights (or at least the difference between heights) of adjacent subtrees.

Recall that the height of the balanced tree for $E_n$ is p where $2^{p-1}<n\leq 2^p$ and its structure is given by $E_{2p-1}+Eq$ where $q<2^{p-1}$. Hence the $(2^{p-1})$th operator is at the top of the tree, the $(2^{p-2})$th operator is one level lower, the $(2^{p-3})$th operator is two levels lower, and so on.

The grammar, $G_5$ mimics our structuring process in so far as whenever a BBT is completed it is surrounded by '(' and ')'. Also, when it is required to increase the height of a tree to accommodate more operands (as in Figure 9 (a) and (b)) this is effected by joining two trees thus:

(CPBT) + (BBT)

- even trivial trees, ie terms, are bracketed by our grammar -, grammatically we reduce ')+(' to + by a suitable production. Before giving the grammar and describing its use, we note that in order to clarify the difference between actual and terminal symbols and synthetic structure-indicators we have used 0 and C (open and close) in place of '(' and ')'. To aid parsing \$ is used as an end marker.

$$G_5 = (\{X,S,E,P,O,C\},\ \{+,T,\$\},\ Q,X)$$

where Q = {(1) E→T, (2) P→+, (3) OEC→EPE,
(4) P→CPO, (5) \$→C\$, (6) PE\$→CPE\$,
(7) S\$→OE\$, (8) S\$→OS\$, (9) X→S\$ }

Now we apply $G_5$ to the reduction and structuring of $E_7$. Recall $E_7$ = T+T+T+T+T+T+T \$ and that it should be structured (see Figure 9 (b)) as $E_7$ = ((T+T)+(T+T))+((T+T)+T)\$ in which the right subtree is a CBBT of height 2 and the left subtree is a BBT of height 2 resulting in $h(E_7)=3$. The expression can be analysed (the reverse process to derivation) as follows:

```
(1) T+T+T+T+T+T+T$       (9) OOEC P ECPE$
(2) EPE+T+T+T+T+T$      (10) OOEC P EPE$
(3) OEC+T+T+T+T+T$      (11) OOEC P OEC$
(4) OECPOEC+T+T+T$      (12) OOE P EC$
(5) OEPEC +T+T+T$       (13) OO OEC C$
(6) OOECC PEPE+T$       (14) OO OE $
(7) OOECC POEC+T$       (15)    S$
(8) OOEC P EC+T$        (16)    X
```

Productions (1) and (2) of Q (which we shall refer to as $Q_1$ and $Q_2$ ...) simply encode $T_5$ and '+' symbols and are used freely throughout the reduction. Stages 1, 2 and 3 transform 'T+T' into 'OEC' represesnting a CBBT of height 1 by means of $Q_3$. The next 'T+T' is similarly treated giving (No 4) 'OECPOEC'. Production Q4 now removes the reverse brackets ')+(' represented by 'CPO' and then using $Q_3$ again we obtain (No. 6) 'OOECC' which corresponds to a CBBT of height 2.

Now the string generated by $E_2$ in Figure 9 (b) is transformed into 'EPE' and then 'OEC' in stages 6 and 7. The central plus, now P, then causes partial combination of these CBBTs. However, the two E's in No. 8 do not combine because they are not of the same height, the second being too low.

The last term comes into play at No. 9. Because $E_7$ cannot be completely balanced we need to apply special productions to cater for the incomplete subtree $E_3$. Production $Q_6$ erases the superfluous 'C' enabling the right-hand subtree to be incorporated with the preceding E (No. 11) $Q_4$ then performs the last proper reduction with two trees of height 2 (No. 12). The final stages using $Q_5$, $Q_8$ and $Q_9$ remove peripheral bracket tokens and achieve the start symbol X.

Pictorially the parse is set out in Figure 10.

We can now extract the desired admissible tree structure from this parse by translating from Q to a related set of context-free productions Q'. This is simply done by ignoring all structuring symbols and trivial productions. Hence we have:

| Q | Q' |
|---|---|
| E → T | E → T |
| P → + | P → + |
| OEC → EPE | E → EPE |
| P → CPO | |
| \$ → C\$ | |
| PE\$ → CPE\$ | |
| S\$ → OE\$ | S → E |
| S\$ → OS\$ | |
| X → S\$ | X → S |

Applying this translation to Figure 10 gives the context-free parse in Figure 11 and further simplication trivially gives us $G_6$ defined by the productions

$$E \to E+E \mid T \qquad \text{(Grammar } G_6\text{).}$$

Consequently the sequence of applications of reductions $Q_3$ and $Q_1$ in grammar $G_5$ gives a balanced parse from the grammar $G_6$. Figure 11 is, as required, a syntactic equivalent of the semantic structure in Figure 9 (b).

Expressions Involving Two Associative Operators

The construction of the context-sensitive grammar $G_5$ from the context-free grammar $G_4$ can be mirrored for any language based on a single associative operation. Hence, we could for instance take $G_1$

$$E \to E+E \mid T \text{ , } T \to T*P \mid P, \quad P \to (E) \mid I \qquad \text{(Grammar } G_1\text{).}$$

and slice off the second, multiplicative, layer

$$T \to T*P \mid P$$

and then add extra productions to allow both left and right-associativity. Thus, to generate L(T)

we have $\quad T \to T*P \mid P*T \mid P$

or alternatively $\quad T \to T*T \mid P$ .

To deal with more than one such operator is more complicated and methods for the construction of grammars to cope with the general situation are currently under review. Of more immediate concern is the case noted above. We know that, by convention, multiplication has a higher precedence than addition and hence whenever these operators are adjacent in an expression we need to ensure that any tree balancing takes account of these precedence relations. Of course for any sub-expression involving only one kind of operator the method of section 3 can be used.

To illustrate how these operators inter-relate with each other and with the tree-balancing processes we give an example. To aid description of multiplicative sub-structure we shall use $T_n$ to represent a sub-expression involving n operands combined multiplicatively, and to aid analysis we use '\$' as a delimiter rather than just a terminator for the entire expression.

Example: Consider the expression, $I_1+I_2+I_3*I_4*I_5$ which should be parenthesised thus:

$$(I_1+I_2) + ((I_3*I_4)*I_5)$$

As before the reduction starts by forming $I_1+I_2$ into a (C)BBT of height 1. However, the '*' following $I_3$ prevents $I_3$ from being immediately included in an additive BBT of height 2, and causes a multiplicative tree to be formed from $I_3$ and $I_4$ and subsequently $I_5$ is also included. In order that the correct combination should be achieved we need to check both operators adjacent to non-peripheral operands. In this particular situtaion we could use productions such as

$$T \to I \qquad (E)+ \to T+T+$$

to prevent incorrect manipulation of $I_3$.

Because '*' has the highest precedence of (arithmetic) operators present in the expression we can invoke the balancing of multiplicative trees without reference to '+' and hence we use

$$(T) \to T*T$$

Indeed, when an operator of highest precedence is used, no checks on the types of adjacent operators are required except in the case when the operator in question is not associative. In parsing terms the delimited expression

$$\$I_1+I_2+I_3*I_4*I_5\$$$

can now be reduced to $\quad \$(E_2)+(T_2)*I_5\$$

and then to $\quad \$(E_2)+(T_2)*T_1\$$

The (incomplete) multiplicative tree must now be reduced. This can be done in several ways. Following the strategy used in the additive grammar, $G_5$, we can enforce left to right balancing by removing the right-hand bracket from '$(T_2)$' and allowing the replacement bracket to be created at the extreme right of the expression. In consequence we have the productions

$$*T \to )*T \qquad \text{and} \qquad )\$ \to \$$$

However, this gives rise to the abortive reduction sequence

(1) $\$(E_2)+(T_2)*T_1\$$ $\qquad$ (4) $\$(E_2+((T))\$$

(2) $\$(E_2)+(T_2*T_1\$$ $\qquad$ (5) $\$(E_2+(T))\$$

(3) $\$(E_2)+(T_2*T_1)\$$ $\qquad$ (6) $\$(E_2+E))\$$

Reference to the desired admisible tree for this expression (Figure 12) reveals the requirement for the right subtree to be higher than the left subtree. Dealing with this phenomenan as before, by using the production and completely destroys the left-to-right balancing of additive trees and hence an alternative method must be found.

Such a method, based on the strict use of brackets to preserve tree height information and hence requiring no bracket inbalance at any stage, is described in four stages.

(i) First, we return to the (single operator) additive grammar and generate

$$L(X) = \{T(+T)^n : n > 0\} \text{ by the CS productions}$$

(1 $X \to \$S\$$, $\quad$ (2) $S \to \{S\}$, $\quad$ (3) $S \to E$, $\quad$ (4) $+E\} \to \}+E$,
(5) $+ \to \}+\{$, $\quad$ (6) $\{E\} \to E+E$, $\quad$ (7) $E \to T$ $\quad$ (Grammar 7)

Now any reduction sequence associated with grammar 7 has equal numbers of open and close brackets at every stage and before any application of rule 2, the number of balanced bracket pairs indicates the height of the structure tree. However, rule 4 is too general for our purposes, it allows any combination of trees $E_i$ and $E_j$ (in that order) where $h(E_i)>h(E_j)$. In particular it gives rise to the following reduction of $E_7$.

(1) $\$T+T+T+T+T+T+T\$$ $\qquad$ (6) $\$\{\{E+E\}\}+E\$$

(2) $\$E+E+E+E+E+E+E\$$ $\qquad$ (7) $\$\{\{\{E\}\}\}+E\$$

(3) $\$\{E\}+\{E\}+\{E\}+E\$$ $\qquad$ (8) $\$\{\{\{E+E\}\}\}\$$

(4) $\$\{E+E\}+\{E\}+E\$$ $\qquad$ (9) $\$\{\{\{\{E\}\}\}\}\$$

(5) $\$\{\{E\}+E\}+E\$$

This implies a tree of height 4 and relates to the tree in Figure 13 corresponding to the analysis $E_7=(E_4+E_2)+E_1$ instead of $E_7=E_4+(E_2+E_1)$. Both are possible analyses from grammar 7.

So there is a requirement for some indicator to stimulate the accepting of incomplete trees from right to left. This is easily achieved by a simple modification given as grammar 8 in which γ and δ are used as markers; each γ permitting a possible imbalance at each level and the δs preventing such an imbalance from occurring to the left of the central '+' at the corresponding level of the right sub-tree.

```
(1) X→$S$,  (2) S→{S}γ,  (3) S→E,  (4){E}→E+δE,
(5) +→}+{,  (6) E→T,     (7) γ→Λ,  (8) δ→Λ,
(9) +δE}γ→}+δE                      (Grammar 8)
```

(Notice that grammar 8 is not context-sensitive. However, its form enables the balancing methodolgy to be clearly seen and can be modified later to meet CS criteria.) The skeletal derivation of $E_7$ from grammar 8 is as follows: the underlying semantic tree being indicated by the arrowed progression of '+' symbols.

```
              X
 *
 =>  -  ${{{E}γ}γ}γ$
 *
 =>  =  ${{E+δEγ}γ}γ$
                ↓
 *
 => ${{E}  +  {E}γ}γ$
        ↓   ↓   ↓
 *
 => ${E+δE + E +δEγ}γ$
        ↓   ↓   ↓
 *
 => ${E+ E + E +δE}E}γ$
        ↓   ↓   ↓
 *
 =>${E}+{E}+{E}+δE$
     ↓ ↓ ↓ ↓    ↓
 *
 =>$E+E+E+E+E+E+E$
```

A context-senstivive equivalent of grammar 8 is given by the productions

```
(1) X→$S$,   (2) S→{S},   (3) S→{SΓ, (4) S→E,
(5) {E}→E+E, (6) {E}→E+F, (7) +→}+{, (8) E→T,
(9) +FΓ→}+E, (10) +FΓ→}+F           (Grammar 9)
```

(ii) The second stage in our grammar modifications is to introduce different kinds of brackets at different levels of the syntax thus enabling the balancing of multiplicative subtrees to be performed without regard to, and chronologically before, manipulation of any surrounding additive structure. This could be done syntactically but use of a distinct bracketing system allows processing of the entire multiplicative substructure to be analysed and reduced to an atomic additive term (of a suitable height) which can then not be dissected, by subsequent additive manipulation. In particular it can be included at the left- or right-hand side of a higher tree at will without creating an inadmissible tree or a tree which is higher than necessary.

Proceeding in this direction we can take the terminal symbol T of grammar 9, reclassify it as a non-terminal, and define it so as to generate multiplicative subexpressions, balanced from left to right. Accordingly we require the production.

```
(1) T→[T],      (2) T→[TΠ,      (3) T→P,
(4) [P]→P*P,    (5) [P]→P*P,    (6) *→]*[,
(7) *RΠ→]*P,    (8) *RΠ→]*R,    (9) P→I,
                                (Productions 9a)
```

Using the sets of productions 9 and 9a together to generate $\$I_1+I_2+I_3*I_4*I_5\$$ we obtain

```
      X         |   => ${{S}Γ$   |  => ${E}+E$
 => $S$         |   => ${{E}Γ$   |  => $E+E+T$
 => ${SΓ$       |   => ${E+FΓ$   |     etc
```

This reduction implies that h(T)=0 ie the term is atomic whereas the true position would be indicated by [[T]], say, having height 2 and corresponding (additively) to an expression such as $E_4$. Using the same productions, the required information is availble (at stage *) in the subsequent derivation, namely

```
      T            |  =>  [[P]Π   |  =>  *P*P*P
 => [TΠ            |  =>  [P*RΠ   |  =>  I*I*I
 => [[T]Π     (*)  |  =>  [P]*P   |
```

Utilization of this information is dealt with in the next two stages of the balancing process for the class of expressions in question.

(iii)-(iv) We now need to convert multiplicative brackets to additive ones and also allow the expression rather than the grammar to dictate left or right associativity of the resulting subexpression within the surrounding expression. Consequently additive brackets created by this process behave in more general ways than those occurring elsewhere and hence there is a requirement to deal with both of these processes together.

Explicitly, recall that indication of the height of the multiplicative subtree in $\$I_1+I_2+I_3*I_4*I_5\$$ is 2 and could be given by replacing [[T]Π by {{E}}. If done directly this yields the intermediate form $\$\{E\}+\{\{E\}\}\$$ which fails because it balances to the right and not to the left. In order to be able to accept such a form we need to know that the right subexpression is derived from multiplication.

We return to the form: $E}+[[T]Π$
To allow such a composition we need to achieve the progression

```
 =>${E}+[[T]Π$  |  => ${{E+T}}$  |  => ${{{E}}}$
 =>${E+[T]}$    |  => ${{E+E}}$  |
```

- indicating a tree of height 3.

This is done by incorporating the replacement of the leftmost '[' by '{' with the absorption of '}+{' into '+' and with the conversion of the corresponding close bracket, here 'Π', to '}'. To perform the required transformation we use α as a marker in the following productions.

```
(1) +α→}+[,     (2) [α→α[,      (3) Tα→αT,
(4) ]α→α],      (5) Πα→αΠ,      (6) }}→Πα},
(7) }$→Πα$      (8) }}→]α},     (9) }$→]α$,
```

Next, we must migrate any subsequent '[' to the left and convert that bracket and its corresponding ']' into curly brackets. We use $\alpha$ thus

$$\{E+\alpha\rightarrow E+[\quad,\qquad T\alpha\rightarrow\alpha T\quad,\qquad ]\alpha\rightarrow\alpha]$$
$$\Pi\alpha\rightarrow\alpha\Pi \qquad \text{etc. as before.}$$

(This sequence of productions again includes a non context-sensitive rule. At the risk of introducing parsing difficulties, the offending rule could be replaced by

$$Y+\alpha\rightarrow E+[$$

with corresponding changes, such as

$$+E\rightarrow\}+Y\ ,\qquad S\rightarrow Y\}\ ,\qquad S\rightarrow Y\Gamma$$

elsewhere in the grammar.)

The manipulation of high multiplacative subtrees to the left of an addition operator can be dealt with by straight forward bracket conversion and the additive balancing processing provided that extra conversion initiation rules are included, namely

$$\$\{\alpha\rightarrow\$[,\qquad \{\{\alpha\rightarrow\{[$$

and corresponding conversion rules

$$\{+\rightarrow]\alpha+,\qquad \{+\rightarrow\Pi\alpha$$

Yet again the modification would introduce non-context senstivive rules. However, their removal necessitates the inclusion of more markers hence the rules will be left in their present form. Justification for this decision will be given in section 6 of the paper.

We are now in a position to expand the sets of rules 9 and 9a into a full grammar. The only significant change being the omission of the first two rules in 9a, thus ensuring that multiplicative brackets are only obtained by conversion from additive ones.

The productions of the Grammar $G_{10}$, are displayed in two columns denoting the rules used in CBBTs and (additionally) in BBTs respectively.

$$G_{10} = (\{X,S,E,'\{','\}',T,P,'[',']',\ \alpha,\Gamma,F,R,\Pi\},\ \{I,+,*,\$\},P_{10},X)$$

| | | | | | |
|---|---|---|---|---|---|
| where $P_{10}$ = | | | | where | |
| 1) | $\{\{X\rightarrow\$S\$$ | | | | |
| 2) | $S\rightarrow\{S\}$ | $S\rightarrow\{S\Gamma$ | 10) | $P\rightarrow I$ | |
| 3) | $S\rightarrow E$ | | | | |
| 4) | $\{E\}\rightarrow E+E$ | $\{E\}\rightarrow E+F$ | | | $+\alpha\rightarrow\}+[$ |
| 5) | $+\rightarrow\}+\{$ | | 11) | $[\alpha\rightarrow\alpha[$ | |
| | | | 12) | $T\alpha\rightarrow\alpha T$ | |
| 6) | $E\rightarrow T$ | | 13) | $]\alpha\rightarrow\alpha]$ | |
| | | $+F\Gamma\rightarrow\}+E$ | | | $\Pi\alpha\rightarrow\alpha\Pi$ |
| | | $+F\Gamma\rightarrow\}+F$ | 14) | $\}\}\rightarrow]\alpha\}$ | $\}\}\rightarrow\Pi\alpha\}$ |
| 7) | $T\rightarrow P$ | | 15) | $\}\$\rightarrow]\alpha\$$ | $\}\$\rightarrow\Pi\alpha\$$ |
| 8) | $[P]\rightarrow P*P$ | $[P]\rightarrow P*R$ | 16) | $\}+\rightarrow]\alpha+$ | $\}\ \rightarrow\Pi\alpha+$ |
| 9) | $*\rightarrow]*[$ | | | | $\{E+\alpha\rightarrow E+[$ |
| | | $*R\Pi\rightarrow]*P$ | 17) | $\$\{\alpha\rightarrow\$[$ | |
| | | $*R\Pi\rightarrow]*R$ | 18) | $\{\{\alpha\rightarrow\}[\ \}$ | |

We conclude this section with outline derivations of three non-trivial expressions using $G_{10}$.

Example 1: $\$I_1+I_2*I_3+I_4*I_5\$$

(for structure see Figure 14)

| | | |
|---|---|---|
| $\overset{*}{\Rightarrow}\ \$\{\{\{E\}\Gamma\}\$$ | $\Rightarrow\ \$\{\{E\}\}+\{E\}\$$ | $\Rightarrow\ \$E+[T]+[T]\$$ |
| $\Rightarrow\ \$\{\{E+F\Gamma\}\$$ | $\overset{*}{\Rightarrow}\ \$\{E+E\}+[T]\$$ | $\overset{*}{\Rightarrow}\ \$I+I*I+I*I\$$ |
| $\Rightarrow\ \$\{\{E\}+\{E\}\$$ | $\overset{*}{\Rightarrow}\ \$\{E+\alpha T]+[T]\$$ | |

($\Gamma$ indicates RH subtree of 1 unit lower than LH subtree)

Example 2: $\$I+I+I*I*I\$$ (Figure 15)

| | | |
|---|---|---|
| $\overset{*}{\Rightarrow}\ \$\{\{\overset{X}{\{E}\}\}\}\$$ | $\Rightarrow\ \$\{E+[T]\}\$$ | $\Rightarrow\ \$\{E+[T]\Pi\$$ |
| $\overset{*}{\Rightarrow}\ \$\{\{E+T]\alpha\}\$$ | $\Rightarrow\ \$\{E+[T]\Pi\alpha\$$ | $\overset{*}{\Rightarrow}\ \$I+I+I*I*I\$$ |
| $\overset{*}{\Rightarrow}\ \$\{\{E+\alpha T]\}\$$ | $\overset{*}{\Rightarrow}\ \$\{E+\alpha[T]\Pi\$$ | |

($\Pi$ indicates incomplete multiplicative right subtree)

Example 3: $\$I+I+I*I*I+I\$$

Here we have the expression in example 2 included as an admissible subexpression Figure 16 hence,

| | |
|---|---|
| $\overset{*}{\Rightarrow}\ \$\{\{\overset{X}{\{\{E}\}\Gamma\Gamma\Gamma\$$ | $\overset{*}{\Rightarrow}\ \$\{\{\{E\}\}\}+E\$$ |
| $\Rightarrow\ \$\{\{\{E+F\Gamma\Gamma\Gamma\$$ | $\overset{*}{\Rightarrow}\ \$I+I+I*I*I+I\$$ |
| $\Rightarrow\ \$\{\{\{E\}+F\Gamma\Gamma\$$ | |

## Further Extensions to The Grammar

So far we have dealt only with expressions constructed from simple identifiers, I, and the arithmetic operators '+' and '*'. Extending the language to handle more complex arithmetic atoms (such as numeric constants) and, without regard to their semantic properties, the operator symbols '-' and '/', is trivial. Similar methods applied to Boolean expressions are also obviously possible. Moreover, because our grammatical structure now incorporates information about the height of the respective semantic tree, the inclusion of the usual parentheses, '(' and ')', causes no difficulty; although introduction of the markers required to invoke the conversion of matching brackets, '{' and '}', and '[' and ']'; over the syntactic brackets necessitate many more productions.

Extension of these techniques to larger portions of programming languages may not seem feasible. However, recall that our basic construction eminates from the introduction of ambiguities to a suitable operator-precedence grammar. Such a grammar for an Algol-like language already exists, Floyd [9].

From the more pragmatic and practical standpoint we need to investigate implications of 'live-dead' variable analysis, Sheafer [10] etc. If this information could be detected and manipulated by syntactic means it would allow (i) for the inclusion of function calls within expressions and hence (ii) for the handling of statements as expressions with side-effects.

Nevertheless, even without these further semantic constraints, what we have so far is a system that enforces the necessary structure on the evaluation of a language fragment but, because

## Parsing Considerations

All the grammars presented in this paper are, or can be, written to conform to Context-Sensitive requirements. This is apparently the narrowest general classification that is satisfied by our grammars and hence before these grammars can be put to work we need to have a suitable parsing machine. In Abdollazadeh [1] various published algorithms are described and compared. Also included are details of several other parsing strategies applicable to certain of the grammars contained therein. However, most of these algorithms are more powerful than necessary since they can be used to analyse sentences of languages which are not context-free.

This 'sledge-hammer to crack a nut' situation has been studied before by Baker [5] and Aggarwal [2]. Indeed Aggarwal defines the set of so-called SVMT-bounded grammars which are CS but generate CFL's. This class of grammars extend previous classifications guaranteed to generate CFLs but does not include all the CS grammars developed here and in Abdollazadeh [1].

Because of the special form of our grammars and the possibility that the most manageable forms (from the parsing standpoint) may not be context-sensitive but even more general, we need to investigate, inter-dependently, parsing methodology and grammar modification. From our initial investigations it would appear that the most important need is to lessen the non-determinism provided that this is conversant with the retention of sufficient structural information.

## Conclusion

We have demonstrated that it is possible to balance simple arithmetic trees syntactically and indicated how the method can be extended to deal with more complicated program fragments. The balancing technique not only determines the minimal height of the expression tree, (subject to certain semantic constraints) but also allows for potential overlap of adjoining trees. Figures 17 and 18 depict a general and a specific situation.

As is seen from the latter figure, the diagonal length •• indicative of idealised execution time - is related to tree height and additionally, the diagonal width is related to the number of independent processors that could be usefully employed in the evaluation. This visualization is also capable of indicating store access operations and inter-machine transfers.

Our work is now to stretch (semantically motivated) syntactic methods as far as is practical and then introduce the minimal semantic structure (such as left-to-right attributes, Bochmann [60]) so as to move the less complex aspects of a language into the earlier phases of compilation and to facilitate more efficient code generation for however many processors are available at run-time.

## References

1. Abdollahzadeh, F, "Specification and Analysis of the Sequencing of Computing Operations in High Level Languages, with Particular Reference to Parallel Processing," Ph.D. Thesis, Loughborough University of Technology, UK (1981).

2. Aggarwal, S. K. and J. A. Heimen, "A General Class of Non-Context-free Grammars Generating Context-free Languages," Information and Control, Vol. 43, (1979) pp. 187-194.

3. Aho, A. V. and J. D. Ullman, The Theory of Parsing Translation and Compiling, Vols. 1 and 2, Prentice-Hall International (1972).

4. Ashoke, D. and A. Murhapadhyay, "Parallel Algorithms for Evaluations of Arithmetic Expressions," Dept. of Computer Science, Iowa University, Iowa City, Iowa, U.S.A. (1976).

5. Baker, B., "Non-Context-free Grammars Generating Context-free Languages," Information and Control, Vol. 24 (1974) pp. 231-246.

6. Bochmann, G. V., "Semantic Evaluation from Left to Right," Comm. ACM 19 (2), (1976) pp. 55-62.

7. Evans, D. J. and F. Abdollahzadeh, "An Efficient Method for the Construction of Balanced Binary Trees for Parallel Processing," Computer Journal Vol. 26 (1983) pp. 193-195.

8. Evans, D. J. and S. A. Williams, "On The Construction of Balanced Binary Trees for Parallel Processing," Algorithm 99, Comp. J. Vol. 20 (1977) pp. 378-379.

9. Floyd, R. W., "Syntactic Analysis and Operator Precedence," J. ACM 10(3), (1963) pp. pp. 316-333.

10. Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall (1973).

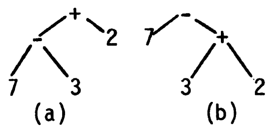

(a) (b)

Figure 1

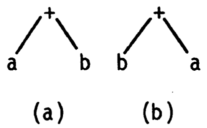

(a) (b)

Figure 2

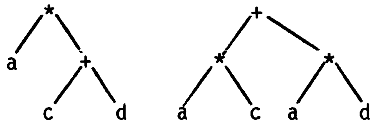


Figure 3

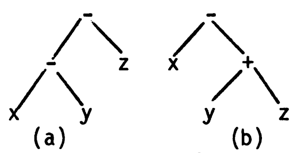

(a) (b)

Figure 4

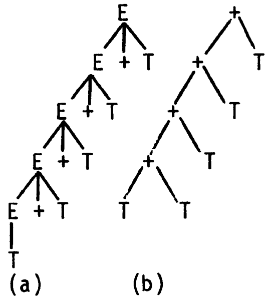

(a) (b)

Figure 5

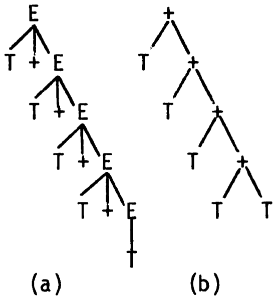

(a) (b)

Figure 6

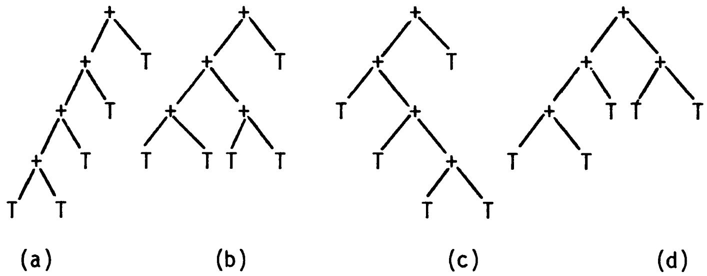
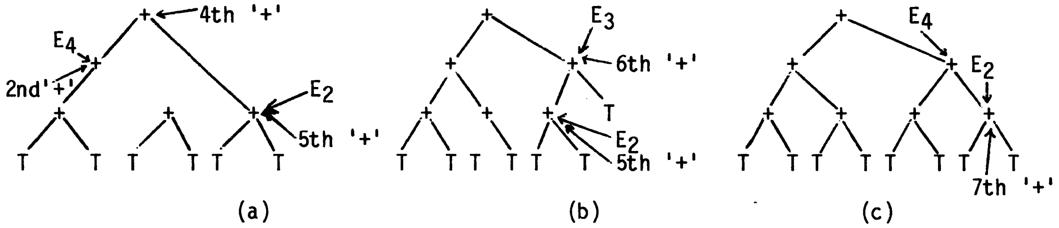

(a) (b) (c) (d)

Figure 7

| | + | T | $ |
|---|---|---|---|
| + | •><• | <• | •> |
| T | •> | | •> |
| $ | <• | <• | |

Figure 8



(a) (b) (c)

Figure 9

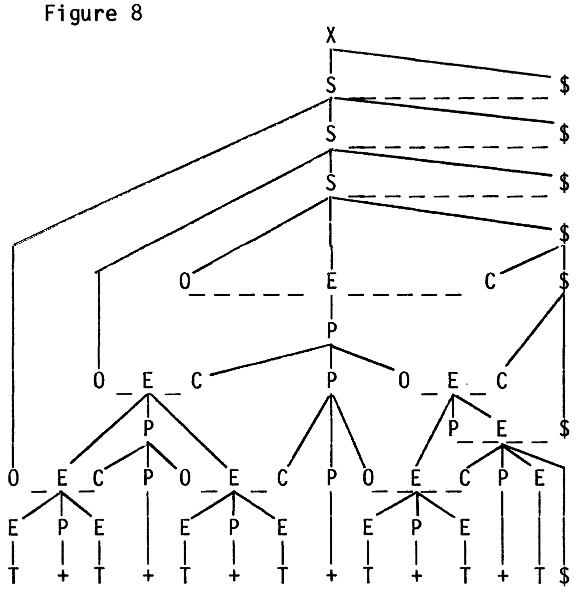


Figure 10

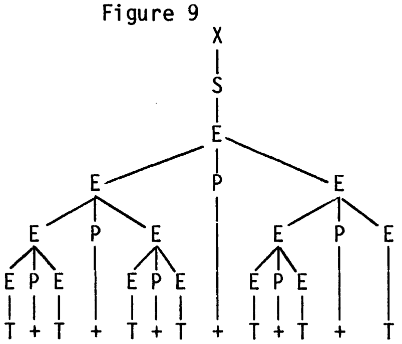


Figure 11

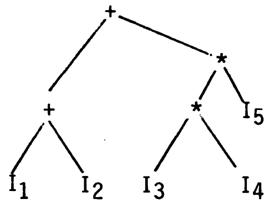


Figure 12

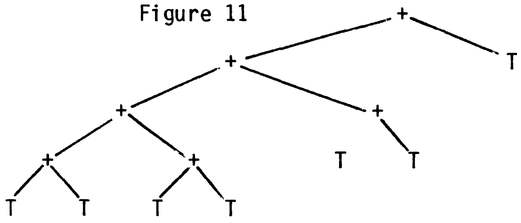


Figure 13

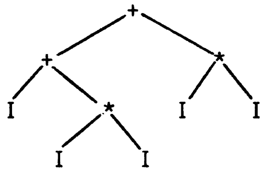


Figure 14

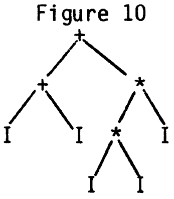


Figure 15

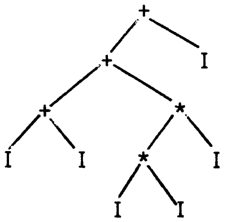


Figure 16

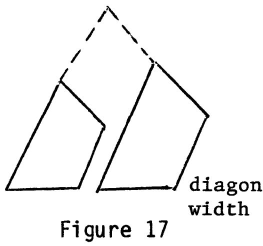


Figure 17

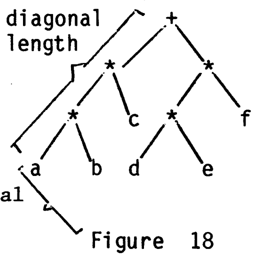


Figure 18

# A Pipelined Solution Method of Tridiagonal Linear Equation Systems
*

GUANG R. GAO

*Laboratory for Computer Science*
*Massachusetts Institute of Technology*
*Cambridge, MA 02139*
*Jan. 1985*

## Abstract

A pipelined solution method for tridiagonal systems of linear equations is proposed which introduces parallelism in a way that may be effectively exploited by static data flow computers. It eliminates the substantial data rearrangement overhead incurred by many existing parallel algorithms, and sustains a relatively constant parallelism during various phases of program execution. Using the new method, we outline a pipepined code mapping scheme. The principle outlined in this paper may be extended to other suitable parallel machine architecture.

## 1. Introduction

Tridiagonal systems of linear equations form a very important class of linear algebraic equations. For example, the heart of finite difference solutions of partial differential equations may consist of tridiagonal systems of equations. A tridiagonal system of linear equations can be solved on a conventional computer using the classical *Gaussian elimination algorithm*. However, such solution method is sequential in nature and hence unsuitable for parallel computers without drastic alteration.

In the past decade, new techniques have appeared for solving tridiagonal systems of equations with parallel computers [11, 15]. The best known parallel algorithm is based on the cyclic reduction technique, first proposed by Golub and Hockney and applied by Buzbee et al, for solving tridiagonal system of equations efficiently [2]. One approach is using such parallel technique to solve the recurrences established by the LU-decomposition method of Gaussian elimination algorithm. One such algorithm, known as recursive doubling suggested by Stone and originally designed for Illiac IV, was later modified for other vector computers [16]. Another approach has resulted from considering the needs of parallel processing in the first place and trying to design fundamentally new algorithms which are inherently more parallel. The *odd-even cyclic reduction* algorithm is base on such a principle [2]. A major difficulty with the algorithms based on cyclic reduction technique is the overhead of data rearrangement between computation steps and the considerable variations of degree of parallelism between computation steps.

In this paper, a new method for solving tridiagonal systems of equations is proposed which introduces parallelism in a way that may be effectively exploited by data flow computers. The algorithm is based on the maximally pipelined solution of linear recurrences presented in a companion paper [8]. It performs a program transformation of the recurrences generated in the Gaussian elimination method to produce machine code which can be executed in a maximally pipelined fashion. The new method eliminates the substantial data rearrangement overhead incurred by many existing parallel algorithms and it sustains a relatively constant parallelism during various phases of program execution. Based on this scheme, the code structure of a maximally pipelined tridiagonal equation solver is outlined for a static data flow supercomputer. The principle outlined in the paper may be extended to other data flow computers.

## 2. Background and Related Work

In this section we state briefly the problem of tridiagonal system of linear equations and review the directed methods for solving them — such as the Gaussian elimination algorithm, in particular the linear recurrences established by the LU-decomposition technique. We also survey the related work of parallel tridiagonal solution methods, such as the well-known cyclic reduction technique.

* The research described in this paper and its two companion papers [8, 10] was supported by the Department of Energy and the National Science Foundation.

0190-3918/86/0000/0084 $01.00 © 1986 IEEE

### 2.1 Statement of The Problem

We consider the solution to the following tridiagonal set of linear equations:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & & 0 \\ a_2 & b_2 & c_2 & & & \vdots \\ 0 & a_3 & b_3 & & & \\ \vdots & & & \ddots & & 0 \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & 0 & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \cdot \\ \cdot \\ \cdot \\ k_n \end{pmatrix}, \qquad (2.1)$$

or expressed in matrix-vector notation

$$Ax = k \qquad (2.2)$$

In this paper, our major concern will be the case where the coefficient matrix A is positive definite or at least pivoting is not required.

### 2.2 LU-Decompusition

There are a number of serial methods for solving the tridiagonal system as expressed in (2.1). The maximally pipelined solution method to be developed in this paper is based on the well-known *LU-decomposition* technique [5]. The Stone's recursive doubling algorithm to be discussed later is also based upon such technique. In this method, we find two matrices, L and U, such that

(1) LU = A;

(2) L is a lower bidiagonal matrix;

(3) U is an upper bidiagonal matrix with 1s on its principal diagonal.

When A is non-simpler, its LU decomposition is unique. In fact, it is shown that

$$L = \begin{pmatrix} e_1^{-1} & 0 & \cdots & & 0 \\ a_2 & e_2^{-1} & & & \vdots \\ 0 & a_3 & e_3^{-1} & & \\ \vdots & & & \ddots & 0 \\ 0 & \cdots & 0 & a_n & e_n^{-1} \end{pmatrix}$$

and

$$U = \begin{pmatrix} 1 & u_1 & 0 & \cdots & 0 \\ 0 & 1 & u_2 & & \vdots \\ \vdots & & 1 & \ddots & 0 \\ & & & \ddots & u_{n-1} \\ 0 & \cdots & & 0 & 1 \end{pmatrix}$$

where

$$\begin{aligned} u_1 &= c_1/b_1 \\ u_i &= c_i/(b_i - a_i u_{i-1}) \qquad i = 2,3 \ldots n-1 \qquad (2.3) \\ e_i &= u_i/c_i \end{aligned}$$

After computing L and U, it is relatively straightforward to solve the system of equations by a two-step process. First, letting Y = Ux, we have

$$Ly = K \qquad (2.5)$$

$$Ux = y \qquad (2.6)$$

and together, we have Ax = LUx = Ly = k.

The equation Ly = k can easily be solved for y as follow.

$$\begin{aligned} y_1 &= k_1/b_1 \\ y_i &= (k_i - a_i y_{i-1})/(b_i - a_i u_{i-1}) \qquad i = 2,3 \ldots n \qquad (2.7) \end{aligned}$$

Note that in the solution process, as indicated by (2.7), there is no need to compute $e_i$ explicitly unless the matrix L is needed in other places. Next, we solve Ux = y for x by noting that

$$\begin{aligned} x_n &= y_n \\ x_i &= y_i - u_i x_{i+1} \qquad i = n-1, n-2 \ldots 1 \qquad (2.8) \end{aligned}$$

The two steps of (2.7) and (2.8) are often called *forward-elimination* and *backward-substitution*. The recurrences (2.3), (2.7) and (2.8) constitute a complete solution for Ax = k. and a sequential algorithm to perform such a solution is the so-called *Gaussian elimination algorithm*.

### 2.3 Cyclic Reduction Technique

In this paper, we make no attempt to survey all parallel algorithms for tridiagonal linear equation solvers, but review only two well-known methods which are based on the cyclic reduction. This discussion will motivate the pipeined solution presented in this paper.

#### 2.3.1 Recursive Doubling Algorithm

The recursive doubling algorithm proposed by Stone [16] began with the observation that the formula required by LU factorization, such as (2.7) and (2.8), are first-order linear recurrences (FLR). The equation (2.3) appears not to be a linear recurrence. However, if we introduce a new variable $q_i$ such that $u_i = -q_i/q_{i+1}$, then (2.3) can be transformed into the following second-order linear recurrence (SLR):

$$a_i q_{i-1} + b_i q_i + c_i q_{i+1} = 0 \qquad i = 2,3,\ldots,n-1 \qquad (2.9)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$.

Stone pointed out that (2.9) can be transformed into a first-order linear recurrence except that the sequence is now a set of vectors instead of scalar values. The recursive doubling algorithm uses standard cyclic reduction method for handling linear recurrences to solve them.

It is helpful to review the cyclic reduction technique for solving linear recurrences before we outline its disadvantages. For instance, we consider the evaluation of the sequence of $x_i$ from the following first-order linear recurrence relation.

$$x_i = a_i x_{i-1} + b_i \qquad \text{for } i = 2 \ldots n \qquad (2.11)$$

where $x_1, a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ are known values. The basic idea of standard cyclic reduction technique is to back up the recurrence in (2.11) such that a new recurrence can be obtained which relates every other term, every fourth term, every eighth term, etc.

For example, from (2.11) we have

$$\begin{aligned} x_i &= a_i a_{i-1} x_{i-2} + a_i b_{i-1} + b_i \\ &= a_i^{(1)} x_{i-2} + b_i^{(1)} \qquad (2.12) \end{aligned}$$

where $a_i^{(1)} = a_i a_{i-1}$, $b_i^{(1)} = a_i b_{i-1} + b_i$. The superscript (1) denotes the fact that this is a first level backup. Such a backup process can be repeated (in a cyclic fashion) and we obtain a set of equations as follow.

$$x_i = a_i^{(l)} x_{i-2^l} + b_i^{(l)} \qquad (2.13)$$

where

$$a_i^{(l)} = a_i^{(l-1)} a_{i-2^{l-1}}^{(l-1)} \qquad (2.14)$$
$$b_i^{(l)} = a_i^{(l-1)} b_{i-2^{l-1}}^{(l-1)} + b_i^{(l-1)} \qquad (2.15)$$

with $l = 0, 1 \ldots \log_2 n$, $i = 2, 3 \ldots n$. An important observation is that if any of $a_i$, $b_i$ or $x_i$ is outside the defined range, its value can be taken as zero. Therefore, when $l = \log_2 n$, all $x_i$ are solved by

$$x_n = b_n^{(\log_2 n)}$$

We can have the following observation:

(1) high parallelism exists at certain phases (steps) of the algorithm, i.e., at a fix level l, (2.14) and (2.15) can be evaluated for all i in parallel;

(2) the parallelism grows roughly linearly with the size of the vectors — i.e. n in this case;

(3) the useful parallelism decreases as the computation progressing through different phases.

The amount of parallelism varies between phases of computation. This will increase the difficulty of fully utilize the parallelism of the machine.

### 2.3.2 Odd-even Reduction Algorithm

The *odd-even cyclic reduction algorithm* is perhaps the most successful cyclic reduction algorithm applied to solve tridiagonal systems [2]. It starts directly from the system of equations defined by (2.1), i.e.,

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = k_i \qquad i = 1, 2 \ldots n^*$$

The algorithm first eliminates the odd numbered variables in the even numbered equations by performing elementary row operations. In each level, we cut down the total number of equations by 1/2, hence, in $\log_2 n$ levels, the middle element $x_{n/2}$ can be computed directly from the coefficients. The remaining unknowns can be found by a refilling procedure. This algorithm also involves the recursive calculation of coefficients for equations at each level. One important advantage of the odd-even reduction over the recursive doubling algorithm is that it reduces the number of operations considerably at each level, and the total number of operations is on the order of O(n).

One major difficulty with odd-even reduction is the data rearrangement of variable and coefficient vectors between phases of computation. For example, on the Cyber 205 one cannot apply vector operations directly to every other elements of the vector. Thus extra operations must be employed to reformat those elements into a new vector [14]. On the Cray it is possible to access elements of a vector at a fixed increment, but this may result in a performance degradation [12]. Because of the overhead of data rearrangement, the cyclic reduction algorithm may run slower than a serial algorithm for sufficiently small n [15].

Another problem is the degree of variation of parallelism between different phases of computation. Because the parallelism decreases very rapidly, this problem becomes more serious than that for the recursive doubling algorithm. A parallel version of odd-even reduction algorithm has been proposed to keep a high parallelism throughout the computation. However, it increases the number of operations significantly to O(nlogn) [11].

## 3. A Pipepined Solution Scheme for Linear Recurrences

### 3.1 Overview of the Pipelined Solution Scheme

In cyclic reduction scheme, the goal is to increase the speed through fully exploiting the parallelism in the original problem. High concurrency is obtained by replicating the

* In the remaining discussion of this section, we assume n is a power of 2, but this is not an essential assumption.

operations as much as necessary to compute all elements in the result vector in parallel. In contrast, we propose a new solution method which can explore and organize the parallelism in a way that best matches a suitable computer architecture, i.e. the static data flow architecture. It is based on a maximally pipelined code mapping scheme developed in our previous work [3,6].

Two forms of parallelism exist in a data flow machine level program, as shown in Figure 1, which consists of seven actors

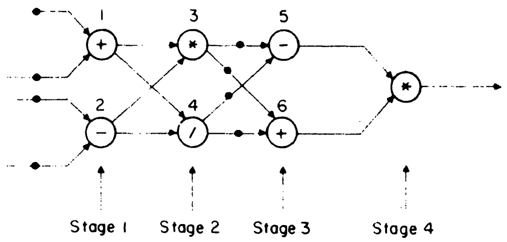


FIG. 1. Pipelining of data flow programs.

divided into four stages. In Figure 1 (a), actors 1 and 2 are enabled by the presence of tokens on their input arcs, and thus can be executed in parallel.* This is called *spatial* parallelism. Spatial parallelism also exists between actors 3 and 4, and between actors 5 and 6. The second form of parallelism is *pipelining*. In static data flow architecture, this means arranging the machine code such that successive computations can follow each other through one copy of the code. If we present a sequence of values to the inputs of the data flow graph, these values can flow through the program in a *maximally pipelined* fashion — i.e. input/output values are consumed/produced at maximum rate allowed by the machine architecture. In the configuration of Figure 1 (b), two set of tokens are pipelined through the graph, and the actors in stage 1 and 3 are enabled and can be executed concurrently. Thus, the two forms of parallelism are fully exploited in the graph.

The power of pipelined computation in a data flow computer can be derived from machine-level programs that form a large pipeline in which many actors in various stages are executed concurrently. Each actors in the pipe are activated in a totally data-driven manner, and no explicit sequential control is needed. With data values continuously flow through the pipe, a sustained parallelism can be efficiently supported by the data flow architecture.

### 3.2 Maximally Pipelined Mapping of Linear Recurrences

An attractive way to implement recurrence on data flow computers is to introduce feedback paths in the data flow graph. This, however, presents particular problems when maximum pipelining of the program is desired. A direct translation of the first order recurrence is shown in Fig. 2. The value $x_i$ depends on the value of $x_{i-1}$, therefore, a *feedback path*, such as the one marked in the graph, is generated. The key is to understand the role of the merge operator (denoted by M in Fig. 2): (1) under the *merge control* input values (<FT...T>), the initial output value of the loop is taken from the second input of the merge, i.e., $x_1$. (2) the upper output of M is routed under the *feedback control* values, i.e. <T...TF>, therefore all but the last element of the array will be fed back; and (3) the lower output of the merge is forwarded as the output of the loop unconditionally. Due to the existence of cycles, the data flow graph produced by such a scheme, in general, cannot be fully pipelined. More specifically, the feedback link between the output of cell 3 and the input of cell 1 prevents the whole graph from being fully pipelined.

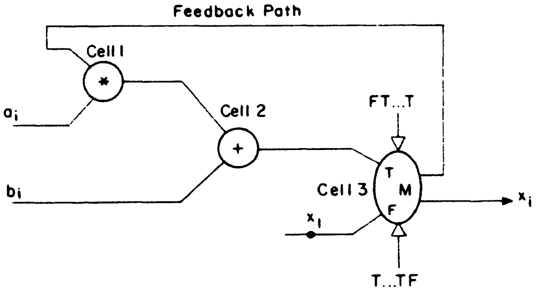


FIG. 2. The pipelined mapping of a first-order linear recurrence.

The problem of the above example and its solution have been studied by the author in [3, 6]. The problem is essentially a mismatch between the *dependence delay*— the dependence inherent in the recurrence (i.e., $x_i$ depends on $x_{i-1}$, therefore a two-stage feedback delay is required) and the *computational delay*—the actual length of the loop in the data flow graph generated by the direct translation scheme (3 stages in this example). In [8], the author described a solution for such a problem on a static data flow computer, based on the concept of *companion functions* [3,13]. It is essentially a way to remove the dependence of $x_i$ on $x_{i-1}$, thus, easing the feedback constraints in order to match the computational delay of the data flow graph. For the above example, we have:

$$x_1 = b_1$$
$$x_2 = a_2b_1 + b_2$$
$$x_i = a_ia_{i-1}x_{i-2} + a_ib_{i-1} + b_i \quad \text{where } i \geq 3 \quad (3.1)$$

* A solid disk on an arc represents the presence of a token.

This transformation is interesting to us because $x_i$ now depends on $x_{i-2}$ instead of $x_{i-1}$. Therefore, we can map our example, now expressed as in (3.1), into a data flow graph as shown in Fig. 3. Note that we have introduced two additional pipelines $a_i'$ and $b_i'$ as denoted by the dotted lined box C, where

$$a_i^{(1)} = a_i a_{i-1}$$
$$b_i^{(1)} = a_i b_{i-1} + b_i \qquad \text{where } i \geq 3.$$

This added pipeline is named the *companion pipeline* in [3], and its structure is shown in Fig. 4. To understand how the scheme works we first examine the loop in Fig. 3. The role of the merge operator is as before except that two initial values are presented to the second input of the merge, i.e. $x_1 = b_1$, $x_2 = a_2 b_1 + b_2$. The ID cell plays the role of a FIFO of size 1, and is inserted to tune the computational delay in the feedback path to match exactly the dependence delay. The two boxes in Fig. 4 are also FIFOs, and they can introduce proper skew needed in the pipelining of array operations [6]. The rest of the graph is self-explanatory and the reader should be convinced that the graph is maximally pipelined.

The transformation shown in the above example is equivalent to dividing the first-order linear recurrence into two equivalent classes of computation. In fact, since $x_i$ now depends on $x_{i-2}$, we may split the sequence of results into two subsequences:

$$X' = \langle x_1, x_3, x_5 \ldots x_{2i-1} \ldots \rangle$$
$$X'' = \langle x_2, x_4, x_6 \ldots x_{2i} \ldots \rangle.$$

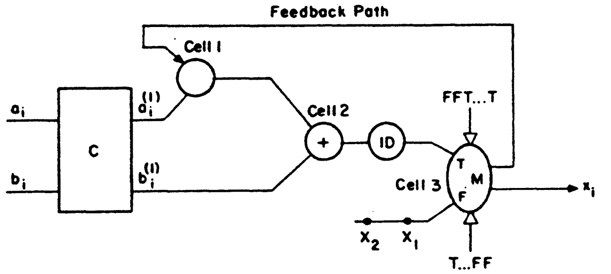


**FIG. 3. Maximally pipelined mapping of first-order linear recurrences.**

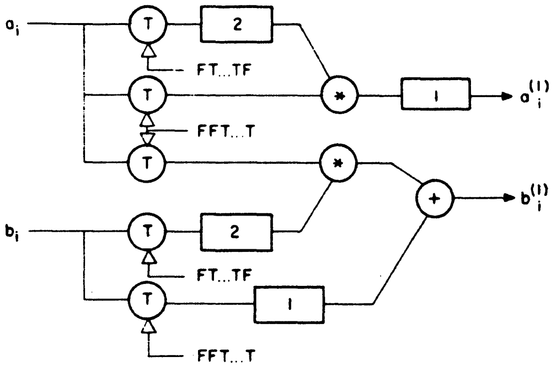


**FIG. 4. The companion pipeline of example (3.1).**

Either sequence can be computed independently by sharing the same loop, and the companion pipeline provides the appropriate input coefficients.

The advantage of this scheme is obvious. First, it does not require data rearrangement during the computation. In fact, it even eliminates the requirement that the input vectors be completely filled before the computation starts. If the input coefficient vectors themselves are generated by some preceding code block in a pipelined fashion, the producer-consumer type of interface technique (as described in [6]) can be applied to save considerable storage space for the intermediate values. Moreover, we observe that the degree of parallelism remains constant (5 floating point arithmetic operations and several other operations) in the computation.* The high throughput is achieved by the maximally pipelined execution of each actor in the data flow program, hence, the storage usage by the machine code is very efficient. Finally, there are no essential limitation on the length of the vectors which can be computed.

## 4. A Maximally Pipelined Method

Starting from LU-decomposition, we can observe that the major equations, such as (2.7) and (2.8), are first-order linear recurrences. Therefore, the pipelined method as described in the last section can be applied directly. Now consider (2.9) which can be transformed into the following second-order linear recurrence:

$$q_i = \alpha_i q_{i-1} + \beta_i q_{i-2} \qquad i = 3,4 \ldots n \qquad (4.1)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$ and

$$\alpha_i = -b_{i-1}/c_{i-1}$$
$$\beta_i = -a_{i-1}/c_{i-1}$$

Performing one level backup we obtain

$$q_i = \alpha_i^{(1)} q_{i-2} + \beta_i^{(1)} q_{i-3} \qquad i=4,5 \ldots n \qquad (4.2)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$, $q_3 = b_2 b_1/c_1 c_2 - a_2/c_2$, and

$$\alpha_i^{(1)} = \alpha_i \alpha_{i-1} + \beta_i$$
$$\beta_i^{(1)} = \alpha_i \beta_{i-1} \qquad i = 4,5 \ldots n$$

Fig. 5 shows a maximally pipelined data flow machine level program for mapping (4.2). The loop in the middle of Fig. 5 can easily be understood by noting its similarity with the loops in Fig. 3. The code in the dotted lined box is the companion pipeline generating values for $\alpha_i^{(1)}$ and $\beta_i^{(1)}$. The node labeled N performs a negation of its input. The boolean value sequences C0 - C5 can be found in Fig. 7. The boxes denote the FIFO buffers which are introduced for balancing the graph [4,6] to achieve maximum pipelining, and the number written inside the box is the number of stages in that buffer. It is easy to check that Fig. 5 correctly computes (2.9) and it is maximally pipelined.

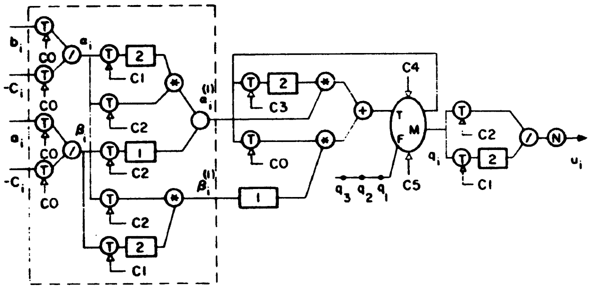


FIG. 5. Pipelined tridiagonal linear equation solver—Part 1.

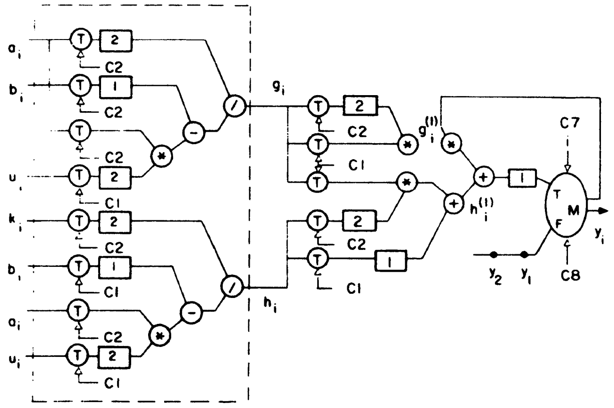


FIG. 6. Pipelined tridiagonal linear equation solver—Part 2.

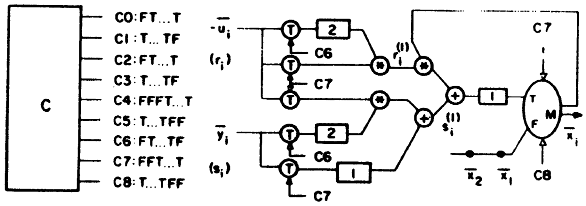


FIG. 7. Pipelined tridiagonal linear equation solver—Part 3.

We rewrite the first-order linear recurrence in (2.7) as

$$y_i = g_iy_{i-1} + h_i \qquad i=2,3...n \qquad (4.3)$$

where $y_1 = k_1/b_1$, $g_i = -a_i/(b_i-a_iu_{i-1})$, $h_i = k_i/(b_i - a_iu_{i-1})$. Performing one level backup we obtain

$$y_i = g_i^{(1)}y_{i-2} + h_i^{(1)} \qquad i = 3,4...n \qquad (4.4)$$

where $y_1 = k_1/b_1$, $y_2 = (-a_2k_1+b_1k_2)/(b_2 - a_2u_1)b_1$,

and

$$g_i^{(1)} = g_ig_{i-1}$$
$$h_i^{(1)} = g_ih_{i-1} + h_i.$$

Fig. 6 shows a maximally pipelined mapping of (4.4). The dotted lined box is the companion pipeline and the boolean sequences C1,C2,C7,C8 can be found in Fig. 7.

Finally, (2.8) can be conveniently treated as a first-order linear recurrence by introducing new variables $\bar{x}_i$, $\bar{y}_i$, $\bar{u}_i$ such that $\bar{x}_i = x_{n-i+1}$, $\bar{y}_i = y_{n-i+1}$, $\bar{u}_i = u_{n-i+1}$. Hence, (2.8) can be rewritten as

$$\bar{x}_i = r_i\bar{x}_{i-1} + s_i \qquad i = 2,3...n \qquad (4.5)$$

where $\bar{x}_i = \bar{y}_i$, $r_i = -\bar{u}_i$ and $s_i = \bar{y}_i$. We can note that (4.5) is a standard first-order linear recurrence, hence we can solve it by one level backup:

$$\bar{x}_i = r_i^{(1)}\bar{x}_{i-2} + s_i^{(1)} \qquad i = 3,4...n \qquad (4.6)$$

where $\bar{x}_1 = \bar{y}_1, \bar{x}_2 = \bar{u}_2\bar{y}_1 + \bar{y}_2$ and

$$r_i^{(1)} = r_ir_{i-1}$$
$$s_i^{(1)} = r_is_{i-1} + s_i$$

Fig. 7 shows a maximally pipelined mapping of (4.6).

Now we have constructed a complete pipelined machine code strucure for a maximally pipelined tridiagonal solver as shown by Fig. 5 - Fig. 7. Fig. 5 and Fig. 6 can be combined into one maximally pipelined data flow graph by observing that the sequence of values of $u_i$ produced by Fig. 5 can be directly fed into Fig. 6. The interface between the outputs of Fig. 5 and

* Here the variation of the parallelism during the start and finish time of the computation is not considered

Fig. 6 and the inputs of Fig. 7 cannot be connected directly. The main reason is that the order in which the elements $y_i$ and $u_i$ are generated by Fig. 5 and Fig. 6 is opposite to the order in which they are used for the maximum pipelining of Fig. 7. Hence, we should first store the values of $u_i$, $y_i$ into memory. Then, Fig. 7 will access the arrays in a reverse order. The code in Fig. 5 and Fig. 6 will sustain a constant parallelism such that there are 20 floating point operations and a number of other operations are concurrently in pipelined operation. When only the code of Fig. 7 is in execution, the parallelism will be reduced to a constant of 5. Although there is such a change of degree of parallelism between the forward elimination and backward substitution phases of the computation, the parallelism remain entirely stable during each phase, hence are easily to be handled by the processors. Furthermore, In Fig. 5 - Fig. 7, the pattern of runtime data routing is regular, thus eliminating the data rearrangement problem for cyclic reduction. Moreover, it essentially can work for tridiagonal systems regardless of their size, hence, has more flexibility and generality than the cyclic reduction scheme.

The reader may wonder if the 5 to 20 folds of parallelism available in the pipelined algorithm may not meet the appetite of a supercomputer. We argue that the major concern should be how the parallelism in the algorithm can be most effectively used by a suitable architecture. First, the new scheme maintains a relatively constant amount of parallelism and relatively simple data routing pattern. Thus, the resource management and allocation problems are more easy to handle, thereby providing better opportunity of parallel processing when the machine has extra power. Second, one is often faced with solving a set of m independent tridiagonal systems (say, m=64), as frequently occurs in the solution of PDE's [11]. In this case, the new scheme can be best used by generating m independent pipelines for each system to obtain 20xm folds of parallelism (more than 1000 if m=64 !). Finally, the new scheme is flexible enough to be extended to obtain more parallelism when such a requirement does occur [9].

## 5. Sumary and Discussions

In developing new parallel algorithms, high importance attached not only to the speed of the greatest computation rate, but to the numerical stability problems as well. The stability aspect of the pipelined tridiagonal solver has been studied by the author in a second companion paper [10].

The entire code for the pipelined tridiagonal linear equation solver has been translated into the machine code of a proposed static data flow supercomputer, and preliminary simulation results indicate that the projected maximally pipelined throughput can be sustained for each of the three loops.

The pipelined solution scheme has several important advantages over other existing parallel algorithms. Although, the primary target machine used in this paper is a static data flow computer, we expect that the principle can also be applied to other data flow computers, such as the dynamic data flow machine [1], although a different perspective of pipelining may be required [7]. It is my belief that the basic ideas may also be useful for a conventional parallel machine architecture. A compiler which can perform the automatic program transformation to implement the pipepined solution scheme is an interesting challenge to the compiler construction for such computers.

## 6. Acknowledgements

The author is indebted to the constant encouragement and direction from Prof. Jack Dennis at MIT. Dr. Bill Ackerman and Kevin Theobald have read the draft and made interesting comments. Natalie Tarbet has provided valuable helps in the improvements of the English for this paper. Finally, the author is grateful to Gao Ping, for her typing of the entire manuscripts and preparation of the figures.

## References

[1] Arvind, Gostelow,K.P. and Plouffe,W, "An Asynchronous Programming Language and Computing Machine", TR-114a, Dept. of Information and Computer Science, Univ. of California, Irvine, Dec. 1978.

[2] Buzbee, B. L. Golub G. H. and Neilson C. W., "On Direct Methods for Solving Poison's Equations", SIAM Journal of Numerical Analysis, 7., 1970.

[3] Dennis, J. B. and Gao, G. R. "Maximum Pipelining of Array Operations on Static Data Flow Machine", Proceeding of the 1983 International Conference on Parallel Processing, Aug 23-26, 1983.

[4] Dennis, J. B., Gao G. R. and Todd, K., "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. on Computers, c-33, No. 7, July 1984.

[5] Forsythe, G. E. and Moler, C. B., "Computer Solution of Linear Algebraic Systems", Prentice-Hall, Englewood Cliffs, N. J., 1967.

[6] Gao, G. R. "An Implementation Scheme for Array Operations in Static Data Flow Computer" MS Thesis, Laboratory for Computer Science, MIT, Cambridge, MA, June 1982.

[7] Gao, G. R., "Maximally Pipelined Throughput and Its Token Storage Requirements for Dynamic Data Flow Processors", IBM Research Report, RC-10785, Computer Science, T.J. Watson Research Center, Oct. 1984.

[8] Gao, G. R. "Maximum Pipelining of Linear Recurrence on Static Data Flow Computers", Computation Structure Group Note 49, Lab. for Computer Science, Aug. 1985.

[9] Gao, G. R. "A Maximally Pipelined Tridiagonal Linear Equation Solver", Computers", To appear on the International Journal On Parallel and Distributed Computing, 1986

[10] Gao, G. R. "Stability Aspects of a Pipelined Tridiagonal Linear Equation Solver", Computation Structure Group Note 48, Lab. for Computer Science, Aug. 1985.

[11] Hockney, R. W. and Jesshope, C. R., "Parallel Computers", Adam Hilger Ltd., 1981

[12] Kershaw, D. "Solution of Single Tridiagonal Linear Systems and Vector Rization of the ICCG Algorithm on the Cray-1", in "Parallel Computations", Ed. by Rodrigue, G. et al., Academic Press, 1982.

[13] Kogge, P. M. "A parallel Algorithm for Efficient Solution of a General Class of Recurrence Equations." IEEE Trans. Comput., Vol. c-22, no. 8, Aug. 1973.

[14] Lambiotte, J. and Voigt, R., "The Solution of Tridiagonal Linear Systems on the CDC Star-100 Computer", ACM Trans. Math Software 1., 1975.

[15] Ortega, J. M. and Voigt, R. G., "Solution of Partial Differential Equations on Vector and Parallel Computers", NASA ICASE Report No. 85-1, 1985.

[16] Stone, H., "Parallel Tridiagonal Equation Solvers", ACM Trans. on Math. Software, Vol. 1, 1975.

# Parallel Decomposition of Matrix Inversion using Quadtrees

David S. Wise
Computer Science Department
Indiana University
Bloomington, IN 47405-4101

**CR categories and Subject Descriptors:**
G.1.3 [**Numerical Linear Algebra**]: Matrix inversion, Sparse and very large matrices; E.1 [**Data Structures**]: Trees; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Array and vector processors, Parallel processors; D.1.1 [**Applicative (Functional) Programming Techniques**].
**General Term:** Algorithms.
**Additional Key Words and Phrases:** Quadtrees, quaternary trees, Gaussian elimination, Pivot Step.

## Abstract

The quadtree representation of matrices is explored, particularly as it admits a parallel matrix inversion algorithm. A version of Gaussian Elimination (full matrix pivoting) is described as an applicative program which minimizes process dispatch by folding the pivot search into the preceding pivot operation. The tree structure incorporates incremental decomposition (for arbitrary, but small, numbers of processors), aids in load balancing, and provides a uniform representation for both scalars and sparse matrices that eliminates all compatibility/bounds checking within the important algorithms. Like other algorithms particularly suited to larger problems (where parallelism pays off), it may be used at the higher level in a hybrid strategy, for example, over pipelined vector-processing on smaller, conventionally represented submatrices.

## Section 1. Introduction

Consider a scenario of parallel or multi- processing with realistic constraints. A machine with $p$ processors is available to implement a matrix algebra package for sparse matrices of size, say, $n \times n$. Restrictions are that $7 < p \ll n$ and that the cost to dispatch/recover a processor is significantly greater than the cost to perform simple arithmetic.

Those restrictions [12] preclude some popular solutions wherein processes are dispatched whenever a processor might be (wished) available and often on processes that are so simple as to be trivial. For an algorithm to be useful under these restrictions, it must admit isolation of *substantial* subprocesses, sufficiently high in the computation tree (of the the chosen algorithm) in order to assure that the $p$ processors can be loaded using as few process dispatches as possible while balancing the load and avoiding duplication of effort. To do that requires identification of independent, arbitrarily large processes as high in that tree as possible—particularly for the cases where $p$ is indeed very small.

What is really needed first is an algorithm, presenting such a tree with the suggested decomposition properties. This paper presents such an algorithm, matrix inversion via classic Gaussian elimination, over a new data structure—the quadtree representation for matrices—in a purely applicative style [3, 5]. This formulation lends itself to satisfying the restrictions set forth above, regardless of the particular values of $n$ and $p$.

Applicative programs are necessarily presented as expressions without assignment statements or control statements except for (pure) function application. Such programs implicitly solve the problem of decomposition into independent processes [4] because each subexpression within the program is necessarily independent; therefore, the syntax tree may be strongly associated with the tree for process decomposition. Intermediate binding still allows intermediate results to be shared—rather than recomputed. While these matrix results are interesting and useful as they stand, an implicit goal of this paper is to encourage further study of algorithms for parallel computation through the philosophy enforced by applicative (or functional) programming style.

Gaussian elimination is hardly new, so what can be said that is really novel? Certainly, no improvement to well-studied asymptotic behavior will be offered. Three results, however, are offered that, together, promise a thoroughly practical algorithm under the envisioned constraints, whether or not their ultimate realization follows the discipline of applicative programming.

First, the relatively new idea of quadtrees for representing matrices [10, 11] is developed further, in a way that unifies our approach both to matrix/scalar algebra and to sparse/dense matrix manipulation. All scalars, $x$, also represent diagonal matrices, blurring the distinction between scalar and matrix, between sparse representation and dense representation. We shall see that one family of algorithms handles all pathologies.

Second is a version of matrix-inversion via Gaussian Elimination, through Pivot Step [6] with the pivot element selected as the largest candidate (in magnitude) from the entire matrix before each pivoting. Known to be most stable, this algorithm is also shown to present a pattern of parallelism. Each pivot step naturally decomposes by quadrants which, most interestingly, is a pattern suitable for the search tree for identifying the next (largest in magnitude) pivot candidate. There is, therefore, no question whether a *search* for the next pivot element should be implemented using parallelism or on a cheaply-dispatched uniprocessor.

0190-3918/86/0000/0092 $01.00 © 1986 IEEE

While such a search surely could use parallel processing, eliminating the explicit search phase by folding it into each (sparse) pivot saves search time, and better amortizes the overhead to dispatch each pivoting process.

Finally, an interesting relationship between padding and processor allocation is proposed to balance the load across independent processes. The quadtree matrix representation appears to be suited only to representation of $2^m \times 2^m$ matrices. When the size of a matrix is not a power of two, some padding is necessary which only wastes space proportional to $m$. What is interesting is that, having embedded an $n \times n$ matrix in (the lower right of) a $2^m \times 2^m$ one, processor allocation can make profitable use of the value of $(2^m - n)$ in partitioning the $p$ processors among subproblems. The argument is cast in terms of familiar matrix operations.

The remainder of this paper is in four parts. The quadtree representation is introduced first, including discussions of its restriction to vectors and generalizations to higher dimensions. The second section explores a Gaussian elimination algorithm under the new parallelism, as outlined above. The third offers a strategy for allocation of processor resources to discount the padding that might be necessary to fill out the quadtree representation. The final section offers some conclusions and hopes for further work.

### Section 2. Matrices

Let any $d$-dimensional array be represented as a $2^d$-ary tree. Here we consider only matrices and vectors, where $d = 2$ suggests quadtrees, and $d = 1$ suggests binary trees.

Matrix algorithms will be arranged so that we may (without loss) perceive any scalar, $x$, as a diagonal matrix of arbitrary size, entirely of zeroes except for $x$'s on the main diagonal; that is, $x = [x\delta_{i,j}]$. Thus, a domain is postulated that coalesces scalars and matrices, with every scalar-like object conforming also as a matrix of any size. Of particular interest are 0 and 1, which are at once the *unique* additive and multiplicative identities, respectively, for scalar/matrix arithmetic. Similarly, the scalar $x$ as a binary tree is interpreted as a vector of arbitrary length, each of whose components is $x$ (much like Daisy's [7] notation $\langle x^* \rangle$.) Inferring the conventional meaning from such a matrix now requires additional information (*viz.* its size), but we can proceed quite far without size information; it only becomes critical upon Input or Output.

Lest it appear that this coalescing of hitherto disjoint types hides too much, it is useful to draw an analogy from ordinary computation on floating-point numbers (FPNs), where details of internal representation are also suppressed. The point is that the way that quadtree-matrices (and FPNs) are commonly represented outside the machine has little to do with their internal representation. There are, in fact, conventional styles for writing matrices on paper—in row-major order (and for writing FPNs in scientific notation), but these may differ wildly from the way that they (and FPNs) are to be represented within the machine. Although the algorithms for translating between such representations are elegant, they are so complicated that they are surely not the first thing that should be shown to those unfamiliar with the nature of these internal representations. Thus, an unfortunate barrier rises in the path before those who would tinker on them: one must first write the I/O translators, which are among the least comprehensible, least efficient, and least exercised programs over the structure.

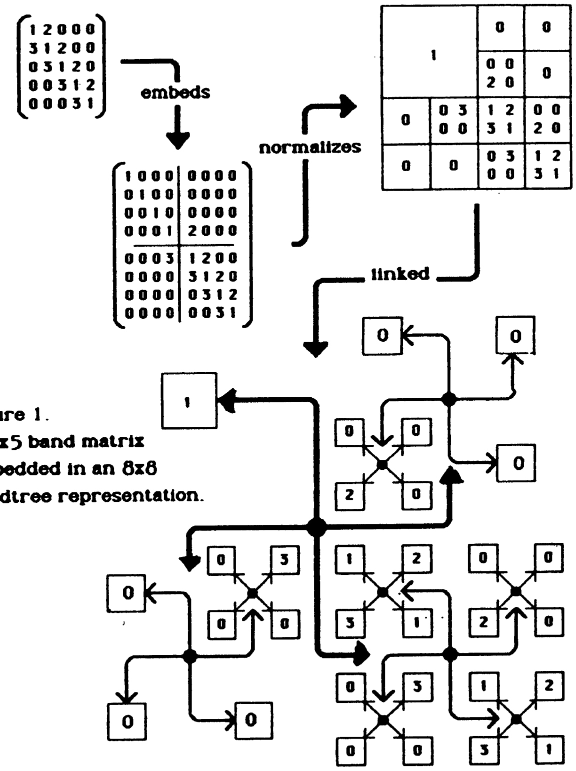


Figure 1. A 5x5 band matrix embedded in an 8x8 quadtree representation.

A matrix (of otherwise-known size) is either a vague 'scalar' or it is a quadruple of four equally-sized submatrices. So that this recursive cleaving works smoothly, we embed a matrix of size $n \times n$ in a $2^{\lceil \lg n \rceil} \times 2^{\lceil \lg n \rceil}$ matrix, justified at the lower, right (southeast) corner with zero padding to the north and west, except for nonzeroes—preferably ones—padded along the northwest diagonal (to avoid introducing an unnecessary singularity; see Figure 1.) The matrix is justified to the southeast, rather than the northwest, so that its eliminant [2] is properly defined.

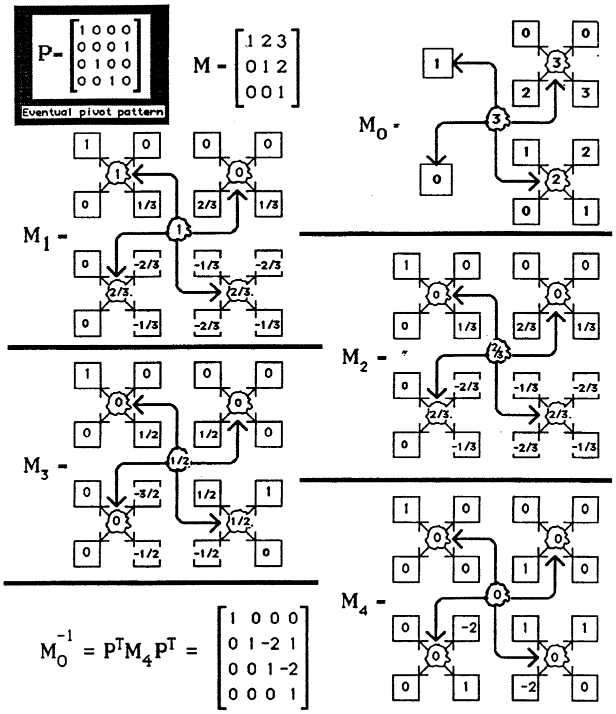
There is also a *normal form* convention. Under this quad representation, no submatrix will ever be composed of four 'scalar' quadrants, of whom the northeast and southwest are zero, and whose northwest and southeast coincide. Such a matrix would be represented, instead, by the latter 'scalar,' standing alone. Thus, the two important identity/annihilator matrices are represented *uniquely* by 1 and 0. If we require that the northwest padding, as in the previous paragraph, is necessarily one, then a *canonical form* results.

Elsewhere [10] I observed how algorithms for matrix addition, transpose, and multiplication follow the desirable pattern of decomposition, generally into $4^m$ or $8^m$ independent processes that can be dispatched high in the computation tree, up to the capacity of the execution environment. Of note is the role of 1 and, especially, of 0 as a constituent quadrant to such an operation. When any addend's quadrant is 0, the effort for matrix addition immediately simplifies by 25% because it is, therefore, unnecessary to descend and to traverse the corresponding quadrant of the other addend; all we need is a borrowed reference to it as one quadrant of the result. A factor's quad of 1 reduces Strassen's decomposition [9] of Gaussian multiplication from eight recursive multiplications to six; not only does a 0 quad similarly annihilate two recursive multiplications, but also it avoids two of the four subsequent additions, as well.

These properties are particularly valuable for matrices with regular patterns of non-zero entries, especially those that are sparse or in diagonal form. No special code is necessary to accelerate conventional operations on them (but one can wish for hardware that accelerates specialized tests like the ubiquitous tests for 0 submatrices.) It is, however, necessary to maintain the normal form so that, like rational numbers being always "reduced to lowest terms", matrices are reduced to their corresponding scalars whenever zero southwest\northeast quadrants and coincident northwest/southeast quadrants permit.

Another advantage shows up upon deeper study of several matrix manipulation programs over quadtrees: algorithms written to accept canonical-form operands need not be sensitive to the usual compatibility requirements. That is, ordinary quadtree algorithms for various operations will work regardless of the depth of their tree/operands; when operands are of different depth, the shorter paths—ending at a scalar—will be interpreted as all-conforming, diagonal matrices. Although their meaning may be questionable, the algorithms will run to completion instead of crashing with array-bounds-violations. In terms of domain theory, we have raised these operations to a higher point in their respective function-domain (given them more meaning by defining results where others fail because of incompatibility), while simplifying the code (by eliminating all the incompatibility tests and signaling thereof).

A "header" above each matrix quadtree might usefully contain two values needed for output translation: the length of the diagonal padding, and the exponent, $m$, for a $2^m \times 2^m$ matrix. The value of $m$ also suffices for runtime compatibility checking. Another bit there indicates whether the quadtree is to be interpreted as transposed, recursively interchanging southwest\northeast quadrants upon any access. Thus, not only does quadtree representation allow us to transpose an entire matrix in constant time—at the cost of building a new header—but also it allows row and column traversal at equally high efficiency, at the cost of symmetric-order traversal [6] of the appropriately projected binary tree.



P = [1 0 0 0; 0 0 0 1; 0 1 0 0; 0 0 1 0]
Eventual pivot pattern
M = [1 2 3; 0 1 2; 0 0 1]
M0 =
M1 =
M2 =
M3 =
M4 =
$M_0^{-1} = P^T M_4 P^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

### Section 3. Pivot Step and Inversion

This section describes an algorithm for matrix inversion, extended from Knuth's [6] for an entirely different data structure (also sparse). His terminology is used because it is readily available. Figure 2 outlines this algorithm applied to his problem, and should be compared against the solution that he provides. The algorithm is Gaussian elimination with full pivoting (although Knuth only selects pivots rowwise.) Much of the description applies to a single Pivot Step, but its most interesting property relates one such step to the next. It is easily simplified to one that only finds the root of the linear equation, $Ax = y$, for Matrix $A$ and Vector $y$.

Let a non-singular matrix, $M$, be represented with a quadtree as described in the previous section. Each nonterminal node, however, is to be decorated with additional information: the magnitude of the largest element in that quadrant, and its local horizontal and vertical coordinates. (It will be necessary to qualify the decoration further to exclude any element in an already-pivoted row or column from decorations. At this point, however, assume that no pivoting has yet occurred.) These coordinates need not be traditional indices—though it is easy to think of them that way. A cheaper implementation is just a pair of bits at each node selecting one subtree; the catenation of these subtrees identifies a path through each quadtree to the appropriately largest scalar.

Let us consider an algorithm to invert $M$, a matrix represented as a canonical-form quadtree, padded along its northwest diagonal as described in the previous section. The first step in computing the inverse of $M$, therefore, is to traverse its tree representation in postorder, installing these decorations at all internal nodes. (Sibling subtrees, of course, may be traversed in parallel.) Call the decorated matrix $M_0$. Decorations appear in florets in Figure 2; to save space, however, only the local maxima (no local coordinates) are shown.

Also needed are two trivial binary trees, each initially the scalar 0 indicating a boolean vector of all zeroes, and one trivial quadtree, $P_0$, also initially 0, similarly. For each $i$, Quadtree $P_i$, indicates the exact position of the first $i$ pivot elements—none so far; it will be filled in to become a permutation matrix, $P_{2^m}$, by the time the fully pivoted matrix, $M_{2^m}$, is computed. The boolean vectors indicate which rows/columns have already been pivoted; they will be filled with 1's until they both indicate that all rows/columns have been eliminated. The two vectors, therefore, are merely row (column) projections from the corresponding $P_i$, but in a format useful for directing subsequent decorations.

Having established the initial values of $M_i$ and $P_i$, for the next of $2^m$ pivot steps, we discover that the decoration at the root of this tree identifies the next pivot element. We presume here that $M_i$ is not a scalar. If it were (and thereby identified as the pivot scalar), then its reciprocal is the pivoted matrix.

Otherwise, the $M_i$ may be decomposed into quadrants, each distinguished by the decoration. One is *pivotquad,* distinguished because the pivot element lies within. Another is *rowquad,* named because it lies horizontally from *pivotquad.* The third is *columnquad,* because it lies above or below *pivotquad.* The last is *offquad,* so called because it does not coordinate on *pivotquad,* but lies diagonally from it.

The description that follows discusses the transformations on the four quadrants in reverse order from that just above. It turns out that *offquad*'s transformation is simplest, but it does depend on intermediate results derived from processing the other three quadrants. So many additional results are needed from handling *pivotquad,* moreover, that its description will be considerably eased by working backwards in order to justify them.

An important property of functions is that one invocation may return several results of differing types. This feature has been lost in many programming languages (perhaps because their designs presume that a computer has but one accumulator,) but it is critical to applicative style and allows one function invocation to return all these intermediate results.

Knuth's transformation of the matrix,

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & a & \cdots & b & \cdots \\ & \vdots & & \vdots & \\ \cdots & c & \cdots & d & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

where $a$ is the pivot element, $b$ is any element in the pivot row, $c$ is any element in the pivot column, and $d$ is any off pivot element coordinating on $b$ and $c$, is

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & \frac{1}{a} & \cdots & \frac{b}{a} & \cdots \\ & \vdots & & \vdots & \\ \cdots & \frac{-c}{a} & \cdots & d-\frac{bc}{a} & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

respectively. This is the transformation to be made, though it is to be done here recursively and (likely) in parallel.

The transformation of *offquad* requires the values of $d$ contained therein, and two vectors of values $b/a$ and $c$ coordinating on each $d$, occurring in the pivot row and column, respectively. Since these two vectors, represented as binary trees in normal form, occur in *rowquad* and *columnquad* they will have been extracted as intermediate results while processing those quadrants. If either of these vectors is zero, however, then the transformation collapses to the identity function; all values of $bc/a = 0$ and not even the internal decorations in *offquad* change, as neither the newly eliminated pivot row nor pivot column cross it. (This is *the* savings of sparse representation for Pivot Step.)

When *offquad* is a scalar, $d$—even if it is a normalized representation of a larger matrix (notably if $d = 0$) and $b/a$ and $c$ are vectors—then the correct transformation is the decorated form of $d$'s difference with their outer product, $d - bc/a$. In order to decorate the difference, the appropriate halves of the boolean vectors identifying rows and columns eliminated from $M_i$ are necessary parameters. (If $d$ is scalar and these vectors are non-trivial, then $d$ must be expanded into $\begin{pmatrix} d & 0 \\ 0 & d \end{pmatrix}$ and decomposed, as below.) When $d - bc/a$ is a scalar, either zero or its absolute value becomes its decoration, depending on whether it lies in an already pivoted row/column, or not.

When *offquad* is decomposed into four quadrants, then each of these is treated as an *offquad,* with the vector parameters cleaved in half to provide the four sets of vector arguments, and the four results decorated and normalized into the transformed quadtree.

The treatment of *rowquad* and *culumnquad* are similar, so only that of the former is presented here; the latter's is nearly dual to what follows. The treatment of *rowquad* requires the inverse of the pivot element, $a$, and a vector that is the portion of the pivot column that lies in *pivotquad*, but for the pivot element, itself. It is sufficient to represent both as a copy of the pivot-column vector with $1/a$ in place of the pivot element (which will have been extracted during the treatment of *pivotquad*). Also needed are a relative index locating the pivot row and halves from the boolean vectors indicating which rows/columns crossing *rowquad* have already been eliminated.

Results are the transformed *rowquad* and half of the pivot row (as a binary tree) extracted from *rowquad* for use in handling *offquad*. That vector is also needed for handling most of *rowquad*, itself because, unless *rowquad* is trivial, it must be decomposed into two sub*rowquad*s and two sub*offquad*s, the latter of which coordinates on that half of the pivot row.

Thus, the transformation of *rowquad* focuses first on the pivot row. When the row index indicates that *rowquad*–call it $b$— is entirely within the pivot row, then the residue column vector is just $1/a$, and the needed results (matrix and row-vector) are each the product, $b/a$. Decorated as a matrix, the local maximum is 0 because this row is being eliminated.

If *rowquad* is to be decomposed into submatrices, then the local index will indicate whether the pivot row crosses the upper or lower half. Accordingly, the pivot column and boolean vectors (identifying already eliminated rows/columns) are split, and those two containing the pivot row are treated (as *rowquads*) first. The pieces of the pivot row, extracted thereby, are joined at a binary node to become the vector-result of treating *rowquad, and* are passed as arguments with the other two quadrants for treatment as *offquad*s. Then these four quadrants are assembled and decorated into the matrix-result of this treatment.

Finally, we consider the treatment of *pivotquad*, upon which all the other three quadrants' treatments depend; it must occur first and yield as a partial result the transformed, decorated version of *pivotquad*; and as intermediate results: the row and column indices of the pivot element, and vector copies of the pivot column (with $1/a$ in place of the pivot element) and pivot row (with $-1/a$ in place of the pivot element). Moreover, while locating the pivot element, updated versions for $P_i$, the permutation matrix (which will only change in the quadrant corresponding to *pivotquad*), and for its projections, the two boolean vectors of eliminated rows/columns should be constructed. That's eight results, four of which are intermediate—not to be included directly in an answer, but to be used in treating sibling quadrants.

Three observations complete this description. First, the treatment of *pivotquad* is the same as the treatment of the whole matrix, $M_i$; the only difference is the unneeded, extra four results, beyond $M_{i+1}$, $P_{i+1}$, and the two boolean vectors projected therefrom. Secondly, the arguments to each Pivot Step (and pivoting successive *pivotquads*) are $M_i$, $P_i$, and the two boolean projections from $P_i$ (and the corresponding quadrants/halves therefrom.) The last three arguments are used as seeds for updated results, and the last two also help to place 0 decorations.

Finally, if $M_i$ (correspondingly, *pivotquad*) is a scalar, $a$, then all eight results are trivial: $M_{i+1} = 1/a$ and is decorated as 0 (now that both it's row and column have been eliminated); $P_{i+1} = 1$ and both its projections are 1, also; the pivot column is $1/a$, and the pivot row is $-1/a$; and both relative indices are 1.

When $M_i$ is not scalar, it decomposes into four quadrants, one of each type considered above. Algorithms for three of them (*rowquad, columnquad, offquad*) have been discussed above, and the fourth (*pivotquad*) is to be treated recursive as a Pivot Step, with basis stated the preceding paragraph.

The **parallelism** in this algorithm manifests itself in the interdependence of these recursive decompositions. For instance, treatment of successive candidates for *pivotquad* must precede transformation of all other quadrants; the depth of this recursion is at most $m$. After each is completed, its associated *rowquad* and *columnquad* may be dispatched simultaneously, and, of these, half must be transformed before the other half. Again, the depth of recursion is at most $m$. Thereafter, however, all *offquads* at all levels of the quadtree may be treated *simultaneously*. They generate most of the effort in a Pivot Step, but and thier transformations are mutually independent.

It should be clear that a pivot step can change lots of decorations from those in $M_i$; is there, then, sufficient information to restore them? Yes, because decorations will only change where scalar values have changed, or because a local maximum is disqualified because it resides in either the current pivot row or current pivot column. Such decorations have already been visited by this algorithm! (This point is most important when inverting sparse matrices, where little traversal is necessary.) We need only arrange that, as each interior node in the quadtree (that becomes the pivoted matrix) is reassembled, it must be redecorated with the appropriate maximum and local coordinates. Therefore, the position of each scalar encountered is resolved against the boolean vector indicating already-eliminated rows and columns; if it is to be excluded, treat its magnitude as zero *for the purposes of finding the maximum local magnitude.* If all four local maxima are zero, then the subtree is decorated with zero (and the local coordinates may be left undefined.)

As the four new quadrants are reassembled, it is necessary to find the maximum of their decorations and its two-bit coordinates, according to which of the four quadrants it came from. Internal zero decorations do not propagate, because some decoration must be positive if the original matrix was non-singular.

That completes the description of a single pivot step, $M_i \mapsto M_{i+1}$. It only remains to observe that the results of one step, including the pivoted, decorated matrix, the two vectors (binary trees) of eliminated rows and columns, and the building permutation matrix are passed from one step directly along to the next. *There is no need to search for the next pivot element,* because it has already been located. Moreover, it has been located by parallel processes already dispatched for the pivot step, itself, in parallel. Thus, there is no dispatch/recovery overhead for the parallel search!

Finally, observe that the desired inverse, $M^{-1}$ is readily available after permutations:

$$M^{-1} = P_{2^m}^T \times M_{2^m} \times P_{2^m}^T. \cdot$$

In fact, the code in the appendix builds up $P_i^T$, rather than $P_i$, anticipating this transpose..

This entire algorithm proceeds on non-singular matrices without any counters; even the outer control over of $2^m$ successive pivot steps may be set up as a loop until decoration becomes zero. In some sense, then, it is more abstract, and more useful, than algorithms that depend heavily on size declarations and bounds tests.

### Section 4. Subprocess Balancing

Suppose an $n \times n$ matrix is embedded in a $2^m \times 2^m$ matrix, where $k = 2^m - n$ and $0 \leq k < 2^{m-1}$ as in Figure 3. Section 2 suggested that the values of $k$ and $m$ might need to be available to the system at run-time, even though they remain unnecessary to (in particular) the abstract multiplicative operators. This section proposes another use for this same information: load balancing among the processors.

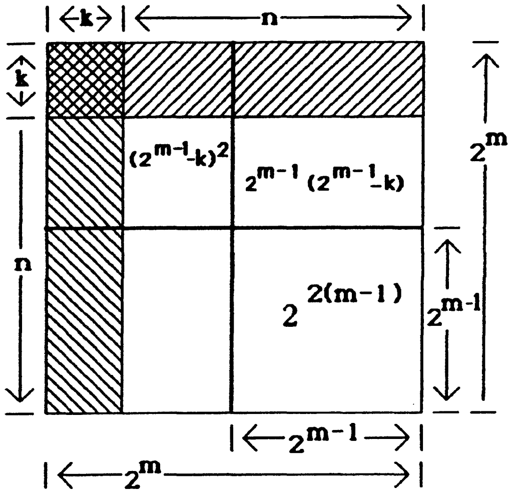


Figure 3. Areas within quadrants, excluding padding of k rows/columns.

The following discussion does not depend on denseness or sparseness of matrices. It does presume that the distribution of non-zero entries is uniform; if more patterns are known, then further inferences might be possible. The values of $m$ and $k$ indicate the size of a matrix and what proportion of it is trivial. From them we can determine what portion of each of the four quadrants is serious, *i.e.* likely to cause serious effort for the processors, and we can use this information to distribute the $p$ processors among the four quadrants.

Consider matrix addition, for instance. The quad recursion pattern is simple; each quadrant requires addition effort in proportion to its "serious" area (Figure 3). The serious area of such a matrix is $(2^m - k)^2$. It is divided up among its four quadrants in the following proportions:

| Quadrant | Relative share of area |
|---|---|
| Northwest | $\frac{2^{m-1} - k)^2}{(2^m - k)^2}$ |
| Northeast | $\frac{2^{m-1}(2^{m-1} - k)}{(2^m - k)^2}$ |
| Southwest | $\frac{2^{m-1}(2^{m-1} - k)^2}{(2^m - k)^2}$ |
| Southeast | $\frac{2^{2(m-1)}}{(2^m - k)^2}.$ |

It is unlikely that these proportions have integer products with $p$. If only a fraction of a processor is available to a quadrant, a good solution is to combine quadrants on shared processors in a way that the individual requests for a processor sum to an integer. A likely grouping is to set the northwest, northeast, and southwest sums (the partial quadrants) on one processor, and to set the southeast sum alone on another.

Therefore, if one had $p$ processors to add such matrices, one could use this information to distribute them to four quadrant process in these proportions in a top-down pattern. As mentioned in the introduction, it is important that processor allocation be done as high in the data structure/computation tree as possible, so that the overhead of process dispatch/recovery not be paid repeatedly.

In this way it is likely that larger quadrants (southeast) gets more processors. When a quadrant receives but one processor to compute its result, it operates as a uniprocessor; if it receives more than one, it can apply the same idea to divide up its processor resource once again, and so forth. The first three quadrants make the most interesting further processor allocations; the southeast quadrant is presumed to be uniform and so its share will just be divided in even fourths.

These same proportions could apply to the Pivot Step algorithm above. Unlike addition, however, we saw that there was some serial behavior to Pivot Step. Thus, (using terminology from before) *rowquad* and *columnquad* may be dispatched together, but they are only two of four quadrants. Nevertheless, their computational effort may also be approximated by the relative size of two areas (diagonal from each other), in the table above. Although we may not know which processors until run time, we may select the proper proportion then and split the processor resources in that manner.

Gaussian matrix multiplication (under Strassen's formulation [9]) easily decomposes into eight products, which are pairwise summed to form a four-quadrant answer. Excluding the effect of scalars (*i.e.* a quadrant of 1 or 0 avoids a quadrant multiplication) and asserting that the $O(n^3)$ algorithm does require a processor resource proportional to $(2^m - k)^3$, we determine the proportion of this resource that each of the eight products needs. This was done by setting two matrices, as in Figure 3, as multipliers and extrapolating the effort to multiply each of eight sub-products from the areas of the sub-multipliers (quadrants.)

The eight ratios are best described by combining them pairwise, as their associated products are to be added, to yield the proportion of effort invested in building each of the four sums that become the matrix product. The four ratios coincide with those already derived above! Furthermore, proportions associated with each of the eight products may be obtained by multiplying each of these four ratios by $2^{m-1}/(2^m - k)$ and by $(2^{m-1} - k)/(2^m - k)$, respectively. Thus, the four-quadrant ratios are exactly as before, and the eight-quadrant ratios are uniform extensions from these.

Such process allocation could be determined statically at compile time [8] when the language requires matrices to be of declared, constant size and uniform sparseness. The algorithms proposed here do not require bounds declarations, but if they were available, it would be possible to avoid much communication with, and system saturation of, a dynamic scheduler.

## Section 5. Conclusions

Because we assumed that the number of processors, $p$, is small compared to the size of the matrix, $n$, we need to cleave matrix manipulations into a *few* subprocesses of *balanced* size, so that the resource $p$ can be allocated deliberately. If one cleaving does not consume all the processes, the pieces may be further split. Avoiding repeated dispatch and recovery, these algorithms have the virtue of splitting at the base of the quadtree—at the root of the problem.

Although these algorithms are described as if only scalars could be leaves of the quadtree, that arrangement is not necessary. It is perfectly possible that sizable matrices dwell at the leaves, matrices that might be represented in traditional row-major order, and manipulated using traditional iterative and pipelining algorithms, programmed in FORTRAN *et fils*. (Pipelines would need an extra input stream of bits, identifying candidates for updated local maximum, and each could then compute updated vector decorations during a vector update.) The size of such leaf matrices should be chosen to balance the efficiencies of existing style and existing machinery against the obvious need for multiprocessing techniques to accelerate large matrix computation.

Derived from a purely applicative approach, they better suit a computing environment with many processors, with memory banked at varying distances from the different processors, and with contention for access to shared resources as a real constraint on efficiency. The quadtree approach shows us how to represent matrices so that useful pieces may be localized where some processor can make computational headway on the problem without excessive interference from all its brethren. It has long been known [4] that applicative style solves the problem of decomposing an extant algorithm; the problem addressed here is the discovery of good algorithms that cleave into usefully sized pieces.

One last issue raised by this work relates to all sparse matrix techniques. I have seen only vague definitions of a what makes matrices *sparse*, possibly because sparse representations are so different from one–another, and maybe because sparseness has been purely an operational concept. Quadtrees, as we have seen, offer a reasonable representation for sparse matrices that is consistent with what we would use for dense matrices; based on that observation, *alone*, I suggest a **measure of sparseness** for matrices: the ratio between its average path length in its quadtree representation (from root to leaf) and the logarithm of its size. Ratios closer to zero indicate sparser matrices..

· Quadtree representation of matrices was motivated by studies in applicative programming and as part of an effort to study its impact on Matrix Algebra for MIMD multiprocessors. The results of the effort are more than satisfactory: not only does the technique apply to a well-worked problem, but also it yields new insight (through a distribution of searching across Pivot Step) on optimal hardware/software solutions. Thus, there we have more support for using applicative programming (and its algebra) for programming parallel architectures and a suggestion that those architectures provide a multi-banked heap.

There is already interest using quadtrees as *the* uniform representation for matrices in a large computer algebra system [1], a sophisticated piece of software running on fairly conventional hardware. They also have an important role to play within conventional languages used on very sophisticated hardware, an experiment that remains to be done.

**Acknowledgement:** Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 84-05241. I would like to thank John Lash and Kamal Abdali for ideas and encouragement that helped these ideas mature.

### References

1. S. K. Abdali. Personal communication (1985).

2. S. K. Abdali. & D. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* **40**, 2,3 (1985), 257-274.

3. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21**, 8 (August, 1978), 613-641.

4. D. P. Friedman & D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4 (April, 1978), 289-296. Preliminary version appeared as: The impact of applicative programming on multiprocessing. *Proc. 1976 International Conference on Parallel Processing,* 263-272.

5. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations,* M.I.T. Press, Cambridge, MA (1984).

6. D. E. Knuth. *The Art of Computer Programming,* **I**, *Fundamental Algorithms,* 2nd Ed., Addison-Wesley, Reading, MA (1975), 299-318 + 401, 556.

7. A.T. Kohlstaedt. Daisy 1.0 Reference Manual. Tech. Rept. 119, Computer Science Dept., Indiana University (November, 1981).

8. V. Sarkar & J. Hennessy. Compile-time partitioning and scheduling of parallel programs. it Proc. SIGPLAN 86 Symp. on Compiler Construction, SIGPLAN Notices **21**, to appear.

9. V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13**, 4 (August 19, 1969), 354-356.

10. D. S. Wise. Representing matrices as quadtrees for parallel processors (extended abstract). *ACM SIGSAM Bulletin* **18**, 3 (August, 1984), 24-25.

11. D. S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters* **20** (May, 1985), 195-199.

12. M. F. Young. A functional language and modular arithmetic for scientific computing. In Jean-Pierre Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 305-318.

### Appendix

The following examples, all of which evaluate to 1, are useful as an introduction to the Daisy [7] code that follows. They exemplify a new style of applicative programming that depends on functional combination and *data recursion* [5] to specify multiple and interdependent results without cognizance of a necessary sequence of evaluation. All primitives used here, however, have been in Daisy from its birth. The forms

```
let:[  identifierStruc  binding  result]
rec:[  identifierStruc  binding  result]
```

return *result* computed in an environment enhanced with *identifierStruc* bound to *binding*. Evaluation is lazy, and the list structure of *binding* must match the structure of *identifierStruc*, wherein bound identifiers become bound according to their position within that list. Functional combination [4] is indicated by a list structure as a function, to the left of the colon, the "apply" operation. In the layout below, one may perceive that the constituent components of that combination is applied vertically to the (transposed) argument matrix. Thus, the intermediate results of all the functional combinations below is that of ⟨10⟨3 1⟩⟩.

```
let:[   [sum    [quotient          remainder]]
         <add    <div               rem      >>:<
         <6      <        10        *        >>
         <4      <        3         *        >> >
    remainder]

divide = ^λ[a b].<div:<a b> rem:<a b>>
let:[   [sum    [quotient          remainder]]
         <add              divide            >:<
         <6                10                >
         <4                3                 > >
    remainder]

rec:[   [sum    [quotient          remainder]]
         <add              divide            >:<
         <6                sum               >
         <4                3                 > >
    remainder]
```

The skeleton of the Daisy code for Pivot Step follows. It presumes that quadtree-matrix arguments/results are decorated at non-terminal nodes. This code has been cut back to remove all provision for normalized (sparse matrix) representation and for permuting the quadrants. Normalized matrices may require expansion of scalar arguments into a quadruple (two of which are zero; two of which are the scalar) upon function entry. and collapse of such quadruple patterns to the scalar upon exit. Without provision for permuting, this code will pivot only on the northwest-most (upper left) scalar entry; the expanded code tests two bits of "decoration" (*ibit* and *jbit*), and provides permutations on arguments and results to translate to/from this northwest orientation.

Notice that the instances of functional combination in *PIVOT*, *ROW*, *COLUMN*, and *OFF* are

$$[PIVOT\ ROW\ COLUMN\ OFF],$$

$$[ROW\ OFF\ ROW\ OFF],$$

$$[COLUMN\ OFF\ COLUMN\ OFF],$$

$$[OFF\ OFF\ OFF\ OFF],$$

respectively, reflecting the recursive decomposition of each kind of quadrant. It is also important that *OFF* immediately tests whether either vector argument (the projection from the pivot row or pivot column) is zero, and acts as an identity function in that instance. Also, this code builds the transpose of the permutation matrix ($P_i^T$ from the paper) directly.

```
PIVOT = ^λ[decoratedMtx elimrow-col PermutT]. if:<
        Scalar?:decoratedMtx    let:[ inverse  reciprocal:decoratedMtx
            < inverse [TRUE TRUE] 1 <negate:inverse inverse> [1 1]> ]
        let:[  [[max ibit jbit] ! mtx]
                decoratedMtx
        let:[[ [epivot [] [] [eright ebot]]     [permutHEAD ! permutTAIL] ]
             <      spread4:elimrow-col                  PermutT          >
        rec:[
            [   [i [eleft etop] permutPIVOT [pleft ptop] [ipos jpos]     ]
                                [ii pright]      [iii pbot]      iv      ]
                <PIVOT          ROW              COLUMN          OFF>:   <
                                        mtx
                <epivot         <eright etop>    <eleft ebot>    <eright ebot>>
                <permutHEAD     ptop             pleft           <pright pbot>>
                <[]             ipos             jpos            []  >   >
        let:[   [elimrow-col pivotrow-col]
                <<<eleft eright> <etop ebot>>   <<pleft pright> <ptop pbot>> >
        <decorate:<elimrow-col  <i ii iii iv> >
            elimrow-col         <permutPIVOT ! permutTAIL>
            pivotrow-col        <twice:ipos twice:jpos> >       ]]]]>


ROW = ^λ[decoratedMtx elimrow-col pivcol index]. if:<
        one?:index let:[ BoverA
                         decorateproduct:<elimrow-col pivcol decoratedMtx>
                         <BoverA BoverA> ]
        let:[   [iresidue ibit]           divide:<index 2>
            rec:[ [ [i left]    [ii right]      iii         iv          ]
                    <ROW         ROW             OFF         OFF         >:<
                            tail:decoratedMtx
                            spread4:elimrow-col
                                                      let:[ [top bot] pivcol
                    <top         top             <bot left> <bot right> >]
                    <iresidue    iresidue        iresidue    iresidue    > >
                < decorate:<elimrow-col <i ii iii iv>>  <left right> > ]]>

COLUMN = ^λ[decoratedMtx elimrow-col prow index]. if:<
        one?:index
            <decorateproduct:<elimrow-col prow decoratedMtx> decoratedMtx>
        let:[   [jresidue jbit]          divide:<index 2>
            rec:[ [ [i top]     ii           [iii bot]   iv          ]
                    <COLUMN      OFF          COLUMN      OFF         >:   <
                            tail:decoratedMtx
                            spread4:elimrow-col
                                                      let:[[left right] prow
                    <left        <top right> left        <bot right>>]
                    <jresidue    jresidue    jresidue    jresidue    >    >
            < decorate:<elimrow-col <i ii iii iv>>       <top  bot   > > ]]>

OFF = ^λ[decoratedMtx elimrow-col prow-col index]. if:<
        anyzero?:prow-col        decoratedMtx
        one?:index decoratedifferencE: <elimrow-col decoratedMtx
                                   decorateproduct:<elimrow-col prow-col> >
        decorate:<elimrow-col
                <OFF         OFF          OFF          OFF         >:    <
                        tail:decoratedMtx
                        spread4:elimrow-col
                        spread4:prow-col
                <half:index half:index  half:index  half:index>     > >>
```

# SK-Banyans: A Unified Class of Banyan Networks

Julio de Melo
Roy M. Jenevein
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712

## Abstract

A unified class of banyan networks is proposed which presents better distance properties than SW-banyans. This new class is called SK-banyans for the *SK*ewing scheme that is adopted for the connections between nodes at different levels. Its definition is given along with the characteristic properties relative to SW-banyans. A consideration is given for the virtue of symetric connection schemes in relation to their distance properties. Examples of different Characteristic Connection Patterns (CCP's) for SK-banyans are presented. A general construction scheme is proposed and a comparison of distance properties of SK- and SW-banyans is presented. Optimality criteria have been developed to determine whether a given CCP provides the best distance properties.

## 1 Introduction

Interconnection networks have become an important component of parallel or distributed computer systems, and as such have stimulated much interest in recent years [16]. Topics such as topologies, routing, data manipulation, fault-tolerance and VLSI design, to name but a few, have by themselves spawned new research areas where so much is yet to be done.

On the issue of topologies, the topic to which this paper is related, new structures have been proposed recently [6,10,16], as well as old structures have been subject of renewed studies [11].

Banyan networks [5,14] constitute one such class of the many proposed topologies, and reports on their fault- tolerance properties, resource allocation algorithms and performance evaluation can be found in the literature [3,4,7,8,12,15].

It was with much hesitation that we considered putting forward a paper on "yet another network topology". The reason, however, for doing so is twofold. First, the SK-banyan represents a unification and not a fragmentation of the networks issue, because the SW-banyan (and all of its isomorphic counterparts) is a member of the class of SK-banyans. Second, significant understanding of the issue of network connectivity and distance properties have been attained by the study of SK-banyans.

Our research focuses on the evaluation of the distance properties of a specific class of banyans, motivated by results reported in [13] regarding the distance properties of KYKLOS multiple-tree networks. These results showed that, by changing the connections of the second tree in an appropriate and regular way, the distance properties of this so modified double-tree were shown to be better than for the conventional double-tree. This represented a break from traditional symetric interconnection schemes of the past which have lead to symetric redundancy. The connections, while a break from the past trends, are regular and predictable from stage to stage. As the banyan networks have an embedded tree structure, they might also show some improvement by using this modified connection scheme. It must also be pointed out that the modified connection still provide a tree topology between the apex and base of the structure.

The emphasis on the study of the distance properties of this new banyan network is justified by the fact that the delay observed in the transmission of messages across an interconnection network is closely related to their distance properties [1]. Since the apex-distance is always $log_2 N$, the distance referred to here is from any node to any other node. This is particularly useful when studying single-sided networks, because base-to-base distance becomes quite important. Also, the fault-tolerance properties of the ICN may show some improvements as well, due to the existence of alternative routes. This is true for both KYKLOS and SK-banyans where better distance properties are obtained by avoiding redundant connections between sets of nodes at different levels.

## 2 Banyan Networks

A *banyan* network is defined in terms of a directed graph representation, which is a Hasse diagram of partial ordering with a unique path between every base and apex. A *base* is defined as a node of out- degree 0 and an *apex* is defined as a node with in-degree 0. In this paper, we assume the convention that levels are enumerated from apex to base, with level 0 corresponding to the apex. Two examples of banyan networks are shown in figure 1. The left network is irregular and is restricted to special applications, such as mapping an algorithm into a network. The right network, being regular, is very appropriate for use in interconnection networks, both because data routing algorithms can be easily specified, and because of its good properties for data manipulation, due to its embedded tree structure. A regular banyan is characterized by the fact that all of its nodes (with the exceptions mentioned above) have the same in-degree (called *spread* in banyan terminology) and the same out-degree (called *fanout*). These parameters are represented by the letters $s$ and $f$, respectively.

Regular banyans can be further classified into different networks, according to the connection scheme applied to the nodes. Two of them have been reported in the literature, the SW-banyan and the CC-banyan. The former is defined as a recursive expansion of a crossbar structure (which itself can be thought of as a one-level SW-banyan), by interconnecting $s^{l-1}$ such crossbars and $f$ identical $l-1$ SW structures, all with the same fanout and spread.[1]

SW-banyans can also be further divided into *rectangular* and *non-rectangular* SW-banyans, the only difference being that for the former the spread and fanout are the same. SW-banyans are described by a set of numbers in the format $(s,f,l)$ for non-rectangular banyans, and by $(s,l)$ or $(f,l)$ for rectangular banyans. This latter class of SW- banyans has been shown to be topologically equivalent to a number of other proposed topologies [16], and as such the results presented here could also be applied to these isomorphic topologies.

[1] $l$ is the number of levels in the SW-banyan.

0190-3918/86/0000/0100 $01.00 © 1986 IEEE

## 3 Skewed Banyan Topology

### 3.1 Basic principle

One property of (s,f,l)SW-banyans is that $s$ nodes at level $i$ have each one of their links connected to the same $f$ nodes at level $i+1$. In other words, we can say that $f$ nodes at level $i+1$ are connected to the same $s$ nodes at level $i$. This property is illustrated in figure 2 for a (2,3,2) SW-banyan.

With this connection scheme we can notice that $f$ edge-disjoint paths of length 2 exist between the $s$ nodes of every set at level $i$. Although the redundant paths improve the reliability of connections between $s$ nodes, and only between these, they represent a bottleneck if we consider their distance to other nodes at the same level or to nodes at different levels(the numbers at each node in figure 2 represent their distance to the node marked '0'). If the connections in the original network are rearranged in a way such that two nodes at level $i$ have no more than a single node in common at level $i+1$, the remaining $f-1$ links for each node at level $i$ might be connected to nodes at level $i+1$ that connect to different nodes at level $i$, rather than back to the original two nodes. The basic idea then is to reduce the distance between nodes at the same level by avoiding redundant connections between them through nodes at level $i+1$. For the original network of figure 2, the connections can be rearranged as shown in figure 3. Comparing the distance properties between these two connection schemes, we can observe some improvement caused by the use of a skewing scheme (the numbers represent distances to node '0' as in figure 2).

That these improvements are scalable to larger size networks or to networks with different fanouts and spreads can be easily verified, and examples in a later section will show this.

### 3.2 Definition of SK-banyans

As an SK-banyan is closely related to an SW-banyan, we will define the former in relation to the latter.

**Definition 1:** An (s,f,l) SK-banyan is a banyan network for which the following conditions hold :

1. it is regular;
2. it has the same number of nodes per level as an (s,f,l) SW-banyan;
3. it has the same number of links between levels as an (s,f,l) SW-banyan.

We continue by defining an important property of SK-banyans, namely, the existence of two characteristic levels.

**Definition 2:** The characteristic levels of an (s,f,l) SK-banyan are the two levels that correspond, on an (s,f,l) SW-banyan, to the levels whose respective connected subgraphs have the following property :

$$n_i = \begin{cases} s \times f & \text{if } s \leq f \\ s \times s & \text{if } s > f \end{cases}$$

$$n_{i+1} = \begin{cases} f \times f & \text{if } s \leq f \\ f \times s & \text{if } s > f \end{cases}$$

where :

$n_i$ - number of nodes at level i
$n_{i+1}$ - number of nodes at level i+1
s - spread
f - fanout

Another related definition is presented as follows.

**Definition 3:** The Characteristic Connection Pattern **(CCP)** of an (s,f,l) SK-banyan is an undirected graph formed by the characteristic levels and the connection pattern between them.

In figure 4, a (2,3,3) SW-banyan is shown. For this case, $s < f$ and then $n_i = s * f = 6$ and $n_{i+1} = f * f = 9$. Of all the connected subgraphs in this particular banyan, the only ones that satisfy definition 2 are those marked. They will correspond then to the characteristic levels on a (2,3,3) SK-banyan. If the connections between these levels are rearranged, the particular connection pattern chosen will constitute the Characteristic Connection Pattern (CCP) of the SK-banyan, as defined. Two of the many possible choices for the CCP for a (2,3,l) SK- banyan are shown in figure 5.

The importance of defining the characteristic levels and the characteristic connection pattern will become clear after the following definition.

**Definition 4:** A uniform (s,f,l) SK-banyan is an (s,f,l) SK-banyan for which the connection patterns between levels are reproductions of the Characteristic Connection Pattern, taken on a group-wise basis.

In figure 6, a (2,3,3) SK-banyan is shown, for which the CCP is that shown in figure 5(a), and the nodes at levels 1 and 2 constitute the characteristic levels. If we look at a group of connections between levels 2 and 3, each group with three links, we can recognize the same pattern as the CCP. This is what is meant by "group-wise" in the above definition. Between levels where $n_i > s * f$ and $n_{i+1} > f * f$ (for $s <= f$), a group of connections, rather than individual connections, will follow the CCP, no matter how large the group.

Since the CCP completely defines a uniform SK-banyan, once it is chosen we are able to construct the whole network and to define a routing algorithm. Although not proved here, it may also be anticipated that non-uniform SK-banyans, where the connection patterns between levels are not necessarily reproductions of the characteristic connection pattern, may be more difficult to construct and as complex as irregular banyans for data routing, thus will not be considered here. Also, by Definition 1, SW-banyans can be considered a case of SK-banyans, for which a specific CCP is used.

For simplicity, we will assume in the following discussion only the case for $s <= f$. The case for $s > f$ obeys the same principles.

We now introduce a matricial representation of the CCP, in order to facilitate its analysis and classification. The CCP is a bipartite graph, where one set has $s * f$ nodes and the second set has $f * f$ nodes, as given by Definition 2. The CCP can be further divided into $s * f$ bipartite subgraphs, each one with $f$ nodes in each of its two sets. The number of different subgraphs is equal to $f!$, and as there are $s * f$ of these subgraphs in a CCP, the total number of different CCP's for a given $s$ and $f$ is equal to $(f!)^{s*f}$. Table 1 shows these values for different spreads and fanouts.

The natural partition of the CCP into bipartite subgraphs of a larger bipartite graph leads to the following definitions.

**Definition 5:** The Connection Matrix (**CM**) of a bipartite subgraph of a CCP, is an $f \times f$ binary matrix $CM = [x_{ij}]$ such that :

$$x_{ij} = \begin{cases} 1 \text{ if node } i \text{ is connected to node } j \\ 0 \text{ if they are not connected} \end{cases}$$

**Definition 6:** A Primitive Connection Matrix (**PCM**) is one element of the set that is composed of all the $f!$ possible Connection Matrices.

**Definition 7:** The Characteristic Connection Matrix (**CCM**) of a CCP is an $s \times f$ matrix whose elements are Connection Matrices taken from the set of Primitive Connection Matrices.

Given the CCP of figure 5 (a), we can partition it into six bipartite subgraphs as shown in figure 7, where the respective CM's are also shown. The CCM will be formed by these CM's, as shown in figure 8. The set of PCM's for this case ($s = 2; f = 3$) is shown in figure 9, along with the respective graphs. Numbering each of the PCM's from *0* to $[(f!) - 1]$, the CCM can be represented as an $s \times f$ matrix whose elements represent the number of the PCM that corresponds to each CM in the CCM. For the CCM of figure 8, this notation would lead to the matrix shown in figure 10. This notation is more compact and as complete as the CCM itself.

### 3.3 Optimality criteria

It is clear, by looking at table 1, that an algorithm or criteria should be proposed in order to determine which of the large range of CCP's provides the best distance properties. Given the size of the problem, and the fact that in order to gather meaningful results we must deal with reasonably large networks, an analytical approach seems unfeasible. Instead, we performed a computation of the distance properties of a subset of all the possible (2,3,4) SK-banyans. From these results, we were able to derive two criteria that can determine, based on the CCM, if the resulting (2,3,l) SK-banyan is optimal in terms of distance properties.

The first criterion is related to the connectivity of the nodes of the upper level of the CCP. As pointed out in section 3.1, the connections of these nodes should be made in a way such that they linked the maximum number of nodes in the lower level. This spreading of connections is represented by a matrix, called Spread Connectivity Matrix, defined as follows.

**Definition 8:** The *k-th* Spread Connectivity Matrix (**SCM**) SCM$^k$ of a CCP is an $f \times f$ matrix whose elements are given by the *k-th*-row-summation of the Connection Matrices of the corresponding Characteristic Connection Matrix, or :

$$SCM^k_{ij} = \sum_{l=0}^{f-1} CCM^l_{ij}$$

$$\text{for } 0 \leq i,j \leq f-1$$
$$0 \leq k \leq s-1$$

As the CCM is an $s \times f$ matrix, there will be $s$ SCM's. We can now state the first optimality criterion.

**Definition 9:** Connectivity Criterion : a (2,3,l) SK-banyan presents optimal distance properties if all the elements of every SCM are equal to 1, or :

$$SCM^k_{ij} = 1$$

$$\text{for } 0 \leq i,j \leq f-1$$
$$0 \leq k \leq s-1$$

This criterion was established by verifying that from the set of PCM's, $f$ of them can always be selected such that their summation, as defined above, leads to a matrix with all the elements equal to 1. This corresponds to the optimal case.

The second criterion is related to the redundancy in connections between two nodes of the upper level of the CCP. As also pointed out in section 3.1, nodes at the upper level should have only one node in common at the lower level. This redundancy is represented by a matrix, called Redundancy Matrix, defined as follows.

**Definition 10:** The Redundancy Matrix (**RM**) of a CCP is an $f \times f$ matrix whose elements represent the number of ocurrences of the indices $i,j$ of the nodes of the upper level of the CCP to which each of the nodes of the lower level are connected.

The nodes at the upper level are numbered as shown in figure 11. The numbers at each node of the lower level indicate the indices of the nodes of the upper level to which they are connected. In this case, each ordered pair occurs only once, and so the Redundancy Matrix will have all of its elements equal to 1. We state the second criterion as follows.

**Definition 11:** Redundancy Criterion : a (2,3,l) SK-banyan presents optimal distance properties if all the elements of the Redundancy Matrix are equal to 1, or :

$$RM_{ij} = 1$$

$$\text{for } 0 \leq i,j \leq f-1$$

This criterion was established by noticing that any redundant connection results in an entry in the Redundancy Matrix equal to 2, whereas the absence of a connection results in an entry equal to 0. The only case where there is no redundancy is when all the entries in the Redundancy Matrix is equal to 1, which means that there are no redundancy or absence of connections between the nodes.

As an example of the application of these criteria, figure 12 shows three cases of different CCP's with their respective CCM's and the Connectivity and Redundancy matrices. Figure 12 (a) refers to a (2,3,l) SW-banyan, (b) refers to a non-optimal CCP, and (c) to an optimal CCP. As can be seen, only in the last case the two optimality criteria are satisfied.

Using these criteria to compute the Spread Connectivity and the Redundancy matrices of all the (2,3,4) SK-banyans takes less than 1/1000 of the processor time needed to compute their distance matrices and average distances. By selecting the PCM's such that the connectivity criterion is satisfied, the range of CCM's to be searched for optimal cases can be significantly reduced. A procedure to perform these selections systematically is being investigated.

### 3.4 Construction Scheme

Given the definitions from the previous section, we now present a construction scheme for (s,f,l) SK-banyans, based on the CM's. One reason for using these matrices is that, once $s$ and $f$ are selected, the PCM's are defined and any CCM can be constructed by assigning to each of its CM's the corresponding PCM.

```
for i = 0 to (number of levels - 1) do

  $gs_i = s * f^i$
  $bs_i = f^i$

  for k = 0 to (number of nodes
              at level i) - 1 do

    offset = k / $gs_i$
    $gn_{k,i}$ = k mod $gs_i$
    $bn_{k,i}$ = (k mod $gs_i$) / $bs_i$

    for l = 0 to (f - 1) do

      skew = skew_factor [ pcm [ cm_num, l]]

      and connect

        node(k,i) to
          node(offset + l * $bs_i$ + skew, i+1)


    where :

      $gs_i$ - group size at level i
      $bs_i$ - block size at level i

      k, i - node number (number, level)

      offset - lowest node number of the group
      gn - group number of the node
      bn - block number of the node

      cm_num - CM number (0 <= CM <= ($s*f$)-1)
      pcm - PCM number (0 <= PCM <= f-1)
```

This algorithm has been implemented in Pascal, and was used to generate all of the examples used in this paper.

# 4 Distance Properties

The distance properties of some examples of SW- and SK-banyans will now be presented, and two metrics will be used. One measures the percentile of nodes that are within a given distance from a particular set of nodes. We will call this measure the *reach factor*. The second measures the average distance between sets of nodes.

## 4.1 (2,5) SW- and SK-banyans

It is also necessary to consider rectangular banyans, hence table 2 shows the distance properties of both banyans, in terms of reach factor. It can be seen that SK-banyans provide better distance properties than SW-banyans. This table also shows a very important property, namely, that the improvement obtained for some levels is not at the expense of others. As can be seen, level 0 in both banyans present the same distance properties, but starting at level 1, the SK-banyan provides improved distance at every level. Figure 13 shows the same results in a graphical form, for levels 0 and 4.

## 4.2 (2,3,4) SW- and SK-banyans

Table 3 shows the distance properties for (2,3,4) SW- and SK- banyans. Again, the SK-banyan shows an improvement over SW-banyans, this time higher than in the previous case. Figure 14 compares the reach factor for the two types of banyans. As before, the improvements obtained for any level is not at the expense of the distance properties of any other level.

## 4.3 (2,3,4) SK-banyans with different CCP's

As mentioned in the previous section , different CCP's may provide different distance properties. Computing the distance matrices for the CCP's of figure 5, we obtain the results shown in table 4. It is important to realize that not only are some improvements achieved by using SK- over SW-banyans, but also by choosing one CCP over another. Figure 15 shows how different CCP's compare to each other and to the respective (2,3,4) SW-banyan.

## 4.4 Average distances

For a specific subset of (2,3,4) SK-banyans (which included the sub-class of SW-banyans), we computed the average distances between sets of nodes at each level. Three results are shown in table 5, respectively for the SK-banyans whose CCM's are those shown in figure 12. As shown, the improvement in average distance varies with the level. For a better visualization figure 16 shows the variation of the average distance with the CCP type, for this particular subset of SK-banyans.

# 5 Conclusions

A new class of banyan networks has been presented which has been shown to have better distance properties than SW-banyans. Although these improvements have been demonstrated only in relation to distance properties, it should be expected that similar improvements would occur for other properties, especially traffic and fault-tolerance. Justification for this reasoning is provided by way of analogy with KYKLOS and many other topologies, which have shown the connection between distance and traffic [9].

Improvements in the distance properties are essential for minimizing communication overhead. An example of this is the architecture proposed for database applications [2]. In this architecture, a set of host computers at the apex of an ICN processes the queries and send requests to a set of I/O nodes at the base of the same ICN. The storage allocation scheme proposed assumes heavy traffic between these I/O nodes, both to decrease the effects of "hot spots" and to reallocate the database as it changes. It is desirable in this case that the distance between these nodes be as low as possible, without aggravating the distance properties of the apex nodes.

The matricial representation of CCP's allowed the construction and classification of SK-banyans, as well as the proposal of criteria for determining the optimality of a CCP. Although results here were provided only for the case where $s = 2; f = 3$, the formulations and definitions are general in nature.

Because SW-banyans are a case of just one CCP, the concept of SK-banyans represent a generalization of the topology , and allow a more unified view of the properties resulting from interconnection variation. The symetry applied to conventional networks, while providing simple routing, has lead to redundant connections. Minimization of this redundancy results in networks with a richer set of properties such as distance and traffic.

## Acknowledgement

This paper was supported in part by grants AFOSRF49620-84-C-0020, DCR-8116099, DEAS05-81ER-10987 and Space and Naval Warfare Systems Command Contract N00039-86-C-0167.

# References

[1] Dharma P. Agrawal and Ja-Song Leu, "Dynamic Accessability Testing and Path Length Optimization of Multistage Interconnection Networks", Proceedings of the 4th. International Conference on Distributed Computing Systems, 1984, pp. 266-277.

[2] J.C. Browne, A.G. Dale, C. Leung and R.M. Jenevein, "A Parallel Multi-Stage I/O Architecture with Self-Managing Disk Cache for Database Management Applications", Preceedings of the 4th. International Workshop on Database Machines, March 1985, pp. 330-345.

[3] V. Cherkassky, E. Opper, M. Malek, "Reliability and Fault Diagnosis Analysis of Fault-Tolerant Multistage Interconnection Networks", 14th. Annual Intl. Symp. on Fault-Tolerant Computing, June 1984, pp. 246-251.

[4] Mark A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communication Networks", IEEE Transactions on Computers, Vol. C-30, No. 4, April 1981, pp. 283-290.

[5] L.R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems", Proceedings of the First Annual Symposium on Computer Architecture, 1973, pp. 21-28.

[6] J.R. Goodman and C.H. Sequin, "Hypertree : A Multiprocessor Interconnection Topology", IEEE Transactions on Computers, Vol C-30, No. 12, December 1981, pp. 923-933.

[7] R.M. Jenevein, D. Degroot and G.J. Lipovski, "A Hardware Support Mechanism for Scheduling Resources in a Parallel Machine Environment", Proceedings of the 8th. Annual International Symposium on Computer Architecture, May 1981, pp. 57-66.

[8] R.M. Jenevein and J.C. Browne, "A Control Processor for a Reconfigurable Array Computer", Proceedings of the 9th. Annual Symposium on Computer Architecture, April 1982, pp. 81-89.

[9] R.M. Jenevein and B.L. Menezes, "KYKLOS : Low Tide, High Flow", to appear on the Proceedings of the 6th. International Conference on Distributed Computing Systems, 1986.

[10] Charles E. Leiserson, "FAT-TREES: Universal Networks for Hardware-Efficient Supercomputing", Proceedings of the 14th. International Conference on Parallel Processing, 1985, pp. 393-402.

[11] J. Lenfant, "Parallel Permutations of Data : A Benes Network Control Algorithm for Frequently Used Permutations", IEEE Transactions on Computers, Vol C-27, No. 7, July 1978, pp. 637-647.

[12] Miroslaw Malek and Eli Opper, "Multiple Fault Diagnosis of SW-Banyan Networks", Digest of Papers, 13th. Annual Intl. Symp. on Fault-Tolerant Computing, June 1983, pp. 446-449.

[13] B.L. Menezes and R.M. Jenevein, "Kyklos : A linear Growth Fault-tolerant Interconnection Network", Proceedings of the 14th. International Conference on Parallel Processing, 1985, pp. 498-502.

[14] U.V. Premkumar, **A Theoretical Basis for the Analysis and Partitioning of Regular SW-Banyans**, Ph.D. Thesis, University of Texas at Austin, August 1981.

[15] U.V. Premkumar and J.C. Browne, "Resource Allocation in Rectangular SW-banyans", Proceedings of the 9th. Annual Symposium on Computer Architecture, 1982, pp. 326-333.

[16] Chuan-lin Wu and Tse-yun Feng, "On a Class of Multistage Interconnection Networks", IEEE Transactions on Computers, Vol C-29, No. 8, August 1980, pp. 694-702.

[17] Chuan-lin Wu and Tse-yun Feng, ed., "Interconnection Networks for Parallel and Distributed Processing", IEEE Computer Society Press, 1984.

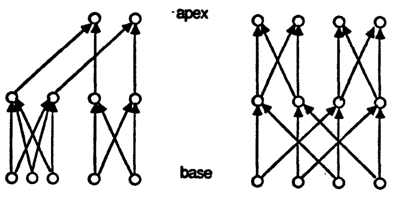


**Figure 1:** Banyan networks

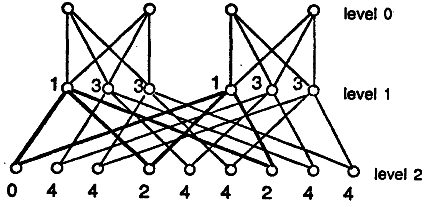


**Figure 2:** Redundant connections on regular banyans

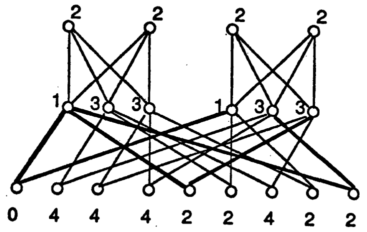


**Figure 3:** SK-banyan

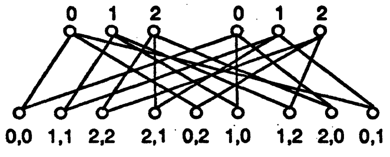


**Figure 11:** Numbering of nodes in the CCP

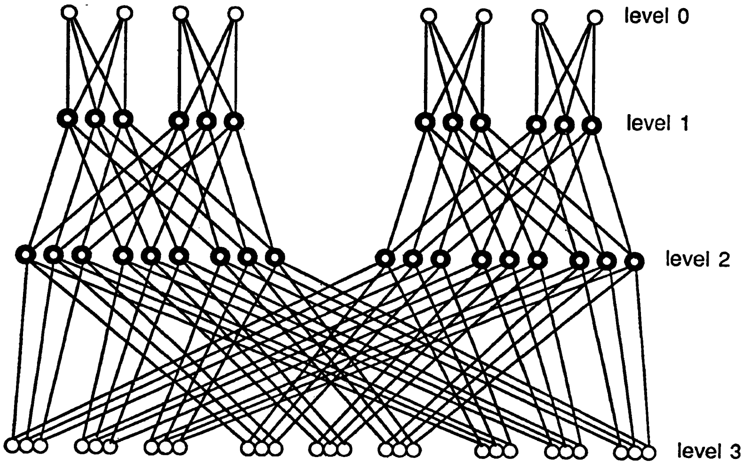
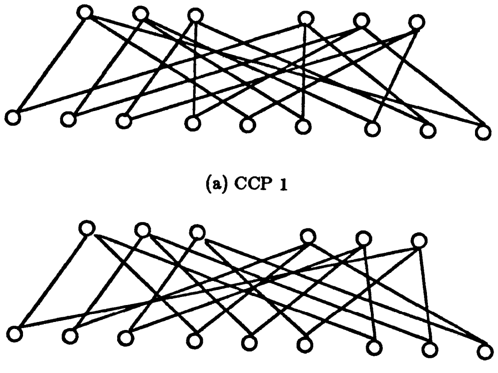


**Figure 4:** (2,3,3) SW-banyan



(a) CCP 1

(b) CCP 2

**Figure 5:** Examples of Characteristic Connection Patterns for a (2,3,l) SK-banyan

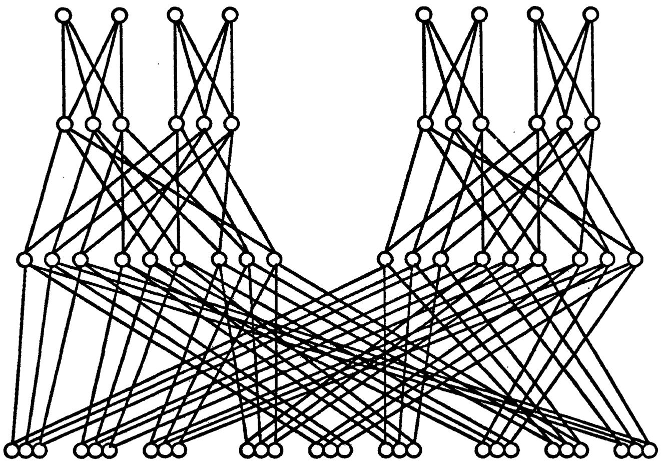


**Figure 6:** Reproduction of a Characteristic Connection Pattern on a (2,3,3) SK-banyan

**Table 1:** Number of PCM's and CCP's as a function of $s$ and $f$

| s | f | # of PCM's | # of CCP's |
|---|---|---|---|
| 2 | 2 | 2 | 16 |
| 2 | 3 | 6 | 46656 |
| 3 | 3 | 6 | 10,077,696 |
| 2 | 4 | 24 | ~ $1.1 \times 10^{11}$ |
| 3 | 4 | 24 | ~ $3.7 \times 10^{16}$ |
| 4 | 4 | 24 | ~ $1.2 \times 10^{22}$ |
| 2 | 5 | 120 | ~ $6.2 \times 10^{20}$ |
| 3 | 5 | 120 | ~ $1.5 \times 10^{31}$ |
| 4 | 5 | 120 | ~ $3.8 \times 10^{41}$ |
| 5 | 5 | 120 | ~ $9.5 \times 10^{51}$ |

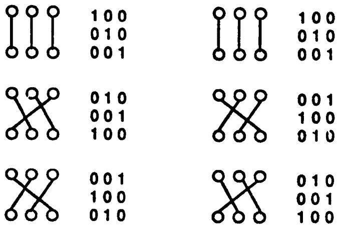


**Figure 7:** Subgraphs of the CCP

| CCP | CCM | | |
|---|---|---|---|
| 0 1 2 | 1 0 0 | 0 1 0 | 0 0 1 |
| 0 2 1 | 0 1 0 | 0 0 1 | 1 0 0 |
| | 0 0 1 | 1 0 0 | 0 1 0 |
| | 1 0 0 | 0 0 1 | 0 1 0 |
| | 0 1 0 | 1 0 0 | 0 0 1 |
| | 0 0 1 | 0 1 0 | 1 0 0 |

**Figure 8:** Example of CCM

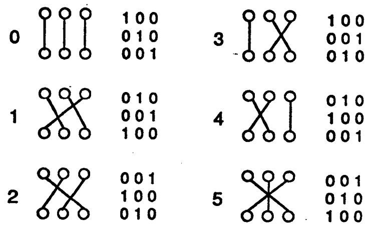


**Figure 9:** PCM's for $s = 2; f = 3$

000 000 012
000 012 021

or

000000 000012 012021

**Figure 10:** Compact notation for CCM's

```
CCP              CCM                 Connectivity

0 0 0    1 0 0   1 0 0   1 0 0     3 0 0
0 0 0    0 1 0   0 1 0   0 1 0     0 3 0
         0 0 1   0 0 1   0 0 1     0 0 3
                                           Redundancy
         1 0 0   1 0 0   1 0 0     3 0 0    3 0 0
         0 1 0   0 1 0   0 1 0     0 3 0    0 3 0
         0 0 1   0 0 1   0 0 1     0 0 3    0 0 3

                   (a) SW-banyan

CCP              CCM                 Connectivity

0 0 0    1 0 0   1 0 0   1 0 0     3 0 0
0 1 2    0 1 0   0 1 0   0 1 0     0 3 0
         0 0 1   0 0 1   0 0 1     0 0 3
                                           Redundancy
         1 0 0   0 1 0   0 0 1     1 1 1    1 1 1
         0 1 0   0 0 1   1 0 0     1 1 1    1 1 1
         0 0 1   1 0 0   0 1 0     1 1 1    1 1 1

                 (b) Non-optimal CCP

  CCP            CCM                 Connectivity

0 1 2    1 0 0   0 1 0   0 0 1      1 1 1
0 2 1    0 1 0   0 0 1   1 0 0      1 1 1
         0 0 1   1 0 0   0 1 0      1 1 1
                                           Redundancy
         1 0 0   0 0 1   0 1 0      1 1 1    1 1 1
         0 1 0   1 0 0   0 0 1      1 1 1    1 1 1
         0 0 1   0 1 0   1 0 0      1 1 1    1 1 1

                   (c) Optimal CCP
```

**Figure 12:** Examples of Connectivity and Redundancy Matrices

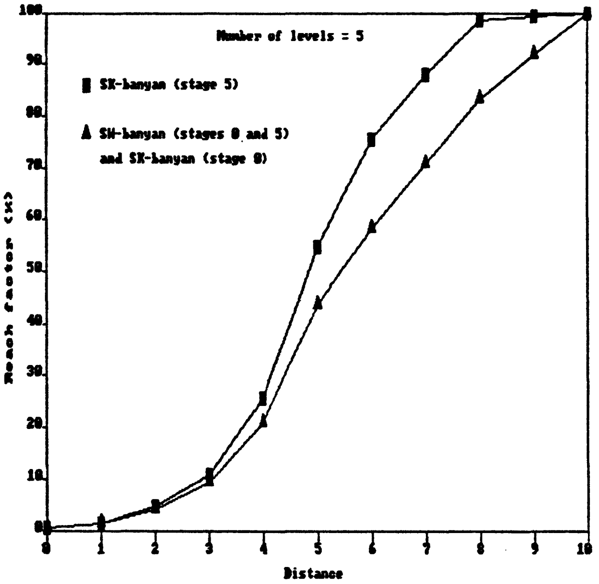


**Figure 13:** Reach factor for (2,5) SW- and SK-banyans

**Table 2:** Distance properties of (2,5) SW- and SK-banyans

| | | distance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| level | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | sk | 2 | 4 | 9 | 21 | 44 | 58 | 71 | 83 | 92 | 100 |
| | sw | 2 | 4 | 9 | 21 | 44 | 58 | 71 | 83 | 92 | 100 |
| 1 | sk | 3 | 6 | 14 | 28 | 41 | 59 | 74 | 85 | 98 | 100 |
| | sw | 3 | 6 | 13 | 27 | 40 | 58 | 73 | 83 | 96 | 100 |
| 2 | sk | 3 | 9 | 19 | 30 | 47 | 63 | 79 | 94 | 99 | 100 |
| | sw | 3 | 8 | 16 | 25 | 38 | 52 | 71 | 88 | 96 | 100 |
| 3 | sk | 3 | 9 | 20 | 33 | 50 | 69 | 87 | 97 | 99 | 100 |
| | sw | 3 | 8 | 16 | 25 | 38 | 52 | 71 | 88 | 96 | 100 |
| 4 | sk | 3 | 7 | 17 | 35 | 54 | 75 | 92 | 98 | 100 | 100 |
| | sw | 3 | 6 | 13 | 27 | 41 | 58 | 73 | 83 | 96 | 100 |
| 5 | sk | 2 | 5 | 11 | 26 | 55 | 76 | 88 | 98 | 99 | 100 |
| | sw | 2 | 4 | 9 | 21 | 44 | 58 | 71 | 83 | 92 | 100 |

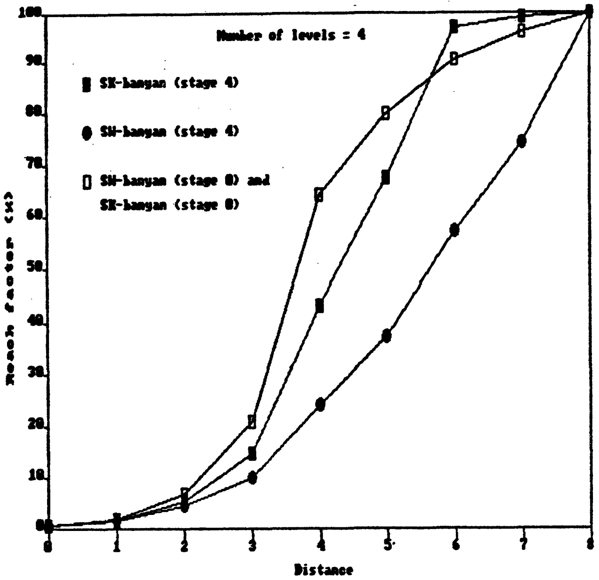


**Figure 14:** Reach factor for (2,3,4) SW- and SK-banyans

**Table 3:** Distance properties of (2,3,4) SW- and SK-banyans

| | | distance | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| level | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | sk | 2 | 7 | 21 | 64 | 80 | 91 | 96 | 100 |
| | sw | 2 | 7 | 21 | 64 | 80 | 91 | 96 | 100 |
| 1 | sk | 3 | 9 | 30 | 55 | 91 | 96 | 100 | 100 |
| | sw | 3 | 9 | 27 | 41 | 74 | 87 | 96 | 100 |
| 2 | sk | 3 | 12 | 30 | 65 | 82 | 100 | 100 | 100 |
| | sw | 3 | 10 | 18 | 33 | 48 | 82 | 94 | 100 |
| 3 | sk | 3 | 8 | 29 | 51 | 88 | 97 | 100 | 100 |
| | sw | 3 | 6 | 16 | 25 | 45 | 60 | 91 | 100 |
| 4 | sk | 1 | 5 | 15 | 43 | 68 | 97 | 99 | 100 |
| | sw | 1 | 4 | 10 | 24 | 37 | 57 | 74 | 100 |

**Table 4:** Distance properties of (2,3,4) SK-banyans with different CCP's

|  |  | distance |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| level |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | sk | 2 | 7 | 21 | 64 | 80 | 91 | 96 | 100 |
|  | sw | 2 | 7 | 21 | 64 | 80 | 91 | 96 | 100 |
| 1 | sk | 3 | 9 | 30 | 55 | 91 | 96 | 100 | 100 |
|  | sw | 3 | 9 | 27 | 42 | 75 | 89 | 98 | 100 |
| 2 | sk | 3 | 12 | 30 | 65 | 82 | 100 | 100 | 100 |
|  | sw | 3 | 12 | 23 | 49 | 64 | 90 | 98 | 100 |
| 3 | sk | 3 | 8 | 29 | 51 | 88 | 97 | 100 | 100 |
|  | sw | 3 | 8 | 23 | 39 | 72 | 87 | 100 | 100 |
| 4 | sk | 1 | 5 | 15 | 43 | 68 | 97 | 99 | 100 |
|  | sw | 1 | 5 | 13 | 35 | 55 | 85 | 92 | 100 |

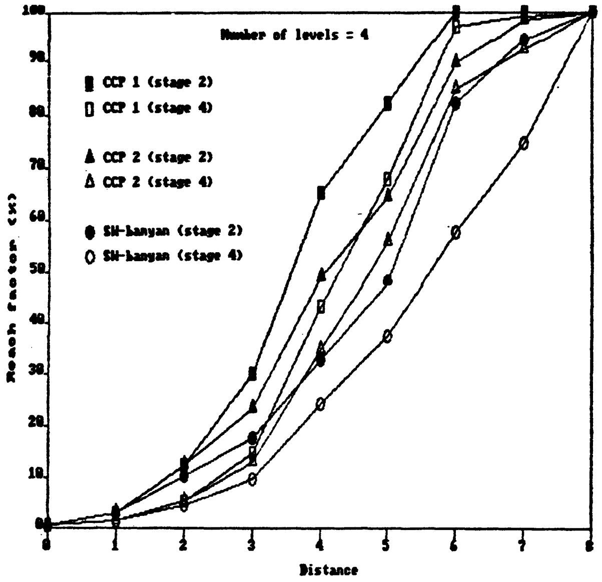


**Figure 15:** Reach factor for different CCP's for (2,3,4) SK-banyans

**Table 5:** Average distances (per level and total) for (2,3,4) SK-banyans

CCP type : 000000 (SW-banyan)
Average distances by level :

| From \ To | 0 | 1 | 2 | 3 | 4 | All |
|---|---|---|---|---|---|---|
| 0 | 6.125 | 5.250 | 4.500 | 4.000 | 4.000 | 4.389 |
| 1 | 5.250 | 5.583 | 4.833 | 4.333 | 4.333 | 4.630 |
| 2 | 4.500 | 4.833 | 5.611 | 5.111 | 5.111 | 5.119 |
| 3 | 4.000 | 4.333 | 5.111 | 6.037 | 6.037 | 5.531 |
| 4 | 4.000 | 4.333 | 5.111 | 6.037 | 7.012 | 5.905 |

Total : 5.415

CCP type : 000012 (Non-optimal CCP)
Average distances by level :

| From \ To | 0 | 1 | 2 | 3 | 4 | All |
|---|---|---|---|---|---|---|
| 0 | 6.125 | 5.250 | 4.500 | 4.000 | 4.000 | 4.389 |
| 1 | 5.250 | 4.708 | 4.417 | 4.167 | 4.333 | 4.417 |
| 2 | 4.500 | 4.417 | 4.333 | 4.333 | 4.667 | 4.483 |
| 3 | 4.000 | 4.167 | 4.333 | 4.630 | 5.148 | 4.678 |
| 4 | 4.000 | 4.333 | 4.667 | 5.148 | 5.778 | 5.128 |

Total : 4.766

CCP type : 012021 (Optimal CCP)
Average distances by level :

| From \ To | 0 | 1 | 2 | 3 | 4 | All |
|---|---|---|---|---|---|---|
| 0 | 6.125 | 5.250 | 4.500 | 4.000 | 4.000 | 4.389 |
| 1 | 5.250 | 4.417 | 3.833 | 3.667 | 4.333 | 4.156 |
| 2 | 4.500 | 3.833 | 4.167 | 3.778 | 4.222 | 4.076 |
| 3 | 4.000 | 3.667 | 3.778 | 4.407 | 4.556 | 4.242 |
| 4 | 4.000 | 4.333 | 4.222 | 4.556 | 5.284 | 4.711 |

Total : 4.395

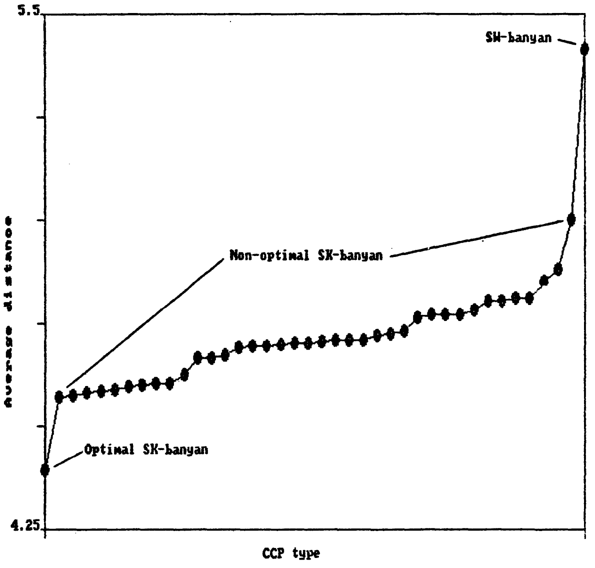


**Figure 16:** Range of total average distance for a subset of (2,3,4) SK-banyans

# PERFORMANCE STUDIES OF MULTIPLE-PACKET MULTISTAGE CUBE NETWORKS AND COMPARISON TO CIRCUIT SWITCHING

Nathaniel J. Davis IV
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433 USA

Howard Jay Siegel
PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907 USA

ABSTRACT -- This paper extends the modeling and analysis of packet switched multistage cube interconnection networks to include the use of multiple-packet message formats. A multiple-packet message format is needed within an interconnection network when the message to be transmitted exceeds the network's packet size. The packet offset time is introduced and used to gauge the transmission rate differential between the network interchange boxes and the system PEs. Based on assumed network and system operating characteristics, optimum network performance offset values are given. A methodology for evaluating the relative performance of circuit and packet switching transmission modes is developed. For an assumed set of network and system parameters, results indicate that, in contrast to inter-computer networks, the performance of packet switching networks is better than that of circuit switching networks for long message lengths.

## 1. INTRODUCTION

An integral part of any parallel processing system is its interconnection network, used to link the processors and, possibly, memory modules together. One class of networks suitable for use in a parallel processing system is the multistage cube networks [24]. This class includes topologically equivalent networks such as the baseline [26], the indirect binary n-cube [18], the generalized cube [22], the omega [14], the flip [1], and the SW-banyan (S=F=2) [11]. Multistage cube networks have been used in the STARAN [1] and BBN Butterfly [4] and have been proposed for use in many future systems such as: the IBM RP3 [19], PASM [23], Ultracomputer [10], and database machines [7].

Multistage cube networks can be designed to function using circuit or packet switching methodologies for data transmission within the networks. A general framework is developed in this paper that quantizes system and network parameters and thereby facilitates the comparison of circuit and packet switched networks. A specific set of network and system parameters are used to demonstrate the utility of the framework. The general framework can be used by network designers to select which network switching methodology best meets their system requirements. Networks can also be designed that incorporate both circuit and packet switching modes [20]. In networks where both modes are available, this analysis framework can be used to identify which transmission mode should be used for a given set of operating conditions.

In a packet switched network, when the amount of data to be transmitted exceeds the network's packet size, multiple data packets must be generated by the network source and then be transmitted to the appropriate destination. The performance analysis of networks operating in a multiple-packet environment is considered in this paper. Packet switching network models are extended, through the use of simulations, to include the transmission of multiple-packet messages. A tradeoff analysis, based on assumed network and system operating characteristics, is made to compare the performance of this format to that of circuit switching. This research has been motivated by the design of the PASM parallel processing system system at Purdue University [23, 16].

Section 2 defines the system and network operating environments. The performance analysis of multiple-packet message transmissions is then presented in Section 3. Performance comparisons between circuit and packet switching network operations is discussed in Section 4. Conclusions are presented in Section 5.

## 2. NETWORK MODEL AND SYSTEM OPERATION

In this paper, the generalized cube network will be used as a model of all of the networks within the multistage cube class. The network model will be the same as the models developed in previous analyses for omega networks [2, 3] and baseline networks [15]. Assume the network has N inputs (sources) and N outputs (destinations), where $N = 2^n$. The network consists of n stages with each stage being composed of N/2 2-by-2 interchange boxes. Interchange boxes in stage *i* pair I/O lines with labels that differ only in the *i-th* bit position. The same labeling is used for both the input and the output lines connected to an interchange box. A generalized cube network is shown in Figure 1, with N=8. The network's sources and destinations are assumed to be *processing elements (PEs)*, processor-memory pairs; i.e., PE i is connected to network input port i and network output port i, $0 \leq i < N$. This configuration is generally referred to as a *PE-to-PE* system architecture. Distributed routing control is assumed, with the settings of the individual interchange boxes (straight or exchange -- broadcasting will not be considered in the model) determined by routing information contained within the routing tag of each message [14, 22]. Prior to the transmission of data between a source-destination pair in a circuit switched network implementation, a complete path linking the pair must be established and then held until the completion of the data transfer. Using packet switching, a message consists of one or more data packets. Each packet makes its way from stage to stage releasing links and interchange boxes immediately after using them.

The network is assumed to be operating under the following assumptions [8]. Each source PE generates its messages independently from all other sources. Each source generates messages for each destination PE with equal probability. Messages will be generated based on a predetermined, fixed loading factor. The destination PE can receive data from the network faster than the network can transmit the data (this simplification implies that an output device will not act to bottleneck the operation of the network itself). Finally, a message, or a message packet, consists of two parts -- the routing tag and one or more data words.

This research was supported by the Rome Air Development Center, under contract number F30602-83-K-0119.

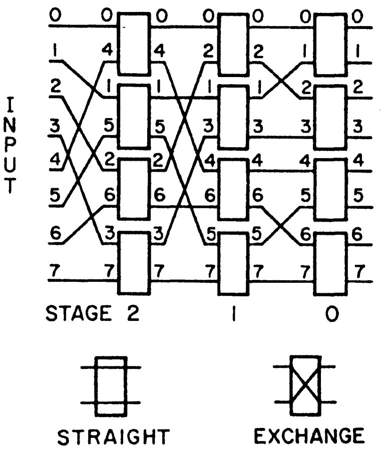


Figure 1. The generalized cube network for N=8.

In an interchange box, if two different message requests (routing tags) attempt to reserve the same output link, or (in the case of circuit switching) if a request is blocked by an already established message path, a conflict is said to have occurred. A conflict resolution algorithm must be invoked to resolve the conflict. In packet switching networks, the algorithm selects one of the requests and permits it to pass through the interchange box. The other request is held in an input buffer and will attempt to traverse the box at a later time. In circuit switching networks, the message that is blocked at the switch is either dropped and then resubmitted to the network (the drop algorithm) or is held in the box until the blockage is removed (the hold algorithm). These two circuit switched algorithms have been investigated in, for example, [15, 6].

## 3. MULTIPLE-PACKET NETWORK OPERATION

In this section, the effects of multiple-packet messages on the operation of a packet switching generalized cube network are investigated. The system is assumed to be operating in the MIMD mode. The message to be transmitted will consist of a routing tag and one or more data words. If, however, the message size exceeds the size of the network transmission packet, multiple dependent packets must be generated. Each packet will contain the same routing tag and a portion of the data to be transferred. The packets are processed sequentially by the network since only one network path exists between any source-destination pair.

Operational dependencies exist between multiple-packet messages that do not exist when single-packet message formats are used. These dependencies center on the sequential generation of packets (within any given message) all being routed to the same network output port (recall that in single-packet messages, the routing of all of the packets is assumed to be a strictly independent process [8]). As an example, consider two multiple-packet messages, where the packets of one message are being held in one input buffer of an interchange box and the second message's packets are held in the other input buffer. Any conflict (or lack of a conflict) that occurs when the first packets in the two buffers are processed will also occur when the succeeding packets are processed.

The dependencies that exist between packets in multiple-packet message transmissions cause network performance analysis using "standard" Markov chain modeling techniques, as in [8, 3], to be extremely complex. Interconnection network simulators such as PUGS [5] can, however, be used to accurately predict the performance of multiple-packet messages. PUGS is capable of simulating both single- and multiple-packet message formats. When using the multiple-packet format, performance statistics for both individual packets and complete messages (composed of dependent packets) can be obtained. The accuracy of PUGS output data has been validated through comparisons with other researchers' published network simulation and Markov chain performance data, such as [6, 3, 15, 8]. To increase the accuracy of the statistics, the length of the simulation runs was adjusted so as to insure the simulated network had reached a "steady-state" operating condition. This occurred after the first 250-300 simulated packet cycles. The actual simulation lengths were then set to approximately ten times this value -- 2000 simulation cycles. Statistics gathered before steady-state was reached were discarded. Lastly, for each combination of network variables that were evaluated, each simulation was repeated ten time and the results averaged so as to further reduce the statistical variations. Data and statistics obtained from PUGS will be used throughout this paper.

Assume that the data to be transferred is stored in the memory of the source PE. As a result of the slow memory access time within the PE (due to memory device speed limitations and long inter-IC and inter-circuit board transmission delays) and the comparatively fast interchange box logic (small, fast packet buffers and straightforward control logic), the time required by a PE to construct a data packet will be much longer than the time required by an interchange box to process the same size packet. This operational speed differential between the PE and the network (i.e., the interchange boxes) requires that packets be held in a PE's output buffer and not released to the network until the packet is completely formed.

In analyzing multiple-packet message performance, the *packet cycle time*, defined as the time delay incurred by a packet in traversing a network interchange box, will be used as the basic unit of system/network timing. This delay is the amount of time required to move a packet from the front of one of an interchange box's input buffers to the input buffer of an interchange box in the next stage. As discussed above, this time delay will be much shorter than the time delay associated with the generation of a packet within a PE. A *packet offset time*, the time (in packet cycles) between packet generations of a multiple-packet message, can be used to quantize the speed differential between the network and the system PEs. If the time to generate a packet equals the time to process a packet in an interchange box, the packet offset would be one. If a message consisted of w packets and the first packet was generated and then submitted to the network at time t, then, in general, a y-cycle packet offset would cause packets to be submitted at times t, t+y, t+2y, $\cdots$ t+y·(w−1). The total message transmission delay is the time difference between the beginning of the generation of the first packet of the message and the completed transmission time of the last packet of the message. Note that the time from the generation of the first packet to its submission to the network is y. This delay will be y·(w) + (the transmission delay of the last packet), where the first term is the delay incurred by the message before the last packet is available for transmission. Recall that, because of pipeline effects, the delays of the other packets in the message will be reflected in the transmission delay of the last packet.

In [13], Kruskal and Snir have presented an alternate, more constrained, discussion of multiple-packet network operation. They assumed that all of the message packets were generated simultaneously (this corresponds to a packet offset of zero) and that the packets moved through the network as a single "group." Additionally, the inclusion of the packet generation time in the overall transmission time has not been incorporated into previous analyses. In [8], for example, timing delay calculations start when a packet is accepted by a network input port. Time devoted to generating a packet and any time lost waiting for the availability of an network input port is not included in their analysis.

Representative performance analysis results of multiple-packets messages, obtained using PUGS, are shown in Table 1. Results in this table are for a 5-stage network (N=32) and multiple-packet messages consisting of 2, 4, and 8 packets. The packet offset ranges from 1 to 15 packet cycles and the loading factor is 100% in each case. The loading factor is the probability in a given time cycle of a network source generating a new message, given that it is not currently transmitting a message. Buffers at the inputs to each interchange box, capable of holding four packets, are assumed to be in use (buffer size is based on analysis results in [8]). The packet and message delay times can be used to gauge the effects of the packet offset on network performance. Ideally, a packet would require k packet cycles to traverse a k-stage network. An m-packet message would, in turn, require k+(m−1) packet cycles to completely traverse the network (this time value assumes the packets are transmitted through the network in a pipeline fashion). The normalized packet and message delays can therefore be computed by dividing the packet and message delay times by k and k+(m−1), respectfully.

| Number of packets per message | Packet offset | Delay | | Normalized delay | |
|---|---|---|---|---|---|
| | | Packet | Message | Packet | Message |
| 2 | 1 | 17.90 | 20.94 | 3.58 | 3.49 |
| | 2 | 16.35 | 20.91 | 3.27 | 3.49 |
| | 5 | 6.75 | 16.60 | 1.35 | 2.77 |
| | 10 | 6.57 | 26.36 | 1.31 | 4.39 |
| | 15 | 6.56 | 36.35 | 1.31 | 6.06 |
| 4 | 1 | 21.24 | 28.18 | 4.24 | 3.52 |
| | 2 | 17.66 | 27.19 | 3.54 | 3.40 |
| | 5 | 6.40 | 26.29 | 1.28 | 3.29 |
| | 10 | 6.30 | 46.51 | 1.26 | 5.81 |
| | 15 | 6.28 | 66.21 | 1.26 | 8.28 |
| 8 | 1 | 26.40 | 40.89 | 5.28 | 3.41 |
| | 2 | 21.53 | 40.83 | 4.30 | 3.40 |
| | 5 | 6.29 | 46.36 | 1.26 | 3.86 |
| | 10 | 6.18 | 86.53 | 1.24 | 7.21 |
| | 15 | 6.16 | 126.20 | 1.23 | 10.52 |

Table 1. Performance results for multiple-packet message transmissions (5-stage network (N=32) and 100% network load).

From Table 1, it can be seen that, for any given message size, as the packet offset is increased, the delay experienced by a packet in the network decreases and approaches the minimum traversal delay (normalized delay of 1). This is to be expected since the *apparent* network loading decreases as the packet offset is increased, thus reducing the network conflict delays that a packet will experience. The overall delay experienced by the entire message is also seen to decrease as the packet offset begins to increase from its initial value of 1, again a result of decreased network conflicts. However, as the packet offset continues to increase, the message delay time is seen to drop to a minimum level and then begin to increase. The increase is due to the longer delays between packet submissions that unnecessarily increase the message transmission delay. Packet offsets that lead to minimized overall message delays are in the range of 2 to 5. For example, message delays for two and four packet messages are minimized with a packet offset of 5, while an offset of 2 produces minimum delays in an eight-packet message. PUGS was also used to collect statistics for many different combinations of network operating parameters, such as: network size, number of packets per message, network loading, and packet offset values. In all cases, packet offsets in the area of 2-5 have been seen to produce the minimum message delays through the network.

In a typical system development process, the speed of the PEs is determined by the selection of a target microprocessor chip/chip set. The packet offset will then be fixed by the specification of the network interchange boxes and their packet cycle times. The analysis discussed above allows the most cost-effective tradeoff point for the network's hardware design to be predicted. The network's estimated performance over a range of cycle times can be used as a guide in determining the required hardware sophistication of the network design. The most desirable design, in terms of the cost-performance tradeoff, would be the slowest and typically the least expensive network that did not act as a bottleneck to the anticipated inter-PE message flow.

## 4. PACKET vs. CIRCUIT SWITCHED OPERATIONS

In inter-computer networks, the classic tradeoff between circuit and packet switching is one of message length. Long message formats are best supported by circuit switching whereas short, ''bursty" formats are ideally suited for packet switching. The design of these networks tends to neglect propagation delays through the switch points in the network and instead concentrates on the transmission channel (link) delays [21, 25]. The differences in the two switching methods are not as clear-cut in intra-computer networks found in parallel processing systems. Because the processors are generally ''close" physically, the interchange box delays dominate over the transmission link delays. System characteristics such as the operational mode (e.g., SIMD or MIMD), the architecture supporting the network, as well as the anticipated message format and system loading, greatly influence the selection of the switching method.

Systems that utilize the network solely for inter-PE data transfers (e.g., the PE-to-PE system architecture) can be characterized by lightly loaded, low message conflict networks. In this case, circuit switched networks can give very acceptable performances using circuitry that is much less complex than a comparable packet switched network (e.g., message queues and their control logic are not needed in the interchange boxes). Implementation of a circuit switched network is particularly advantageous in SIMD systems performing data permutations with no network conflicts where, once the path connecting the source-destination PE pair is established, the interchange boxes contribute only minimal gate delays to the transmission time.

In systems where the network is used to connect processors to memory modules (e.g., the processor-to-memory system architecture), inter-processor communication *and* instruction fetch operations utilize the network. The network loading would be expected to be high due to the preponderance of instruction and data fetch operations that must be supported. Packet switched networks have been proposed for this environment [8, 9]. Note that, through the use of cacheing techniques, the network loading can be substantially reduced and, in effect, giving a network performance similar to the PE-to-PE architecture's. PE-to-PE architectures will be assumed in this paper.

When packet switching is used for inter-PE transfers, if the message exceeds the packet length, multiple packets must be sent through the network. The processors must absorb the added overhead of message packetization (at the source PE) and packet recombination (at the destination PE). Each packet must independently perform its own routing/path establishment. These delays, coupled with the queueing delays at each switch, can increase the overall message transmission time in a packet switching network.

In this section, performance tradeoffs between packet and circuit switched interconnection networks will be explored. The analysis will be done using the same external system conditions in both switching modes. SIMD and MIMD operations will be considered separately.

### 4.1 SIMD operations

SIMD system operations that utilize interconnection network transfers generally employ data permutations, where data is transferred from a set of network sources to a set of network destinations using a particular transfer scheme. Cube-type networks have been found to perform most useful SIMD permutations in one pass through the network, with no conflicts [14, 18]. Permutations that cannot be performed without conflicts in a single pass can be performed in multiple, conflict free, passes through the network. As a result, the analysis for SIMD operations will assume that data transfers through the network will take place without network conflicts.

To enable direct comparisons of the performance of circuit and packet switched networks, a number of assumptions must be made concerning the operation of the computer system external to the interconnection network. These assumptions are listed below:

1. Data that is to be transferred through the network will originate from the same point within the source PEs (i.e., data for circuit switched transfers will not be in a PE's memory while the data for packet switched transfers is in the PE's registers). This ensures that the time overhead required by the source PEs to fetch the data will be the same for both switching modes.
2. The message transmission time will start when the first word of the message is generated by the source PE.
3. The overhead of computing the routing tag is not included in the message transmission time. In the case of multiple packet messages, the routing tag is computed only once (at the beginning of the message) and is stored for later use in subsequent packets.
4. Processing of data at the destination PEs will not degrade the performance of the network or impede the message generation process of the source PEs.
5. Packets must be fully formed at the source PE before they can be submitted to the network.

In the following discussions, assumptions are made about values for various network parameters to demonstrate the methodology developed. The framework established here can, of course, also be used to evaluate different systems by modifying the values of these parameters. The values used below are estimates developed from our design of the PASM system prototype [16].

Using circuit switching, the message transmission process can be decomposed into two phases -- set-up (or path establishment) and the actual data transfer. During the set-up phase, the message request (routing tag) must propagate through interchange boxes in each stage of the network and establish the desired data path. Let the time to establish a path through an interchange box be $(T_{cu} + T_{ib} + T_t)$. $T_{cu}$ is the time required by the box's control unit to check the routing tag and establish the needed box setting and $T_{ib}$ is the propagation delay through the interchange box's switching logic. $T_t$ is the link transmission time between interchange boxes in adjacent stages as well as between the PEs and the network input and output stages. From this, the path set-up time, $T_{setup}$, can be calculated and will require

$$n(T_{cu} + T_{ib} + T_t) + T_t + n(T_{ib} + T_t) + T_t \qquad (1)$$

time units to establish a path through an n-stage network. Note that the first two terms represent the total time required by the message request to traverse the network from the source to the destination. The second two terms represent the time required by a message grant signal to be returned from the destination to the source.

Once the path through the network has been established, the actual data transfer can begin. The delay experienced by a data word in traversing an n-stage network is

$$T_{delay} = 2(n{+}1)T_t + 2(n)T_{ib} \, . \qquad (2)$$

The factors of 2 represent the time to transmit the data word itself to the destination plus a returned data acknowledge signal generated by the destination PE. Let L be the total message length, excluding the routing information. Additionally, let $T_{generation}$ be the time required by a PE to fetch a data word from memory and send it to the network input port. The actual data transmission time, $T_{xmit}$, will be

$$T_{xmit} = \begin{cases} T_{generation} + L{\cdot}T_{delay}, & T_{delay} > T_{generation} \\ L{\cdot}T_{generation} + T_{delay}, & T_{delay} \le T_{generation} \end{cases} \qquad (3)$$

If the generation time of a word is less than the transmission delay of a word, the total delay will be the generation time of the first word plus L transmission delays. Under these conditions, the performance of a circuit-switched network could possibly be improved through the use of data pipelining (with data buffers placed between the network stages) or the use of a message-switched network design [12]. If, on the other hand, the generation time is greater than the transmission delay, the total time will be L generation delays plus the transmission delay of the last data word. In other words, if the transmission delay of the network is small with respect to time required by the source PEs to generate the data words, the overall data transmission time of a message will be determined by the speed of the PEs -- not the speed of the network. In these instances, circuit-switched network performance enhancements such as data pipelining will not improve the overall message transmission time. Hybrid techniques such as pipelining and message-switching will not be considered in this paper.

For example, the Motorola MC68010 microprocessor, operating at a clock frequency of 10 Mhz, is capable of performing one memory-to-memory data transfer in, at best, 800 ns (the network source and destination ports are considered to be memory addresses by the PE) [17]. If the 2-by-2 switching logic of the interchange boxes is implemented using multiplexers or by two levels of tri-state buffers, $T_{ib}$ can be expected to be approximately 20 ns (worst-case, TTL logic). $T_t$ can be assumed to be no more than 5 ns. A reasonable value for $T_{cu}$ is 100 ns. The total delay experienced by a data word would then be (from Equation 2)

$$\begin{aligned} \text{Network delay} &= 10(n{+}1) + 40n \qquad (4) \\ &= 50n + 10 \, . \end{aligned}$$

Clearly, for system and network sizes that are currently implementable, the speed of the PE will be the determining factor in the data transmission time. Using the representative timing values, listed above, the set-up time (from Equation 1) and the data transmission times can be combined into the overall message transmission time, $T_{message}$:

$$\begin{aligned} T_{message} &= T_{setup} + T_{xmit} \\ &= (150n + 10) \\ &\quad + L{\cdot}(\text{PE word generation rate}) \, . \qquad (5) \end{aligned}$$

Next, consider the operation of a packet switched network. Note that, because there are no conflicts within the network, there will be no queueing delays within the network. The size of the interchange box buffers will not effect the overall network operation. With the exception of the logic to form and control the interchange box buffers, the architecture and speed of a packet switched interchange box will be the same as that of the circuit switched box, discussed above. The values for $T_{cu}$, $T_{ib}$, and $T_t$ will be the same. Let $T_q$ be the time needed to access a buffer in an interchange box. Because the buffer will be physically close to the rest of the circuitry for the box, if not on the same VLSI chip, $T_q$ will be substantially less than the time required to perform a PE memory access. Assume that buffer reads and writes can be overlapped. The delay, $T_{packet}$, that a data packet would experience in traversing a packet switched box would be

$$T_{packet} = T_{cu} + (T_q + T_{ib} + T_t)(P{+}1) \, . \qquad (6)$$

The factor (P+1) is the *actual* packet size including the routing tag (P data words plus one routing word). Time required for handshaking between adjacent interchange boxes can be overlapped with the other phases of $T_{packet}$.

Using dual-ported memories for the packet buffers, $T_q$ would be at least 100 ns. Substituting this and the previously assumed values, Equation 6 then reduces to

$$T_{packet} = 125(P+1) + 100 \text{ ns} . \qquad (7)$$

The time required by a PE to generate a packet would be (P+1)(PE word generation time). Continuing with the MC68010 example, this would be, at best, 800(P+1) ns. The ratio of the packet generation time and $T_{packet}$, the packet offset, will be approximately 6. This leads to an alternate view of the packet offset time for SIMD operations.

When more than one packet is generated per message, if the number of stages in the network is less than or equal to the packet offset, a packet will completely traverse the network before the next packet in the message can be formed by the source PE and submitted to the network. The pipelining of message packets will not occur -- effectively eliminating one of the general advantages of packet switched networks. An L-word message, divided into $\lceil L/P \rceil$ data packets will have a transmission time of

$$T_{message} = \lceil L/P \rceil(\text{Packet generation time}) + n \cdot T_{packet} . \qquad (8)$$

The first term represents the delay between the generation of the first word in the message (contained in the first packet) and the time the last packet is submitted to the network. The second term is the time required by the last packet to traverse the network.

Table 2 lists the message transmission times for various sized messages using both packet and circuit switching modes. The message transfer times were calculated using the representative timing values for the interchange box components and PEs, discussed above. It is clear that, for SIMD operations, the circuit switched mode provides lower message transmission delays since the transmission of individual data words can be overlapped with the PEs' generation of the next data words to be transmitted. The overlap or pipeline performance normally associated with packet switched operations is precluded by the processing speed differential between the PEs and the packet switched network. From Table 2, it is also observed that packet sizes of two or four words yielded generally lower message transmission times than the other packet sizes. Small packet sizes incur an additional generation overhead of having to write the routing tag to many different packets (e.g., for a 4-word message, four 1-word packets with routing tags must be generated compared to only two 2-word packets and one 4-word packet). As the packet size increases, the packet delay time through an interchange box, $T_{packet}$, increases, as does the packet generation time. The combination of these two effects causes an increase in the total transmission delay through the network of each packet and negates the relative advantage of having to transmit fewer total packets for a given message size. The performance of the 2- and 4-word packets were a compromise between the two extremes.

## 4.2 MIMD operations

The performance analysis and comparison of circuit and packet switched networks presented in the last subsection was based on the assumption that network transfers would take place without conflicts. In MIMD operations, this assumption is not valid. Here, the performance of a network will be affected by conflicts within the network and the associated queueing delays.

For MIMD networks, the circuit switched message transmission process can still be decomposed into the set-up and the data transfer phases, as before. The conflict and blocking effects on the network's MIMD performance can be predicted using the Markov analysis of [15] or by simulation. The analysis of packet switched networks must also include the effects of queueing delays within the buffers associated with each interchange box. The simulation methodology described in Section 3 for multiple-packet messages can also be used here.

Any attempt to directly compare the performance of circuit and packet switched networks must use the same operational environment for both networks. External (to the network) factors such as the system size, processing speed of the PEs, and the network loading factor must be the same for both networks. Additionally, the fundamental design of the two networks must also be the same (e.g., data path width, the technology used in the interchange box implementations, and the PE-network interface methodology). When these internal and external factors are fixed, a comparison between the two switching methods can be made.

| Network size | Switching mode | Packet size | Data transfer size | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 4 | 8 | 16 |
| 4 stages | circuit | - | 1.41 | 2.21 | 3.81 | 7.01 | 13.41 |
| | packet | 1 | 3.00 | 4.60 | 7.80 | 14.20 | 27.00 |
| | | 2 | 4.30 | 4.30 | 6.70 | 11.50 | 21.10 |
| | | 4 | 6.90 | 6.90 | 6.90 | 10.90 | 18.90 |
| | | 8 | 12.10 | 12.10 | 12.10 | 12.10 | 19.30 |
| | | 16 | 22.50 | 22.50 | 22.50 | 22.50 | 22.50 |
| 5 stages | circuit | - | 1.56 | 2.36 | 3.96 | 7.16 | 13.56 |
| | packet | 1 | 3.35 | 4.95 | 8.15 | 14.55 | 27.35 |
| | | 2 | 4.78 | 4.78 | 7.18 | 11.98 | 21.56 |
| | | 4 | 7.63 | 7.63 | 7.63 | 11.63 | 20.53 |
| | | 8 | 13.33 | 13.33 | 13.33 | 13.33 | 20.53 |
| | | 16 | 24.73 | 24.73 | 24.73 | 24.73 | 24.73 |

Table 2. SIMD performance comparison of circuit and packet switched networks (times in microseconds, packet size does not include routing tag).

Assume the networks are constructed such that the timing values used in Section 4.1 remain valid. Specifically, $T_{cu} = 100\text{ns}$, $T_{ib} = 20\text{ns}$, $T_t = 5\text{ns}$, and $T_q = 100\text{ns}$. Furthermore, assume that the PEs are implemented using a clock frequency of 10 Mhz, as in an MC68010 microprocessor (the microprocessor requires 800ns to write one word from memory to the network). Under these assumed conditions, Tables 3 and 4 list simulation results for the message transmission times and throughput performances of circuit and packet switched networks for a range of loading values and message lengths. Comparing the data in these tables to that of Table 2, it can be seen that the MIMD performance of both network types approaches the SIMD performance values as the network loading is decreased (as would be expected). The performance relationship between the circuit switched conflict resolution algorithms is as predicted by [15] -- the drop algorithm provides generally lower transmission times and higher message throughputs than does the hold algorithm.

Over the range of message lengths considered in Tables 3 and 4, minimum message transmission times for a given data transfer size (message length) can be obtained in the packet switched network when the entire message is contained in only one or two packets. As in SIMD operations, the performance tradeoff is lighter network loading (number of packets) with larger packet sizes versus shorter packet transmission times of the individual packets with smaller packet sizes.

A representative plot of the message transmission times for both circuit and packet switched networks is shown in Figure 2. For each plot, the data used was taken from the conflict resolution algorithm (for circuit switching) or the packet size (for packet switching) that provided the lowest transmission times. The circuit switched network provided superior network performance for smaller message sizes while packet switching performed best with larger message sizes. This can be attributed to, in circuit switch-

| Switching mode | Packet size | Loading factor | Data transfer size | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 4 | 8 | 16 |
| Circuit (hold algorithm) | - | 100 | 2.21 | 4.40 | 7.78 | 15.05 | 27.55 |
| | | 50 | 2.11 | 4.18 | 7.73 | 14.81 | 27.47 |
| | | 10 | 1.63 | 3.57 | 7.21 | 14.52 | 27.31 |
| | | 5 | 1.44 | 3.07 | 6.89 | 14.48 | 27.06 |
| Circuit (drop algorithm) | - | 100 | 2.25 | 3.96 | 7.36 | 13.74 | 24.41 |
| | | 50 | 2.22 | 3.95 | 7.27 | 13.64 | 23.81 |
| | | 10 | 1.73 | 3.53 | 7.00 | 13.63 | 23.64 |
| | | 5 | 1.44 | 3.07 | 6.39 | 13.08 | 23.47 |
| Packet | 1 | 100 | 6.54 | 5.99 | 10.27 | 18.63 | 35.42 |
| | | 50 | 4.26 | 5.74 | 9.92 | 18.30 | 35.10 |
| | | 10 | 3.54 | 5.66 | 9.88 | 18.27 | 35.08 |
| | | 5 | 3.52 | 5.65 | 9.85 | 18.26 | 35.06 |
| Packet | 2 | 100 | 8.89 | | 8.13 | 13.94 | 25.28 |
| | | 50 | 5.78 | | 7.79 | 13.46 | 24.84 |
| | | 10 | 4.81 | | 7.68 | 13.40 | 24.79 |
| | | 5 | 4.79 | | 7.66 | 13.36 | 24.78 |
| Packet | 4 | 100 | 13.55 | | | 12.40 | 21.27 |
| | | 50 | 8.83 | | | 11.90 | 20.55 |
| | | 10 | 7.34 | | | 11.73 | 20.46 |
| | | 5 | 7.29 | | | 11.69 | 20.40 |
| Packet | 8 | 100 | 21.76 | | | | 21.05 |
| | | 50 | 14.98 | | | | 20.18 |
| | | 10 | 12.45 | | | | 19.90 |
| | | 5 | 12.37 | | | | 19.84 |
| Packet | 16 | 100 | 39.45 | | | | |
| | | 50 | 27.16 | | | | |
| | | 10 | 22.57 | | | | |
| | | 5 | 22.51 | | | | |

Table 3. MIMD message transmission times for circuit and packet switched networks (4-stage network (N=16), delays given in microseconds).

| Switching mode | Packet size | Loading factor | Data transfer size | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 4 | 8 | 16 |
| Circuit (hold algorithm) | - | 100 | 7.20 | 3.68 | 2.08 | 1.04 | 0.48 |
| | | 50 | 7.12 | 3.68 | 2.00 | 1.04 | 0.48 |
| | | 10 | 5.84 | 3.44 | 1.92 | 0.96 | 0.48 |
| | | 5 | 4.40 | 2.96 | 1.68 | 0.96 | 0.48 |
| Circuit (drop algorithm) | - | 100 | 7.04 | 4.00 | 2.24 | 1.20 | 0.64 |
| | | 50 | 6.80 | 3.92 | 2.16 | 1.12 | 0.56 |
| | | 10 | 5.60 | 3.44 | 2.00 | 1.04 | 0.56 |
| | | 5 | 4.40 | 2.96 | 1.84 | 1.04 | 0.56 |
| Packet | 1 | 100 | 26.37 | 6.54 | 2.40 | 1.06 | 0.45 |
| | | 50 | 22.63 | 5.71 | 2.29 | 1.03 | 0.48 |
| | | 10 | 4.60 | 2.89 | 1.66 | 0.89 | 0.46 |
| | | 5 | 2.37 | 1.80 | 1.23 | 0.74 | 0.43 |
| Packet | 2 | 100 | 19.43 | | 4.82 | 1.77 | 0.78 |
| | | 50 | 16.67 | | 4.21 | 1.68 | 0.76 |
| | | 10 | 3.39 | | 2.13 | 1.22 | 0.65 |
| | | 5 | 1.75 | | 1.33 | 0.91 | 0.55 |
| Packet | 4 | 100 | 12.73 | | | 3.16 | 1.16 |
| | | 50 | 10.92 | | | 2.76 | 1.10 |
| | | 10 | 2.72 | | | 1.39 | 0.80 |
| | | 5 | 1.44 | | | 0.87 | 0.59 |
| Packet | 8 | 100 | 7.50 | | | | 1.86 |
| | | 50 | 6.47 | | | | 1.63 |
| | | 10 | 1.31 | | | | 0.82 |
| | | 5 | 0.68 | | | | 0.51 |
| Packet | 16 | 100 | 4.14 | | | | |
| | | 50 | 3.56 | | | | |
| | | 10 | 0.72 | | | | |
| | | 5 | 0.37 | | | | |

Table 4. MIMD message throughputs for circuit and packet switched networks (4-stage network (N=16), throughputs expressed as the number of messages processed per microsecond).

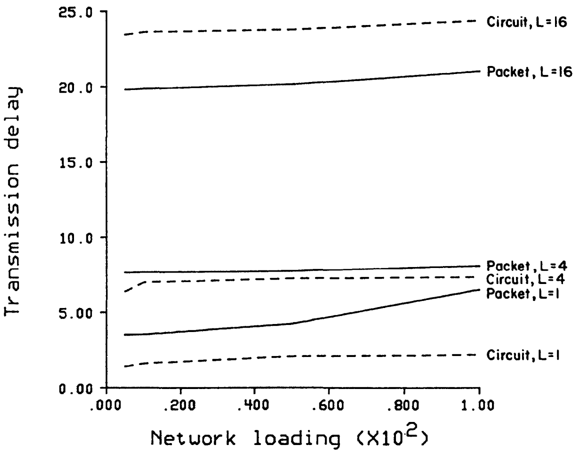


Figure 2. Comparison of message transmission delays for both circuit and packet switched networks (4-stage network (N=16), 100% network loading, times in microseconds).

ing, significantly higher conflict rates and blocking times that result when the network paths are held for periods of time corresponding to the transmission of "long" messages, as reported in [3, 15]. This does not occur in packet switched networks. In contrast, packet switched networks tend to perform better with longer messages, where the overhead of the packet generation (i.e., the routing tag) is proportionately smaller (when compared to the actual message transmission time) and the delays due to network conflicts do not increase as fast as in circuit switched networks. For small message transfers, the blocking delays within a circuit switched network will be reduced, thus producing good overall message delay characteristics. This, coupled with the relatively high overhead of packet generation and queueing delays experienced by packet switched messages (when compared to the transmission time), enables circuit switched networks to function better than packet switched networks for small message lengths.

## 5. CONCLUSIONS

This paper has centered on the modeling and analysis of multiple-packet message formats and their relative performance advantages when compared to circuit switched networks. A multiple-packet message format is needed within an interconnection network when the message to be transmitted exceeds the network's packet size. In order to quantize network performance, the operation of interchange boxes, networks, and external systems were expressed as time functions. General equations relating these time functions to the performance of a network were developed. By using these equations and the PUGS simulator, the effects of any set of internal and external environment assumptions could be evaluated and network performance -- either packet or circuit switched -- could be predicted.

During the analysis of packet switched networks, the effects of various packet sizes was examined. The packet offset figure of merit was introduced and used as a performance gauge of the transmission rate differential between the network and the PEs. It was observed that the optimum network performance (in terms of low transmission delays and reasonable network component costs) could be obtained with a packet offset in the range of 2 to 5.

Comparing the performance of circuit and packet switched networks for a specific set of time function values, circuit switching was shown to provide minimum transmission delays in all

SIMD operations and in short message transfers in MIMD operations. Packet switched networks functioned better than circuit switched networks as the message length increased. The dominant element of the network performance was seen to be the processing rate of the PEs themselves.

A general methodology has been developed that quantized system and network parameters, thus allowing the performance of a network design to be evaluated under a set of anticipated operating conditions. By adjusting the assumed values of these parameters, this methodology can be used by other researchers as an aid in the design of their own networks.

## 6. REFERENCES

[1] K. E. Batcher, "The flip network in STARAN," *1976 International Conference on Parallel Processing*, August 1976, pp. 65-71.

[2] P-Y. Chen, J. E. Lilienkamp, and D. H. Lawrie, *Performance of Circuit Switched Multistage Interconnection Networks in Multiprocessing Systems*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[3] P-Y. Chen, *Multiprocessor Systems: Interconnection Networks, Memory Hierarchy, Modeling, and Simulations*, Report Number UIUCDCS-R-82-1083, Department of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[4] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," *1985 International Conference on Parallel Processing*, August 1985, pp. 531-540.

[5] N. J. Davis IV, J. Ott, H. J. Siegel, and A. L. Overvig, *Simulation Studies of the Generalized Cube Interconnection Network*, Technical Report, School of Electrical Engineering, Purdue University, to appear.

[6] N. J. Davis IV and H. J. Siegel, "The performance analysis of partitioned circuit switched multistage interconnection networks," *Twelfth Annual Symposium on Computer Architecture*, June 1985, pp. 387-394.

[7] J. B. Dennis, "Data flow supercomputers," *Computer*, Vol. 13, November 1980, pp. 48-56.

[8] D. M. Dias and J. R. Jump, "Analysis and simulation of buffered delta networks," *IEEE Transactions on Computers*, Vol. C-30, April 1981, pp. 273-282.

[9] D. M. Dias and J. R. Jump, "Packet switching interconnection networks for modular systems," *IEEE Computer*, Vol. 14, December 1981, pp. 43-53.

[10] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing an MIMD shared-memory parallel computer," *IEEE Transactions on Computers*, Vol. C-32, February 1983, pp. 175-189.

[11] G. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *First Annual Symposium on Computer Architecture*, December 1973, pp. 21-28.

[12] P. Kermani and L. Kleinrock, "A tradeoff study of switching systems in computer communication networks," *IEEE Transactions on Computers*, Vol. C-29, December 1980, pp. 1052-1060.

[13] C. P. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Transactions on Computers*, Vol. C-32, December 1983, pp. 1091-1098.

[14] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, Vol. C-24, December 1975, pp. 1145-1155.

[15] M. Lee and C-L. Wu, "Performance analysis of circuit switching baseline interconnection networks," *Eleventh Annual Symposium on Computer Architecture*, June 1984, pp. 82-90.

[16] D. G. Meyer, H. J. Siegel, T. Schwederski, N. J. Davis IV, and J. T. Kuehn, "The PASM parallel system prototype," *IEEE Computer Society Spring Compcon 85*, February 1985, pp. 429-434.

[17] Motorola, Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual (fourth edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[18] M. C. Pease III, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, Vol. C-26, May 1977, pp. 458-473.

[19] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," *1985 International Conference on Parallel Processing*, August 1985, pp. 764-771.

[20] U. V. Premkumar, R. N. Kapur, M. Malek, G. J. Lipovski, and P. Horne, "Design and implementation of the banyan interconnection network in TRAC," *AFIPS 1980 National Computer Conference*, June 1980, pp. 643-653.

[21] M. Schwartz, *Computer-Communication Network Design and Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[22] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer*, Vol. 14, December 1981, pp. 65-76.

[23] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Transactions on Computers*, Vol. C-30, December 1981, pp. 934-947.

[24] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, D C Heath & Co., Lexington, MA, 1985.

[25] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[26] C-L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Transactions on Computers*, Vol. C-29, August 1980, pp. 694-702.

# ANALYSIS OF CC-BANYAN NETWORKS

Vladimir Cherkassky
Department of Electrical Engineering, University of Minnesota
Minneapolis, MN 55455

Miroslaw Malek
Department of Electrical Engineering, University of Texas
Austin, TX 78712

ABSTRACT: A reconfigurable multicomputer architecture based on rectangular CC-banyan multistage interconnection network is presented. A graph-theoretical approach is used to study this network's permuting and structural properties. It is shown that a CC-banyan has a modular structure and hence can be recursively defined. A method for evaluation of the total number of permutations in CC-banyans is presented. Using this method, we derive the analytical expressions for the number of permutations in CC-banyans with fanouts two and three.

## 1. INTRODUCTION

There is growing interest in multicomputer systems for increased performance and improved reliability. The operation of these systems is critically dependent on the interconnection network which connects system resources such as processors and memories (or computers). As a result of the rapid advances in LSI and VLSI technology, extensive research is being conducted on multicomputer systems. Many of these systems are based on multistage interconnection networks (MINs).

Various MINs implemented with 2x2 switching elements, such as baseline, modified data manipulator, flip, omega, indirect binary n-cube and regular rectangular SW-banyan (s=f=2) are known to be topologically equivalent to each other [1]. These equivalent networks possess a "buddy property," i.e., the outputs of two switching elements at stage i are connected as inputs to only two switching elements at the (i+1)th stage [2].

Banyans [3] represent a large class of interconnection networks. An SW-banyan with fanout two describes a class of equivalent MINs implemented with 2x2 switching elements. Also, many non-equivalent MINs [4] represent special cases of cylindrical crosshatch (CC) banyans [3]. It has been shown in [5] that the so-called 'non-equivalent' delta network [6] is isomorphic to a CC-banyan, and that the ADM network [7] can be viewed as two overlapping CC-banyans. Permuting and partitioning properties of SW-banyans and of equivalent MINs have been extensively studied elsewhere [2,3,8,9, 10,11]. In this paper we focus on non-equivalent networks. We analyze the recursive structure and evaluate the total number of permutations supported by a CC-banyan. Detailed derivations of results presented in this paper can be found in [12].

## 2. SYSTEM DESCRIPTION

The system model is composed of the resources and the interconnection network. In this paper we use the rectangular CC-banyan interconnection network to study system mapping properties.

In the following we give a brief description of CC-banyans. A banyan network [3] is a network with a unique path from each source to each sink vertex. A multistage interconnection network is a network in which vertices can be arranged in stages, with all the source vertices at stage (level) 0, and all the outputs at stage i connected to inputs at stage i + 1. An L-level banyan, or an L-stage MIN, is a network in which every path from any source (base) to any sink (apex) has length L. An (f,L) banyan is an L-level banyan in which the indegree (spread-s) of every intermediate vertex equals its outdegree (fanout-f).

(f,L) regular rectangular CC-banyan: An (f,L) banyan in which there is an edge from vertex i at vertex level k+1 to a vertex j at vertex level k whenever $j=(i+mf^k) \bmod f^L$ for m = 0, 1, 2,...,f-1, $0 \leq i, j \leq f^L-1$ and $0 \leq k \leq L-1$.

Examples of (3,2) and (2,3) regular rectangular CC-banyans are shown in Fig. 1 and 2, respectively. The vertex levels are numbered from bases (level 0) to apexes (level L). The interpretation of the network graph is the one commonly used with respect to banyan networks: a vertex represents a tie point and an edge represents a switch contact. According to this interpretation, connections through vertices in a banyan graph are mutually exclusive.

An example of a multicomputer system based on a rectangular CC-banyan network is shown in Fig. 2. In this system each computer (processor with local memory) is attached to both sides of an (f,L) CC-banyan. A similar approach utilizing a different network topology but similar connection of processing nodes to both sides of the MIN is used in the Star network [13]. The communication through the network is unidirectional and the control is based on the routing tag technique. The labeling and routing schemes for a system based on a CC-banyan are discussed in [14]. We just note here that the routing scheme in CC-banyan leads to relative addressing, based on the difference between destination and source address, in contrast to the systems based on SW-banyans which use absolute addressing.

## 3. RECURSIVE STRUCTURE OF CC-BANYAN

Whereas the original definition of CC-banyans is not recursive, it will be shown here that CC-banyans can be recursively defined. Conceptually, it is convenient to view an (f,L) rectangular CC-banyan as

This work was supported in part by the Graduate School, University of Minnesota.

a union of L stages, where each stage represents a bipartite graph [4]. For example, three bipartite graphs for (2,3) rectangular CC-banyan are shown in Fig. 3a.

Theorem 1: In an (f,L) rectangular CC-banyan a bipartite graph of stage k ($0 \le k \le L-1$) contains $f^k$ identical components (disjoint subgraphs).

Different components of a (2,3) CC-banyan are shown in Fig. 3b.

Corollary 1: In an (f,L) rectangular CC-banyan, the bipartite graphs of all L stages are not isomorphic (to each other).

Note that in the case of equivalent MINs (e.g., baseline, omega) the bipartite graphs of all stages are isomorphic, since these MINs have the "buddy property" [2].

Corollary 2: Each component of stage k ($0 \le k \le L-1$) has $2*f^{L-k}$ vertices.

Corollary 3: In an (f,L) rectangular CC-banyan the bipartite graph of stage-0 at the base side contains one component; and the bipartite graph of stage-(L-1) at the apex side contains $f^{L-1}$ components, which are (fxf) crossbar graphs.

Note that the recursive decomposition of a CC-banyan into networks of smaller size may be only possible from the base side (due to Corollary 3). In contrast to CC-banyans, SW-banyans and networks equivalent to them can be recursively decomposed from either side.

Theorem 2: An (f,L+1) CC-banyan can be recursively decomposed from the base side into f disjoint (f,L) CC-banyans.

Based on Theorem 2, we give the following recursive definition.

Definition: An (f,1) CC-banyan is simply an (fxf) crossbar graph. An (f,L+1) CC-banyan is constructed from f (f,L) CC-banyans by the following procedure:

1. Assign a number m (m=0,1,...f-1) to each (f,L) CC-banyan.
2. In the m-th (f,L) CC-banyan, label the vertices at every vertex level by the numbers
   $m+jf$ , where $j = 0,1,2,\dots,f^L$
   (see Fig. 4a).
3. Form stage-0 (base stage) of the composite (f,L+1) CC-banyan by connecting the bases numbered $i(i=0,1,\dots,f^{L+1}-1)$ to the bases of the component (f,L) CC-banyans numbered $(i+mf)\bmod f^{L+1}$ for all $m=0,1,\dots,f-1$ (see Fig. 4b).
4. Rearrange the vertices of the composite (f,L+1) CC-banyan in the order of increasing label numbers, thus resulting in the conventional representation (see Fig. 4c).

## 4. PERMUTING PROPERTIES

In this section we evaluate the number of various permutations performable by an (f,L) CC-banyan in one pass. We view an (f,L) rectangular CC-banyan as a permutation network that performs a one-to-one mapping (permutation) of a set of $N(N=f^L)$ apexes onto a set of N bases.

Since in a banyan network every input-output connection (base-apex path) is unique, the total number of permutations is equal to the product of all permutations performable by each stage:

$$P = \prod_{k=0}^{L-1} P_k \qquad (1)$$

where $P_k$ is the number of stage-k permutations.

In view of Theorem 1 and Corollary 3,

$$P = (f!)^{f^{L-1}} * \prod_{k=0}^{L-2} (C_{f,k})^{f^k} \qquad (2)$$

Here $C_{f,k}$ denotes the number of permutations performable by each of $f^k$ disjoint subgraphs of stage k. The indices indicate that this number, generally, depends on both fanout f and stage k. An exact analytical evaluation for $C_{f,k}$ does not seem feasible for arbitrary fanout f. However, we present a general approach which enables systematic enumeration of all permutations. Our analysis uses unconventional representation for a stage-k subgraph, in which the vertices of a bipartite subgraph are located on the two concentric circles corresponding to the vertex levels k and k+1. An example of such symmetric circular representation, or circular diagram, is shown in Fig. 5. Consider circular diagram of a stage-k subgraph. As follows from Corollary 2, this subgraph performs permutations on $f^{L-k}$ numbers. A permutation in this subgraph is a one-to-one mapping of its level-(k+1) vertices onto its level-k vertices.

A stage-k circular diagram may be divided into radial sectors of equal size (f-1) using radial cuts as shown in Fig. 5. The radial cuts and radial sectors are numbered clockwise from 0 to $(f^{L-k}-1)/(f-1)$, so that a radial sector m is formed by radial cuts m and m+1, $m = 0,1,\dots, (f^{L-k}-1)/(f-1)$. Then each of the sectors has (f-1) vertices, and the last sector has only one vertex. Consider a set of edges corresponding to a permutation in a circular diagram. An edge corresponding to a permutation is called a cross edge if it crosses a radial cut in a circular diagram. Now we can state an important invariant property of permutations in CC-banyans.

Theorem 3: For any stage-k subgraph permutation in a circular diagram, all radial cuts have the same number of cross edges i ($0 \le i \le f-1$).

An example of two different permutations and corresponding cross edges in a circular diagram is given in Fig. 5.

Theorem 4: In a circular diagram of stage-k subgraph, there exists only one permutation corresponding to 0 or (f-1) cross edges:

$$C^{0}_{f,k} = C^{f-1}_{f,k} = 1.$$

Specifically, the identity permutation has 0 cross edges, and the cyclic shift $i \rightarrow (i+f-1)\bmod f^{L-k}$ has f-1 cross edges (see Fig. 5). Theorem 4 also implies that in a CC-banyan with fanout f=2 all stage-k subgraphs can perform exactly 2 permutations, regardless of k.

Theorem 5: The total number of permutations performable by rectangular CC-banyan with fanout f=2 and number of stages L is:

$$P = 2^{2^L-1}$$

A systematic enumeration of all permutations in the case of arbitrary f uses a "divide-and-conquer" approach based on Theorems 1 and 3. Specifically, the number of permutations performable by a stage-k subgraph can be represented as

$$C_{f,k} = \sum_{i=0}^{f-1} C^{i}_{f,k}$$

where $C^{i}_{f,k}$ is the number of permutations which have exactly i cross edges.

In view of Theorem 4, the above expression is simplified to:

$$C_{f,k} = 2 + \sum_{i=1}^{f-2} C^{i}_{f,k} \qquad (3)$$

The value of $C^{i}_{f,k}$ $(1 < i < f - 2)$ can be found by enumeration of all permutations having the same number of cross edges i. A method for a systematic enumeration of such permutations is discussed below. Consider a set of permutations having the same number of cross edges $i(1 < i < f - 2)$. We may enumerate all permutations in a circular diagram by choosing, successively, i cross edges for every radial cut m, m=0, 1,2,..., $(f^{L-k}-1)/(f-1)$. Since i cross edges in a radial cut (m+1) can be chosen independently from i cross edges in a preceding cut m, we can construct a tree of all possible outcomes (decision tree) for a systematic enumeration of all permutations. We illustrate this method using an example of CC-banyan with fanout f=3. Only the case when the number of cross edges i=1 has to be considered, since $C^{0}_{3,k} = C^{2}_{3,k} = 1$. In this decision tree, a tree level corresponds to the radial cut number (see Fig. 6). Since every sector has f-1=2 vertices, a vertex incident to a cross edge can be chosen by 2 different ways, and the decision tree fanout is two. If the vertices in each sector are numbered from 0 to f-2, as shown in Fig. 5, then the vertex $m(0 < m < f-2)$ can be incident to f-1-m=2-m different cross edges. Therefore, two different cross edges can be chosen for vertex m=0, and only one cross edge can be chosen if m=1. Hence, we assign the weights 1 and 2 correspondingly to every "right" and "left" outgoing link in the decision tree. The total number of permutations can be found by adding the weights of all leaves where the weight of each leaf is a product of all link weights along the path from root to this leaf. Since we may choose outgoing links (with weights 1 or 2) independently at every tree level, the total weight represents a binomial sum:

$$\sum_{j=0}^{h} \binom{h}{j} 2^{j} * 1^{h-j} = 3^{h}$$

where h is the decision tree height, or the number of radial cuts, whose i cross edges can be chosen independently:

$$h = (f^{L-k}-1)/(f-1)$$

Therefore, the number of permutations having the number of cross edges i=1 in a stage-k subgraph of an (3,L) CC-banyan is:

$$C^{1}_{3,k} = 3^{(3^{L-k}-1)/2} \quad , \quad 0 < k < L-2 \qquad (4)$$

The total number of permutations in a (3,L) CC-banyan follows immediately from (2), (3) and (4).

Theorem 6: The number of permutations in an (f,L) CC-banyan with fanout f=3 is:

$$P = (3!)^{3^{L-1}} * \prod_{k=0}^{L-2} [2+3^{(3^{L-k}-1)/2}]^{3^{k}}$$

The method of decision trees can also be used for enumeration of all permutations in the case of general fanout f. However, the process of constructing decision trees becomes quite complex for the big values of f.

## 5. CONCLUSIONS

In this paper we presented a graph-theoretic approach to the analysis of rectangular CC-banyan net works with an arbitrary fanout f and an arbitrary number of stages L. It is shown that CC-banyans, like many other networks can be constructed recursively. This recursiveness enables the modular structure of CC-banyans and can be used in the analysis of its partitioning properties.

We presented a general approach to the analysis of permuting properties of CC-banyans. The analytical expressions for the number of permutations performable by CC-banyan with fanouts 2 and 3 are derived. In the case of general fanout f, we propose a method, based on decision tree analysis, for a systematic enumeration of all permutations.

## REFERENCES

1. C.-L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," IEEE Trans. Comp., Aug. 1980, pp. 694-702.
2. D. P. Agrawal, "Graph Theoretical analysis and design of multistage interconnection networks," IEEE Trans. Comp., pp. 637-648., July 1983.
3. L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor system," Proc. 1st Symp. on Comp. Arch., Dec. 1973, pp. 21-28.
4. D. P. Agrawal and S.-C. Kim, "On non-equivalent multistage interconnection networks," Proc. 1981 ICPP, Aug. 1981, pp. 234-237.
5. V. Cherkassky and E. Opper, "Fault diagnosis and permuting properties of CC-banyan networks," Proc. Real-Time Systems Symp., Dec 1984, pp. 175-183.
6. J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," IEEE Trans. Comp., Oct. 1981, pp. 771-770.
7. R. J. McMillen and H. J. Siegel, "Routing schemes for the Augmented Data Manipulator network in MIMD system," IEEE Trans. Comp., Dec. 1982, pp. 1202-1214.
8. U. V. Premkumar and J. C. Browne, "Resource allocation in rectangular SW-banyans," Proc. of the 9th Symp. on Comp. Arch., April 1982, pp. 326-333.
9. V. Cherkassky and M. Malek, "On permuting properties of regular rectangular SW-banyans," IEEE Trans. Comp., June 1985, pp. 542-546.
10. H. J. Siegel, "The theory underlying the partitioning of permutation networks," IEEE Trans. Comp., Sept. 1980, pp. 791-801.
11. C. Wu and T. Feng, "The reverse-exchange interconnection network," IEEE Trans. Comp., Sept. 1980, pp. 801-811.
12. V. Cherkassky, "Performance and reliability evaluation of banyan networks," Ph.D. Diss., Univ. of Texas at Austin, 1985.
13. W. Lin and C.-L. Wu, "Design of configuration algorithms of commonly used topologies for a multiprocessor-Star," Proc. 1985 ICPP, pp. 734-741.
14. E. Opper, M. Malek and G. J. Lipovski, "Resource allocation in rectangular CC-banyans," Proc. of the 10th Symp. on Comp. Arch., June 1983, pp. 178-184.

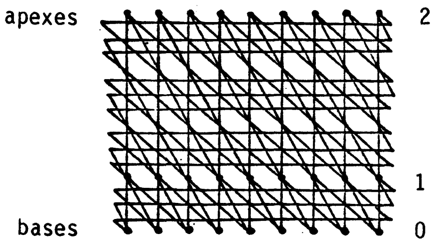
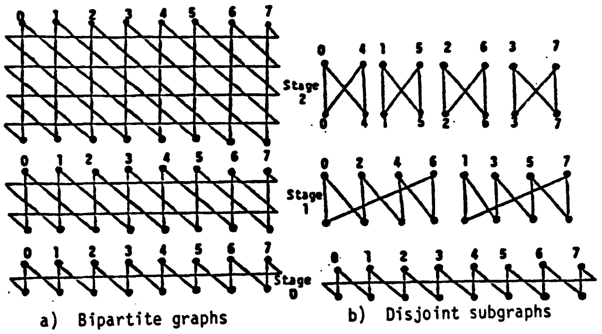


Fig. 1 (3,2) CC-banyan.



a) Bipartite graphs b) Disjoint subgraphs

Fig. 3 Bipartite graphs and disjoint subgraphs of (2,3) CC-banyan.

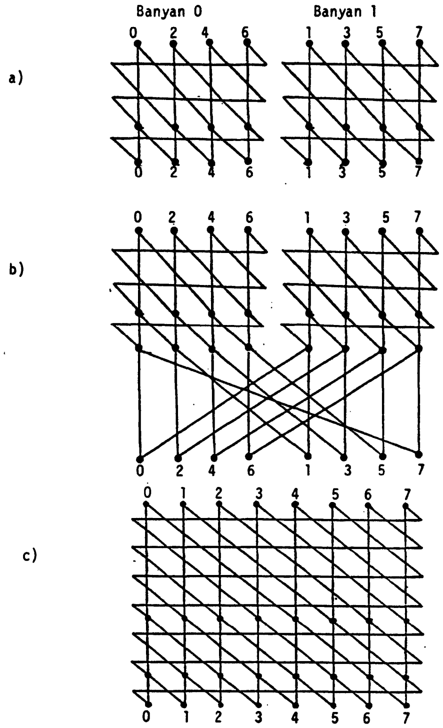


Fig. 4 Recursive construction of a (2,3) CC-banyan.

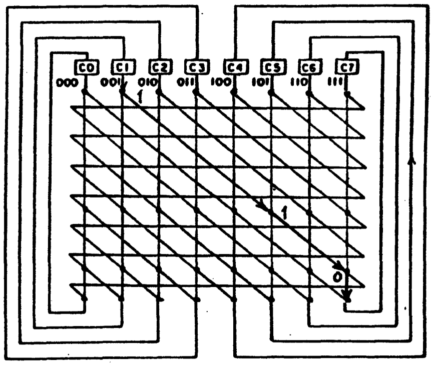
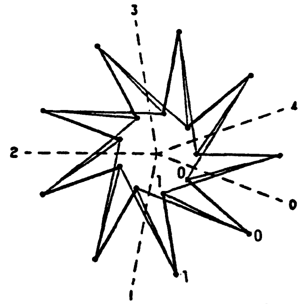
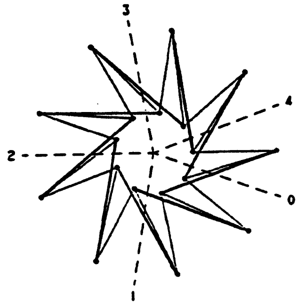


Fig. 2 Multicomputer system based on (2,3) CC-banyan with an example of routing.



a) identity permutation i→i (0 cross edges)



b) cyclic shift i→i+2 (2 cross edges)

Fig. 5 Circular diagram of stage-0 of (3,2) CC-banyan with examples of permutations.

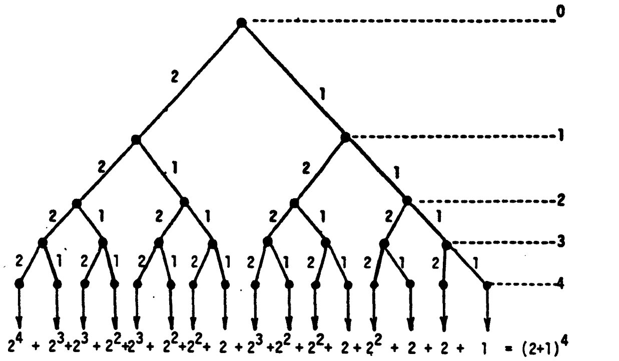


Fig. 6 Decision tree to enumerate all permutations with the number of cross edges i=1.

# Rearrangeability of the 5-Stage Shuffle/Exchange Network for N = 8 *

**C. S. Raghavendra and A. Varma****
**Department of Electrical Engineering– Systems**
**University of Southern California**
**Los Angeles, CA 90089-0781**

## ABSTRACT

In this paper we prove the rearrangeability of a multistage shuffle/exchange network with 8 inputs and outputs consisting of five stages. A lower bound of $(2 \log_2 N - 1)$ stages for rearrangeability of a shuffle/exchange network with $N = 2^n$ inputs and outputs is known; we show its sufficiency for $N = 8$. We not only prove the rearrangeability, but also describe an algorithm for routing arbitrary permutations on the network and prove its correctness. In contrast to previous efforts to prove rearrangeability, which rely on topological equivalence to the Benes class of rearrangeable networks, our approach is based on first principles. We also show that two switches in the network are redundant. The results in this paper are useful for establishing an upper bound of $(3 \log_2 N - 4)$ stages for rearrangeability of a multistage shuffle/exchange network with $N \geq 8$, as demonstrated in [14].

## 1. INTRODUCTION

Shuffle/Exchange networks, initially proposed by Stone [13] have been the subject of extensive treatment by several researchers *(for a survey on shuffle/exchange networks, see* [3]). A shuffle/exchange network with $N = 2^n$ inputs and outputs consists of a perfect shuffle permutation [13] followed by an exchange stage with $N/2$ switches, each of size 2 x 2. A multistage shuffle/exchange network is constructed by cascading multiple shuffle/exchange stages in series. A multistage shuffle/exchange network for $N = 8$ with five stages is shown in Figure 1.

Interconnection networks with the capability of passing all the $N!$ permutations on $N$ elements in one pass through the network are known as *rearrangeable networks* [2]. The Omega network [5], which is a multistage shuffle/exchange network with $\log_2 N$ stages, can perform only a small fraction of the $N!$ permutations when $N$ is large. However, the permutation capability can be improved by adding more stages.

The universality of shuffle/exchange networks was studied by researchers in an attempt to determine how many stages are required to attain the capability of performing arbitrary permutations, *ie.,* rearrangeability [4, 7, 10, 11, 12, 14, 16]. An asymptotic lower bound of $(2 \log_2 N - 1)$ stages of 2 x 2 switching elements for rearrangeability has been known for long [15]; the validity of this lower bound for shuffle/exchange networks can be established for any $N = 2^n$ by showing the existence of permutations which cannot be performed with less than $(2 \log_2 N - 1)$ stages [14]. The sufficiency of this many stages for shuffle/exchange networks has neither been proved nor disproved. Stone observed that, by using an algorithm due to Batcher [1], sorting of arbitrary sequences of data can be performed by cycling through a single shuffle/exchange stage $(\log_2 N)^2$ times when the switching elements are replaced by 2-input, 2-output sorting elements [13]. This algorithm can be used to map arbitrary permutations on a multistage network with $(\log_2 N)^2$ shuffle/exchange stages. An algorithm for performing arbitrary permutations on a single-stage shuffle/exchange network in $2(\log_2 N)^2$ passes was also described by Siegel [10], and the number of passes was subsequently improved to $(3/2)(\log_2 N)^2 - (\log_2 N)/2$[11]. Parker improved this bound by showing that $3 \log_2 N$ stages are sufficient for rearrangeability [7], though he did not specify a control algorithm for finding the states of the switching elements for arbitrary permutations. Wu and Feng observed that $(3 \log_2 N - 1)$ stages are indeed sufficient for rearrangeability [16]; they also showed how to compute the switch settings for arbitrary permutations. Kothari *et al.* observed that $(3 \log_2 N - 3)$ stages are sufficient for $N = 16$ and 32 [4]. More recently, Varma and Raghavendra showed that, if the rearrangeability of $(2 \log_2 R - 1)$ stages can be proved for a shuffle/exchange network of size $R = 2^r$, then $3 \log_2 N - (r + 1)$ stages are indeed sufficient for rearrangeability of a network of size $N > R$ [14].

For networks of small size, it may be possible to use exhaustive techniques to prove or disprove rearrangeability. Notice that, for $N = 4$, a cascade of three shuffle/exchange stages is identical to the Benes binary network and hence rearrangeable. Parker showed, by exhaustive enumeration, that 5 stages of shuffle/exchange are sufficient for rearrangeability when $N = 8$ [8], thereby suggesting that the limit of $(2 \log_2 N - 1)$ may be reachable, at least for small networks. In this paper we provide a constructive proof for showing that the lower bound of $(2 \log_2 N - 1)$ stages for rearrangeability of a multistage shuffle/exchange network is also an upper bound for $N = 8$ by establishing the rearrangeability of the network in Figure 1. As opposed to previous efforts to prove rearrangeability, which relied on reduction of the network to the topology of the Benes rearrangeable network, our approach is based on first principles. It is possible, in fact, to show that the network in Figure 1 cannot be reduced to a topology equivalent to that of the Benes network. We also describe an algorithm for routing arbitrary permutations on the network and prove its correctness. When used in conjunction with the results in [14], this establishes an upper bound of $(3 \log_2 N - 4)$ stages for rearrangeability of a multistage shuffle/exchange network for all $N \geq 8$. Our analysis also shows that two switches in the net-

* This research is supported by the NSF Presidential Young Investigator Award No. ECS-8452003, an AT&T Grant, and DARPA/ARO Contract No. DAAG 29-84-K-0066.

** The author is currently with IBM Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598.

work are redundant and could be eliminated without affecting the rearrangeability (Note that the Benes binary network of size 8 has three such switches).

## 2. PARTITIONING OF PERMUTATIONS

The procedure for routing arbitrary permutations on the 5-stage network is based on partitioning the connections of the permutation into four connection sets, consisting of two connections each, and assigning each connection set to one of the switches in the middle stage of the network. This partitioning is done such that no conflicts arise in either the first two stages or the last two stages of the network.

*Definition* 1: Let $\pi$ be a permutation on the set of integers $\{0,1, \dots ,7\}$ . A connection set $C$ of $\pi$ is a collection of $k$ input–output pairs of $\pi$, $k \leq 8$, expressed as

$$C = \left\{ \binom{x^1}{\pi(x^1)}, \binom{x^2}{\pi(x^2)}, \dots, \binom{x^k}{\pi(x^k)} \right\}, \tag{1}$$

where $\{x^1,x^2, \dots ,x^k\} \subseteq \{0,1, \dots ,7\}$. If $k = 8$, then $C$ defines the whole permutation.

*Definition* 2: The set of input terminals of a connection set $C$ , denoted by $C^-$, is the set of all inputs $x$, $0 \leq x \leq 7$, such that the pair $\binom{x}{\pi(x)}$ belongs to $C$. Formally,

$$C^- = \left\{ x \mid \binom{x}{\pi(x)} \in C \right\} \tag{2}$$

Similarly, the set of output terminals of $C$, denoted by $C^+$, is the set of output terminals of the pairs in $C$.

$$C^+ = \{\pi(x) \mid x \in C^-\} \tag{3}$$

*Definition* 3: A valid partition of a permutation $\pi$ is a partition of $\pi$ into four disjoint connection sets $C_0$, $C_1$, $C_2$, $C_3$ , with two elements each, satisfying the following properties:

**Property 1:** Each $C_i^-$, $0 \leq i \leq 3$, contains exactly one element from each of the sets of input terminals $I_0$ and $I_1$ defined by

$$I_0 = \{0,2,4,6\}; I_1 = \{1,3,5,7\}. \tag{4}$$

**Property 2:** For every $0 \leq x \leq 7$ with $x = x_2x_1x_0$ (in binary), if $x$ is contained in the set $C_0^- \cup C_1^-$, then the element $x' = \bar{x}_2x_1x_0$ is in the set $C_2^- \cup C_3^-$ .

**Property 3:** The sets $C_0^+ \cup C_2^+$ and $C_1^+ \cup C_3^+$ each contain exactly one representative from each of the sets of output terminals $O_0$ , $O_1$, $O_2$, $O_3$, defined by

$$O_0 = \{0,1\}; \; O_1 = \{2,3\}; \; O_2 = \{4,5\}; \; O_3 = \{6,7\} \tag{5}$$

**Property 4:** Each $C_i^+$, $0 \leq i \leq 3$, contains exactly one element from each of the sets $O_0 \cup O_1$ and $O_2 \cup O_3$ .

Properties 1-4 have been selected so that, if Switch $i$ in the middle stage of the network carries the connections in the set $C_i$ , for $0 \leq i \leq 3$, no conflicts can arise in any of the switching stages of the network.

The following algorithm constructs a valid partition of the permutation $\pi$ :

**Algorithm 1**

**Step 1:** Partition the set of output terminals of $\pi$ into four sets $P_0$ , $P_1$, $P_2$, $P_3$, with two elements each, as defined below:

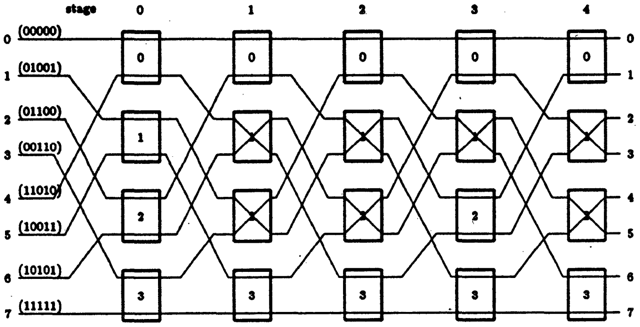


**Figure 1**

**A 5-Stage Shuffle/Exchange Network with $N = 8$**

$$P_0 = \{\pi(0), \pi(4)\}; \; P_1 = \{\pi(1), \pi(5)\}; \quad P_2 = \{\pi(2), \pi(6)\}; \; P_3 = \{\pi(3), \pi(7)\} \tag{6}$$

**Step 2:** Construct two sets $X_0$ and $X_1$, of four elements each, such that each contains exactly one representative from each $P_j$ and $O_j$, for $0 \leq j \leq 3$. That is,

$$|X_i \cap P_j| = |X_i \cap O_j| = 1, \tag{7}$$

for all $0 \leq j \leq 3$ and $i = 0,1$. The set $X_0$ can be constructed in a strictly sequential manner as follows: Start from one of the $P$'s, say $P_i$, and select an element arbitrarily for inclusion in $X_0$. Let this element be '$a$' and let $a \in O_j$ for some $0 \leq j \leq 3$. If '$b$' is the other element in $O_j$ and $b \in P_k$, for some $k$, then $b$ cannot be selected for inclusion in $X_0$ and the other element in $P_k$ has to be chosen. Thus the selection is performed by *looping* from $P_i$ to $O_j$ to $P_k$ and so on, until we reach a set $P_l$ from which an element has already been chosen, which indicates the completion of a cycle. At this point, a fresh cycle may be started by choosing any of the unvisited $P$'s, until all $P$'s are processed. This procedure is identical to the looping algorithm for control of the Benes network [6], and can be likened to the vertex-coloring of a bipartite graph.

Once $X_0$ is constructed, the remaining elements naturally fall into $X_1$ . That is, $X_1 = \{0,1, \dots ,7\} - X_0$.

**Step 3:** Construct sets $R_0$, $R_1$ , $R_2$, $R_3$ as

$$R_0 = X_0 \cap \pi(I_0); \; R_1 = X_0 \cap \pi(I_1); \quad R_2 = X_1 \cap \pi(I_0); \; R_3 = X_1 \cap \pi(I_1) \tag{8}$$

where $\pi(I_0) = \{\pi(x) \mid x \in I_0\}$ and $\pi(I_1) = \{\pi(x) \mid x \in I_1\}$.

**Step 4:** Construct two sets $Y_0$ and $Y_1$, of four elements each, such that each contains exactly one representative from each $R_j$ and $O_j$, for $0 \leq j \leq 3$. That is,

$$|Y_i \cap R_j| = |Y_i \cap O_j| = 1, \; \forall \, 0 \leq j \leq 3 \text{ and } i = 0,1. \tag{9}$$

This construction can be performed in an identical manner as the construction of $X_0$ and $X_1$ in Step 2.

**Step 5:** Construct two sets $X'_0$ and $X'_1$ from $R_0$, $R_1$, $R_2$, $R_3$ as follows:

*Case (i):* If $(X_0 \cap Y_0)$ has representatives from both $\{0,1,2,3\}$ and $\{4,5,6,7\}$ then $X'_0 = X_0$ and $X'_1 = X_1$ .

*Case (ii):* If $(X_0 \cap Y_0)$ has representatives from only one of the sets {0,1,2,3} and {4,5,6,7} then

$$X'_0 = R_0 \cap R_3;\ X'_1 = R_1 \cup R_2. \tag{10}$$

**Step 6:** The connection sets $C_0$, $C_1$, $C_2$, and $C_3$ are now obtained as $C_{2i+j} = \left\{ \begin{pmatrix} x \\ \pi(x) \end{pmatrix} \mid \pi(x) \in X'_i \cap Y_j \right\}, \quad 0 \le i, j \le 1.$ (11)

Before proceeding to prove that the algorithm produces a valid partition of the permutation $\pi$, we show an example to illustrate the idea.

*Example* 1: Consider the permutation

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 4 & 6 & 2 & 3 & 5 & 7 \end{pmatrix}.$$

In this example, $\pi(I_0) = \{0, 4, 2, 5\}$ and $\pi(I_1) = \{1, 6, 3, 7\}$.

*Step 1:*

$$P_0 = \{0, 2\};\ P_1 = \{1, 3\};\ P_2 = \{4, 5\};\ P_3 = \{6, 7\};$$
$$O_0 = \{0, 1\};\ O_1 = \{2, 3\};\ O_2 = \{4, 5\};\ O_3 = \{6, 7\}.$$

*Step 2:*

$$X_0 = \{0, 3, 4, 7\}\ \ X_1 = \{2, 1, 5, 6\}.$$

The set $X_0$ was constructed by first choosing the element 0 from $P_0$; Since $0 \in O_0$, the element 1 cannot be included. As $1 \in P_1$, the element 3 from $P_1$ has to be selected; this completes a cycle. Since $P_2 = O_2$ and $P_3 = O_3$, the representatives from $P_2$ and $P_3$ can be chosen arbitrarily.

*Step 3:*

$$\begin{aligned} R_0 &= X_0 \cap \pi(I_0) = \{0,4\} \\ R_1 &= X_0 \cap \pi(I_1) = \{3,7\} \\ R_2 &= X_1 \cap \pi(I_0) = \{2,5\} \\ R_3 &= X_1 \cap \pi(I_1) = \{1,6\} \end{aligned}$$

*Step 4:*

$Y_0$ and $Y_1$ are constructed by choosing one element from each of $R_0$, $R_1$, $R_2$, $R_3$ so that they have one element in common with each of $O_0$, $O_1$, $O_2$, $O_3$. If we force $Y_0$ to contain the element 0, then

$$Y_0 = \{0, 6, 3, 5\};\ Y_1 = \{4, 1, 7, 2\}.$$

*Step 5:*

Since $X_0 \cap Y_0 = \{0, 3\} \subset \{0, 1, 2, 3\}$, case (ii) applies. Hence,

$$X'_0 = R_0 \cup R_3 = \{0, 1, 4, 6\};$$
$$X'_1 = R_1 \cup R_2 = \{2, 3, 5, 7\}.$$

*Step 6:*

$$X'_0 \cap Y_0 = \{0, 6\};\ X'_0 \cap Y_1 = \{1,4\};$$
$$X'_1 \cap Y_0 = \{3, 5\};\ X'_1 \cap Y_1 = \{2,7\}.$$

Therefore, the connection sets $C_0$ - $C_3$ are given by:

$$C_0 = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 6 \end{pmatrix} \right\};\ C_1 = \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 4 \end{pmatrix} \right\};$$

$$C_2 = \left\{ \begin{pmatrix} 5 \\ 3 \end{pmatrix}, \begin{pmatrix} 6 \\ 5 \end{pmatrix} \right\};\ C_3 = \left\{ \begin{pmatrix} 4 \\ 2 \end{pmatrix}, \begin{pmatrix} 7 \\ 7 \end{pmatrix} \right\}.$$

It may be verified that these sets satisfy Properties 1 through 4.

We now prove that Algorithm 1 produces a valid partition of the permutation $\pi$ as per Definition 3.

*Theorem 1:* The connection sets $C_0$, $C_1$, $C_2$, $C_3$ produced by Algorithm 1 satisfy Properties 1 through 4.

*Proof:* Since $X'_0 \cap X'_1 = Y_0 \cap Y_1 = \emptyset$ and $X'_0 \cup X'_1 = Y_0 \cup Y_1 = \{0,1, \ldots ,7\}$, the connection sets $C_0$, $C_1$, $C_2$, $C_3$ form a partition of $\pi$. We now show that they satisfy each of Properties 1-4.

*Property 1:*

$$C_0^+ \cap \pi(I_0) = (X'_0 \cap \pi(I_0)) \cap Y_0. \tag{12}$$

Assuming that case (i) holds in Step 5 of Algorithm 1,

$$\begin{aligned} C_0^+ \cap \pi(I_0) &= ((R_0 \cup R_1) \cap \pi(I_0)) \cap Y_0 \\ &= (R_0 \cap \pi(I_0)) \cap Y_0, \quad \text{since } R_1 \cap \pi(I_0) = \emptyset \\ &= R_0 \cap Y_0, \quad \text{since } R_0 \subset \pi(I_0). \end{aligned} \tag{13}$$

Now, if case (ii) holds,

$$\begin{aligned} C_0^+ \cap \pi(I_0) &= ((R_0 \cup R_3) \cap \pi(I_0)) \cap Y_0 \\ &= R_0 \cap Y_0, \quad \text{since } R_3 \cap \pi(I_0) = \emptyset \text{ and } R_0 \subset \pi(I_0). \end{aligned} \tag{14}$$

By construction, $|R_i \cap Y_j| = 1$, for $0 \le i \le 3$ and $0 \le j \le 1$. Hence $|C_0^+ \cap \pi(I_0)| = 1$. Similarly, it can be shown that $|C_i^+ \cap \pi(I_j)| = 1$ for all $0 \le i \le 3$ and $0 \le j \le 1$. Thus Property 1 is satisfied.

*Property 2:* If case (i) holds during Step 5, then $X_0 = C_0^+ \cup C_1^+$ and $X_1 = C_2^+ \cup C_3^+$. Hence, Property 2 is satisfied by construction of $X_0$ and $X_1$.

If case (ii) were true, then $X'_0 = C_0^+ \cup C_1^+ = R_0 \cup R_3$ and $X'_1 = C_2^+ \cup C_3^+ = R_1 \cup R_2$. Since $R_0 \subset X_0$ and $R_3 \subset X_1$, the elements in $R_0$ and $R_3$ independently satisfy Property 2. Also, since $R_0 \subset \pi(I_0)$ and $R_3 \subset \pi(I_1)$, $\pi(x_2x_1x_0) \in R_0$ implies $\pi(\bar{x}_2x_1x_0) \notin R_3$. Hence,

$$\pi(x_2x_1x_0) \in (R_0 \cup R_3) \Rightarrow \pi(\bar{x}_2x_1x_0) \notin (R_0 \cup R_3). \tag{15}$$

Similarly,

$$\pi(x_2x_1x_0) \in (R_1 \cup R_2) \Rightarrow \pi(\bar{x}_2x_1x_0) \notin (R_1 \cup R_2) \tag{16}$$

Thus, $X'_0$ and $X'_1$ satisfy Property 2.

*Property 3:*

$$\begin{aligned} C_0^+ \cup C_2^+ &= (X'_0 \cap Y_0) \cup (X'_1 \cap Y_0) \\ &= (X'_0 \cup X'_1) \cap Y_0 \\ &= Y_0 \end{aligned} \tag{17}$$

$$\text{and } C_1^+ \cup C_3^+ = Y_1. \tag{18}$$

By construction of $Y_0$, Property 3 is satisfied.

*Property 4:* $C_0^+ = X'_0 \cap Y_0$.
If case (i) holds during Step 5 of the construction procedure, then $X'_0 = X_0$ and $|(X_0 \cap Y_0) \cap \{0, 1, 2, 3\}| = |(X_0 \cap Y_0) \cap \{4,5,6,7\}| = 1$. This is true of $C_1$, $C_2$, $C_3$, also.

Hence Property 4 holds.

If case (ii) were true, then we have

$$\text{either } (X_0 \cap Y_0) \subset \{0,1,2,3\}, \qquad (19)$$
$$\text{or } (X_0 \cap Y_0) \subset \{4,5,6,7\}. \qquad (20)$$

Equation (19) implies

$$((R_0 \cup R_1) \cap Y_0) \subset \{0,1,2,3\}$$
$$\Rightarrow ((R_0 \cap Y_0) \cup (R_1 \cap Y_0)) \subset \{0,1,2,3\}$$
$$\Rightarrow (R_0 \cap Y_0) \subset \{0,1,2,3\}. \qquad (21)$$

Also, by construction, $|Y_0 \cap \{0,1,2,3\}| = |Y_0 \cap \{4,5,6,7\}| = 2$. Hence,

$$(X_1 \cap Y_0) \subset \{4,5,6,7\}$$
$$\Rightarrow ((R_2 \cup R_3) \cap Y_0) \subset \{4,5,6,7\}$$
$$\Rightarrow (R_3 \cap Y_0) \subset \{4,5,6,7\}. \qquad (22)$$

$$C_0^+ = X'_0 \cap Y_0 = (R_0 \cup R_3) \cap Y_0 = (R_0 \cap Y_0) \cup (R_3 \cap Y_0). \qquad (23)$$

By construction, $|R_0 \cap Y_0| = |R_3 \cap Y_0| = 1$. Hence by Equations (21) and (22), $C_0^+$ has exactly one representative from each of $\{0,1,2,3\}$ and $\{4,5,6,7\}$. Similarly Equation (20) can be shown to imply $(R_0 \cap Y_0) \subset \{4,5,6,7\}$ and $(R_3 \cap Y_0) \subset \{0,1,2,3\}$. Hence $C_0^+$ satisfies Property 4.

In a similar way, each $C_i^+$, $1 \leq i \leq 3$, can be shown to possess representatives from both $\{0,1,2,3\}$ and $\{4,5,6,7\}$. Thus Property 4 is satisfied.

## 3. ROUTING OF PERMUTATIONS IN THE NETWORK

Once a valid partition of the permutation $\pi$ satisfying Definition 3 is obtained, actual routing of the permutation on the network is accomplished as follows:

A 5-bit routing tag is assigned to each of the input–output connections of $\pi$. The routing tag for the connection from input terminal $s = s_2s_1s_0$ to output terminal $d = d_2d_1d_0$ has the form $c_1c_0d_2d_1d_0$, where the bits $c_1$ and $c_0$ are set depending on to which of the sets $C_0$, $C_1$, $C_2$, $C_3$, the connection belongs ($c_1c_0 = 00$ for $C_0$, 01 for $C_1$, 10 for $C_2$, and 11 for $C_3$). The switches in the network are set according to the individual bits of the routing tag — the top output is chosen if the bit is 0 and the bottom output if it is 1. The bit $c_1$ controls Stage 0, $c_0$ controls Stage 1, and $d_2$, $d_1$, $d_0$ control Stages 3, 4, 5, respectively.

*Theorem 2:* If $C_0$, $C_1$, $C_2$, $C_3$ form a valid partition of the permutation $\pi$, the routing described above produces no conflicts in any of the switching stages of the network.

*Proof:* Refer to [9].

Figure 1 shows the routing of the permutation in Example 1 on the 5-stage shuffle/exchange network. The 5-bit vectors shown in parenthesis on the input side are the routing tags for the corresponding connections. Notice that the connections in the set $C_i$ pass through Switch $i$ of the middle stage for all $0 \leq i \leq 3$.

## 4. REDUNDANCY IN THE NETWORK

Due to the flexibility in the construction of $X_0$ and $X_1$ in Step 2 of Algorithm 1, and in the construction of $Y_0$ and $Y_1$ in Step 4, it is possible to fix the state of one switch in each of the last two stages (or the first two stages) of the 5-stage shuffle/exchange network. For example, if we force $X_0$ and $Y_0$ to contain the element 0, then Switch 0 of the last two stages can be set permanently in the *straight* position. Similarly, if $X_0$ and $Y_0$ are forced to contain the element $\pi(0)$, Switch 0 of the first two stages can be set straight permanently. This shows that two switches in the network are redundant. Notice that a Benes binary network of the same size contains three redundant switches [6].

## 5. CONCLUDING REMARKS

In this paper we showed that the lower bound of $(2 \log_2 N - 1)$ stages for rearrangeability of a multistage shuffle/exchange network is also an upper bound for $N \leq 8$. An algorithm for routing arbitrary permutations on the network was also described. The results in this paper have been used in [14] for establishing an upper bound of $(3 \log_2 N - 4)$ stages for rearrangeability of a multistage shuffle/exchange network with $N \geq 8$.

## REFERENCES

[1] K. E. Batcher, "Sorting Networks and their Applications," *Proc. AFIPS Spring Joint Comp. Conf.*, Vol. 32, 1968, pp. 307-314.

[2] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[3] P-Y. Chen, P-C. Yew, D. A. Padua, "Interconnection Networks Using Shuffles," *Computer*, Vol. 14, No. 12, Dec. 1981, pp. 55-64.

[4] C. K. Kothari, S. Lakshmivarahan, H. Peyravi, "A Note on Rearrangeable Networks," *Technical Report*, School of Engineering and Computer Science, University of Oklahoma, Nov. 1983.

[5] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comp.*, Vol. C-24, No. 12, Dec. 1975, pp. 1145-1155.

[6] D. C. Opferman, N. T. Tsao-Wu, "On a Class of Rearrangeable Switching Networks," *Bell Systems Technical Journal*, Vol. 50, May-June, 1971, pp. 1579-1618.

[7] D. S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks," *IEEE Trans. Comp.*, Vol. C-29, No. 3, March 1980, pp. 213-222.

[8] D. S. Parker, *Personal Communication.*

[9] C. S. Raghavendra, A. Varma, "Rearrangeability of the 5-Stage Shuffle/Exchange Network for $N = 8$," *Technical Report CRI-86-05*, Computer Research Institute, University of Southern California, March 1986.

[10] H. J. Siegel, "The Universality of Various Types of SIMD Machine Interconnection Networks," *Proc. 4th Annual Symposium on Computer Architecture*, March 1977, pp. 70-79.

[11] H. J. Siegel, "Partitionable SIMD Computer System Interconnection Network Universality," *Proc. 16th Annual Allerton Conference on Communications, Control, and Computing*, Oct. 1978, pp. 586-595.

[12] J. Sovis, "Uniform Theory of the Shuffle-Exchange Type Permutation Networks," *Proc. 10th Annual International Symposium on Computer Architecture*, June 1983, pp. 185-191.

[13] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Comp.*, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[14] A. Varma, C. S. Raghavendra, "Rearrangeabilit*y* of Multistage Shuffle/Exchange Networks," *Technical Report CRI-86-04*, Computer Research Institute, University of Southern California, March 1986.

[15] A. Waksman, "A Permutation Network," *Journal of the ACM*, Vol. 15, No. 1, Jan. 1968, pp. 159-163.

[16] C. L. Wu, T. Y. Feng, "The Universality of the Shuffle-Exchange Network," *IEEE Trans. Comp.*, Vol. C-30, No. 5, May 1981, pp. 324-332.

# CONTROL ALGORITHMS FOR THE AUGMENTED DATA MANIPULATOR NETWORK

Daeshik Lee
Department of Computer Science
University of Illinois
1304 West Springfield
Urbana, IL 61801

Kyungsook Y. Lee
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210

## Abstract

The cube networks are equipped with very efficient local control algorithms: the bit control by the destination tags. In this paper we present an equally efficient local control algorithm called the **signed bit difference (SBD) tag control** for the ADM (IADM). The signed bit (0, 1, $\bar{1}$) control in place of the bit (0, 1) control is a natural consequence of the (3x3) switch size of the ADM (IADM).

We show that the $\Omega(\Omega^{-1})$-passable permutations are a subset of the ADM(IADM)-passable permutations under the SBD tag control. We also show that for the $\Omega(\Omega^{-1})$-passable permutations the SBD tag can be replaced by the destination tag together with a single bit information of a switch number yielding the **destination tag control** for the ADM (IADM).

A global control algorithm which realizes all the IADM-passable permutations is also given.

## 1. Introduction

In multiple processor systems an interconnection network provides communication paths for processor-processor or processor-memory information exchanges. The (NxN) augmented data manipulator (ADM) network [8] is an interconnection network, which consists of n switching stages each with $N = 2^n$ (3x3) switches plus one output stage. The inverse ADM (IADM) is identical to the ADM except that the direction from the input side to the output side is reversed. An (8x8) IADM is shown in Figure 1 together with its switching stage numbers and switch numbers of each stage. The properties of the ADM (IADM) have been studied actively [1, 11, 8, 13, 9, 10, 6, 4, 5].

The ADM control algorithm is based on the routing tag T which is a representation of (D-S), where D is the destination and S is the source:

$$T = t_{n-1} \ldots t_0 = D-S \pmod N),$$
$$-(N-1) \leq T \leq (N-1), \text{ where } t_i \text{ can be } 0,\ 1 \text{ or } \bar{1}.$$

For a given D and S there exist many different representations of (D-S) modulo N, and each different representation (or tag) corresponds to a possible path from S to D. To realize a permutation we have to choose N such tags without causing a conflict. Definitely some systematic generation of tags is required. One candidate is the **signed magnitude difference (SMD)** tag (called the natural tag in [11])

$$T = t_n t_{n-1} \ldots t_0 = D-S, \ -(N-1) \leq T \leq (N-1),$$

where $t_i$ is 0 or 1, and $t_n$ is the sign bit. Thus the value of T is

$$\text{value}(T) = (-1)^{t_n} \sum_{i=0}^{n-1} 2^i t_i$$

**Unfortunately it was shown in [3] that a very small subset of ADM-passable permutations can be realized with this type of tags. In Section 2, a new tag called the signed bit difference (SBD) tag is defined. It can be regarded as a natural extension** of the bit control of a cube network for the ADM(IADM), which is based on (3x3) switches. It was known that the ADM passes all the omega-passable permutations ($\Omega$) [8]. We show that the set of ADM-passable permutations under the SBD tag control contains $\Omega$. The 2n-bit full routing tag obtained by bitwise exclusive-oring of the source and the destination for the cube-passable permutations [12] is in essence identical to the SBD tag. Also in Section 2 we show that the destination tag together with a single bit information from a switch number can replace the SBD tag for $\Omega(\Omega^{-1})$.

In Section 3 a global control algorithm of the IADM is given, which realizes all the IADM-passable permutations. The conclusions are drawn in Section 4.

## 2. Local Control Algorithms

### *2.1.* The Signed Bit Difference Tag Control Algorithm

Let the source $S = s_{(n-1)} \dots s_1 s_0$, and the destination $D = d_{(n-1)} \dots d_1 d_0$, where $s_i, d_i \epsilon \{0,1\}$ for $0 \leq i < n$. Define the **signed bit difference (SBD) tag**

$$T = t_{(n-1)} \dots t_1 t_0,$$

where $t_i = 0$ if $d_i = s_i$,

$t_i = 1$ if $d_i = 1$ and $s_i = 0$,

$t_i = \bar{1}$ if $d_i = 0$ and $s_i = 1$.

In other words the bit $t_i$ of the SBD tag T is the bit difference $(d_i - s_i)$ represented in the fully redundant binary number system. The IADM can be controlled locally at each switch as follows.

### SBD Tag Control Algorithm of IADM (ADM)

**The switch j of the switching stage i is controlled using the SBD tag in the following way:**

if $t_j = 0$, choose the straight connection,

if $t_j = 1(\bar{1})$, choose $+2^i(-2^i)$ connection,

for $0 \leq j < N$, $0 \leq i < n$.

The SBD tag control of the ADM (IADM) is very natural, in that it covers each bit difference of the destination and the source at each corresponding switching stage, without generating any carry to or borrow from neighbor stages. It is a direct one-to-one mapping of the signed bit difference $(d_i - s_i)$ to the i-th switching stage of the ADM (IADM).

Let $\Omega(\Omega^{-1})$ denote the set of the omega (inverse omega) network passable permutations [2]. Similarly let $A(A^{-1})$ denote the set of the ADM (IADM) passable permutations. It has been known [8] that

$$\Omega \subset A.$$

Because of the symmetry this also implies that

$$\Omega^{-1} \subset A^{-1}.$$

In the following we show that the SBD tag is the control tag for $\Omega(\Omega^{-1})$ on the ADM(IADM).

**Theorem 1.**

The SBD tag control algorithm realizes $\Omega^{-1}$ on the IADM.

**Proof**

For all i, $0 \leq i < n$, the SBD tag has the property:

if $s_i = 0$ then $t_i = d_i = 0$ or $1$,
if $(s_i = 1$ and $d_i = 1)$ then $t_i = 0$
else if $(s_i = 1$ and $d_i = 0)$ then $t_i = \bar{1}$.

Consider two input sources $S_1 = s_{1,(n-1)} \dots s_{1,0}$ and $S_2 = s_{2,(n-1)} \dots s_{1,0}$, where $s_{1,i} = 0$, $S_2 = S_1 + 2^i$ for an i, $0 \leq i < n$. Let $D_1 = d_{1,(n-1)} \dots d_{1,0}$, $D_2 = d_{2,(n-1)} \dots d_{2,0}$, $T_1 = t_{1,(n-1)} \dots t_{1,0}$, $T_2 = t_{2,(n-1)} \dots t_{2,0}$ be the destinations of $S_1$, $S_2$, and the SBD tags of $S_1$, $S_2$, respectively. Then by the above property of the SBD tag:

$$(t_{1,i}, t_{2,i}) = \begin{matrix} (0,0) \\ (1,\bar{1}) \\ (0,\bar{1}) \\ (1,0) \end{matrix}$$

These four cases correspond to the four possible settings of a (2x2) switch in the inverse omega network: straight, cross, upper broadcast, lower broadcast. (Only the first two are relevant for the permutations.)

Q.E.D.

**Corollary 1.**

The SBD tag control algorithm realizes $\Omega$ on the ADM.

### *2.2.* The Destination Tag Control Algorithm

Next we show that $\Omega^{-1}$ can be realized on the IADM using a different tag -- the destination tag together with a single bit information of a switch number as well.

### Destination Tag Control Algorithm of IADM (ADM)

The switch number j of the switching stage i, $j_i = j_{i,(n-1)} \dots j_{i,0}$ is controlled by $d_i$ of the incoming destination tag $D = d_{(n-1)} \dots d_0$ and $j_{i,i}$ as follows, $0 \leq i < n$, $0 \leq j_i < N$:

if $(j_{i,i} = 0$ and $d_i = 0)$ or
  $(j_{i,i} = 1$ and $d_i = 1)$ then
  choose the straight connection,
if $(j_{i,i} = 0$ and $d_i = 1)$ then
  choose the $+2^i$ connection,
if $(j_{i,i} = 1$ and $d_i = 0)$ then
  choose the $-2^i$ connection.

Note that IADM(ADM) destination tag control is exactly like the cube destination control with the $(j_{i,i} = 0)$ condition substituted for the upper input port, and the $(j_{i,i} = 1)$ condition for the lower input port of a switch in a cube network.

**Theorem 2**

The IADM destination tag control algorithm realizes $\Omega^{-1}$ on the IADM.

**Proof**

The two cases of the SBD tag $(t_{1,i}, t_{2,i})$ being (0,0) or $(1,\bar{1})$ correspond to ($s_{1,i} = d_{1,i} = 0$, and $s_{2,i} = d_{2,i} = 1$) or ($s_{1,i} = 0$, $d_{1,i} = 1$, and $s_{2,i} = 1$, $d_{2,i} = 0$), respectively. The switch $j_i$ receives its destination from one of possible $2^i$ sources of stage 0. If $j_{i,i}$ is equal to the i-th bit of any possible source numbers, the IADM destination tag control algorithm is equivalent to the SBD tag control algorithm for $\Omega^{-1}$, thus realizing $\Omega^{-1}$ on the IADM.

Let $S_j = s_{j,(n-1)} \dots s_{j,0}$ for $0 \leq j < N$ denote the N sources at the stage 0. Then $j_0 = S_j$ for $0 \leq j < N$. At stage 1, by the destination tag control algorithm applied to the stage 0, the output from $j_0$ is the input to $j_1$ of the stage 1,

$$j_1 = s_{j,(n-1)} \cdots s_{j,1} \, d_{j,0}$$

In general at stage i, $0 \leq i < n$, by the destination tag control algorithm

$$j_i = s_{j,(n-1)} \cdots s_{j,i} \, d_{j,(i-1)} \cdots d_{j,0}$$

receives the input destination created by $j_0$ of stage 0. Thus

$$j_{i,i} = s_{j,i}$$

for $0 \leq j < N$, $0 \leq i < n$.

Q.E.D.

**Corollary 2**

The destination tag control algorithm of ADM realizes $\Omega$ on the ADM.

The SBD tag control, the destination tag control of the IADM and the destination tag control of the cube networks are shown in Figure 2.

## 3. A Global Control Algorithm

### *3.1.* Background

In the previous section we showed two local control algorithms which can realize $\Omega^{-1}$ on the IADM. The implication of the previous section is the fact that although the ADM is more powerful than a cube network, once a fixed local control algorithm is imposed upon, it is generally equal to a cube network. The power of the ADM is rather in the variety of control structures; many different types of local control tags with different permuting power can be generated, whereas only two control tags (the destination tag, and the tag obtained by bit exclusive-oring the destination and the source [11], which have the same permuting power) exist for a cube network. It seems to be the case that a global control algorithm is in order to achieve the full potential of the ADM. In this section we describe a global control algorithm which is simple, systematic, and can pass all the IADM-passable permutations. If a given permutation is not IADM-passable, it prints the fact.

Define a **control matrix** C(0:(N-1), 0:(n-1)) of which the element C(i,j) contains the switch setting information 0, 1, $\bar{1}$ for the i-th switch of the j-th stage. As before the straight path is chosen for the control signal 0, and $+2^j(-2^j)$ path is chosen for $1(\bar{1})$. At the beginning, C is initialized by setting C(i,j) = j-th bit of the SBD tag at source i. The column C(-,j) represents the switch settings of N switches of the switching stage j, called the **stage setting.** A stage setting creates a conflict if there is any pair C(i,j), C(k,j) choosing the paths leading to the same switch at the next switching stage (j+1).

A permutation is IADM-passable if all the stage settings are conflict free. In [3] a ring model is used to describe the IADM. Due to the $0, \pm 2^i$ connection patterns at stage i, all the switches of stage i are partitioned into $2^i$ equivalence classes modulo $2^{(n-i)}$ (see Figure 3). The stage setting of stage i consists of $2^i$ independent settings of $2^i$ partitions (called **partition setting**).

A stage setting is conflict free if all the $2^i$ partition settings are conflict free. A permutation is IADM-passable if every stage setting is conflict free. Each conflict free partition setting (called a **good string**) can be divided into two types [3]. In the following * ($^+$) denotes a repetition of zero or more (one or more) times, and the length of the string is equal to the cardinality of a partition.

(I) **Good string type 0.**
There is at least a pair of 0s, and can be represented as
$(0(1\bar{1})^*0 + 0^*)^*$

Each type 0 string creates a unique partition at the next stage.

(II) **Good string type 1.**
There is no 0 switch setting in a string. There are four type 1 strings, and all these strings create identical partitions at the next stage:
$(\bar{1})^*$, $(1\bar{1})^*$, $(1)^*$, $(\bar{1}1)^*$

The good string type 0 is based upon the fact that once a pair of switches in a partition are set straight, any switch between the pair should be set 1 followed by $\bar{1}$ repeatedly to avoid a conflict. The basis for the good string type 1 is that if there is no switch set straight, then the given four switch settings are the only conflict free switch settings. There are $(G_k\text{-}2)$ type 0 strings, where $G_k = G_{(k-1)} + G_{(k-2)}$, k(= string size) $\geq 3$, $G_2=3$, $G_1=1$. The relation of $G_N$ to the number of IADM-passable permutations $P_N$ is given as [3]

$$P_N = (G_N - 1)\ P^2_{N/2}$$

with all $(G_N\text{-}2)$ type 0 strings included in $(G_N\text{-}1)$ plus one coming from type 1 strings.

Thus the majority of the IADM-passable permutations have unique switch settings. The choice of switch settings can only be provided when a partition setting does not require any straight switch setting, and thus belong to type 1 good string. Although the four different type 1 strings, $(\bar{1})^+$, $(1\bar{1})^+$, $(1)^+$, $(\bar{1}1)^+$ create identical partitions, they create two different permutations: one by $(\bar{1})^+$ or $(1\bar{1})^+$, and the other by $(1)^+$, or $(\bar{1}1)^+$. Hence given an overall permutation for the IADM a partition can be set at most in two different ways.

### 3.2. IADM-Global-Control Algorithm

We are ready to introduce the IADM-global-control algorithm. First a very brief description of the general flow is given.

**General Flow of the IADM-global-control**

```
Start with the SBD control tags for
the IADM;
loop for all stages 0 to (n-1);
loop for all partitions of each stage;
```

If the partition setting has any 0, then there is a unique partition setting $(0(1\bar{1})^*0 + 0^*)^*$. Change the original partition setting into this form. If it can not be done, backtrack to the most recent stage with an alternative. If there is no such stage, the given permutation is not IADM-passable. Stop.

If the partition setting does not have any 0, change it into $(1)^+$ (an arbitrary choice between $(1)^+$ and $(\bar{1}1)^+$), mark it for backtracking as there is an alternative, $(\bar{1})^+$ (an arbitrary choice between $(\bar{1})^+$ and $(1\bar{1})^+$).

```
end loop
end loop

Conflict free control tags have been
obtained.

end IADM-global-control
```

Next we give the algorithm in detail. The procedures called by IADM-global-control are omitted due to space limitation.

**Procedure** IADM-global-control (D,C);

/* IADM-global-control generates control signals in C if the given permutation D is passable. It prints "not passable" if D is not passable. */

```
        bit array   D(0: N-1, 0: n-1);
         /* N n-bit destinations */
 signed bit array   C(0: N-1, 0: n-1);
          /* control bits 0, 1 or 1,
             for the entire IADM */

 signed bit array   STRING(0: N-1);
          /* control bits 0, 1, or 1,
             for a partition of a stage */

     integer array  BTR.STAGE(0: n-2),
                    BTR.PART(0: $2^{(n-2)}-1$);
          /* stacks for backtracking */

           integer  j, k, m, length,
                    btr-ptr;
           boolean  success;


initialize C with SBD tags;

for all stage j=0 to n-1;

for all partition k=0 to $2^{j}-1$;
    btr-ptr = 0; success = false;
    /* build a string for stage j,
       partition k */
    call build-string(STRING, j, k,
     length);

    if any 0 in the STRING
        then
        /* type 0 string : unique
           setting */
        call make-good-string(STRING,
         length, 0, success);
            if success = false
                then /* can we backtrack
                        to change (1)+ to
                        (1̄)+ in previous
                        stages? */
                if btr-ptr = 0
                    then
                    /* no backtracking
                       possible */
                       print ("not
                        passable
                        permutation");
                       return;
                fi;

                /* start backtracking */
                j = BTR.STAGE(btr-ptr);
                k = BTR.PART(btr-ptr);
                btr-ptr = btr-ptr - 1;
                call  build-string
                 (STRING, j, k, length);
                call  make-good-string
                 (STRING, j, k, length,
                    1, success);
           fi;

      else /* type 1 string:  two
                alternative ways of
                setting (1)$^+$ or ($\bar{1}$)$^+$ */
                /* (1)$^+$ tried first */
                call make-good-string
                (STRING, j, k, length,
                1, success);

                /* store in the back
                   track stacks */
                btr-ptr = btr-ptr + 1;
                BTR.STAGE (btr-ptr) = j;
                BTR.STAGE (btr-ptr) = k;
    fi
    simulate the switch setting by moving
    the rows of C according to the
    control signals for this partition
    (only stages j+1 through n-1).

end for partition;
end for stage;

/* C contains the complete, correct
   switch setting signals for the IADM */

endproc IADM-global-control;
```

### *3.3.* An Example

As an example consider the unshuffle permutation on an (8x8) IADM. We obtain the SBD tags C = $c_2c_1c_0$ by calculating the signed bit difference $d_i$-$s_i$ for $c_i$, $0 \leq i < 3$.

| D | | | S | | | C | | |
|---|---|---|---|---|---|---|---|---|
| $d_2$ | $d_1$ | $d_0$ | $s_2$ | $s_1$ | $s_0$ | $c_2$ | $c_1$ | $c_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | $\bar{1}$ |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | $\bar{1}$ | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | $\bar{1}$ | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | $\bar{1}$ | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $\bar{1}$ |
| 0 | 1 | 1 | 1 | 1 | 0 | $\bar{1}$ | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

The unshuffle is not IADM-passable under the SBD tag control because at the very first stage (stage 0) we have the stage tag

$$0\ \bar{1}\ 1\ 0\ 0\ \bar{1}\ 1\ 0$$

which does not satisfy the condition of type 0 good string $(0\ (1\bar{1})^*\ 0 + 0^*)^*$ as it is. Figure 4 shows the actual conflicts caused by 0 followed by $\bar{1}$.

So by the IADM-global-control C(-,0) is changed to (Figure 5) a good type 0 string $(01\bar{1}0)^2$ or

$$0\ 1\ \bar{1}\ 0\ 0\ 1\ \bar{1}\ 0$$

resulting in new tags ($C_1$). When we simulate the effect of switch settings at stage 0, we obtain $C_2$. For stage 1, we have two independent partitions of switches (0, 2, 4, 6), (1, 3, 5, 7). Both partition tags are $(0\ \bar{1}\ 1\ 0)$ obtained by $c_1$. These should be changed to $(0\ 1\ \bar{1}\ 0)$ as shown in $C_3$. $C_4$ is obtained from $C_3$ by simulating switch settings of stage 1. The stage 2 has four independent partitions of switches (0, 4), (2,6), (1, 5), (3, 7). All the partition tags are (0 0) which is a good type 0 string. Thus $C_4$ contains a complete, conflict free switch setting for the unshuffle. The application of $C_4$ to IADM is given in Figure 6. This permutation did not require any backtracking, as all the partition settings were type 0 tags.

### *3.4.* ADM-global-control Algorithm

The ring model of ADM [3] defines two types of good strings. Based on these the IADM-global-control was developed. We can obviously develop an ADM-global-control in much the same way. The only difference is that the ADM-global-control works from stage n to stage 1 reversing the order. The reverse direction does complicate the conflict free tag calculation. When we start from the lowest order bit, value 0 in that position can not change thus rendering some degree of determinism. This fact is propagated to bit position 1, 2 .... (n-1). But this is not the case when we work from the reverse direction. Thus we might end up with a conflict stage setting for stage i, while producing conflict free stage settings for stages < i, requiring much more backtracking than is required for the IADM in general. Hence, the IADM can be controlled faster than the ADM for the realization of all the possible permutations. Instead of presenting another very similar procedure to the IADM-global-control, we simply show an example of the effect of such a procedure.

Consider a shuffle on the (8x8) ADM. We first calculate SBD tags C = $c_2c_1c_0$.

| D | | | S | | | C | | |
|---|---|---|---|---|---|---|---|---|
| $d_2$ | $d_1$ | $d_0$ | $s_2$ | $s_1$ | $s_0$ | $c_2$ | $c_1$ | $c_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | $\bar{1}$ |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | $\bar{1}$ | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | $\bar{1}$ |
| 0 | 0 | 1 | 1 | 0 | 0 | $\bar{1}$ | 0 | 1 |

$$\begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array} \qquad \begin{array}{ccc} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \qquad \begin{array}{ccc} \bar{1} & 1 & 0 \\ 0 & \bar{1} & 1 \\ 0 & 0 & 0 \end{array}$$

The stage 2 consists of four independent partitions of switches (0, 4), (2, 6), (1, 5), (3, 7) with partition settings $(0\ \bar{1})$, (1 0), $(0\ \bar{1})$, (1 0), respectively. These partition settings are not good type 0 strings, so we change all of them into (0 0) in $C_1$ (Figure 7). The stage 1 consists of two independent partitions of switches (0, 2, 4, 6) and (1, 3, 5, 7) with partition settings $(0\ 1\ \bar{1}\ \bar{1})$, $(1\ 1\ \bar{1}\ 0)$, respectively. We change both into $(0\ 1\ \bar{1}\ 0)$ in $C_2$. By simulating stage 1 stage settings $C_3$ is produced. The stage tag of stage 0 in $C_3$ is a good type 0 string, $(0\ 1\ \bar{1}\ 0\ 0\ 1\ \bar{1}\ 0)$. The switch setting of the (8x8) ADM for the shuffle by $C_3$ is shown in Figure 8.

## 4. Conclusions

The signed bit difference (SBD) tag was defined. Due to the three possible paths of each switch in the ADM a signed bit $(1, 0, \bar{1})$ is required to control each switch. The signed bit control algorithm for the ADM can be regarded as a natural extension of the bit control algorithm for a cube network. The SBD tag control algorithm is the signed bit control algorithm, which maps the bit difference in the destination and the source directly onto the corresponding switching stage. The $\Omega(\Omega)^{-1}$-passable permutations are shown to be a subset of ADM(IADM)-passable permutations under the SBD tag control.

We also showed that for the cube passable permutations the SBD tag can be replaced by the destination tag itself together with a single bit information from each switch (the i-th bit of the switch number in the i-th stage). Thus the destination tag control algorithm for the ADM was defined.

A global control algorithm which realizes all the IADM-passable permutations was presented.

Both local control algorithms are applicable to the gamma network [7]. A global control algorithm for the gamma network is being investigated.

## 5. References

[1] Adams, G. B. and Siegel, H. J.
The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems.
*IEEE Transactions on Computers* C-31(5):443-454, May, 1982.

[2] Lawrie, D. H.
Access and Alignment of Data in an Array Processor.
*IEEE Transactions on Computers* C-25(12):1145-1155, December, 1975.

[3] Lee, Daeshik.
Counting Problems in the Augmented Data Manipulator.
submitted for publication.

[4] Leland, Mary Diane Palmer.
*Properties and Comparison of Multistage Interconnection Networks for SIMD Machines.*
PhD thesis, University of Wisconsin, December, 1983.

[5] Leland, Mary Diane Palmer.
On the Power of the Augmented Data Manipulator Network.
*Proceedings of 1985 International Conference on Parallel Processing* :74-78, August, 1985.

[6] McMillen, R. J. and Siegel, H. J.
Routing Schemes for the Augmented Data Manipulator Network in an MIMD System.
*IEEE Transactions on Computers* C-31(12):184-196, December, 1982.

[7] Parker, D. S. and Raghavendra, C. S.
The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths.
*Proceedings of the 9th Annual Symposium on Computer Architecture* :73-80, 1982.

[8] Siegel, H. J. and Smith, S. D.
Study of Multistage SIMD Interconnection Networks.
*Symp. Comp. Arch.* :223-229, 1978.

[9] Siegel, H. J.
Interconnection Networks for SIMD Machines.
*Computer* 12:57-65, June, 1979.

[10] Siegel, H. J.
The Theory Underlying the Partitioning of Permutation Network.
*IEEE Transactions on Computers* C-29:791-800, September, 1980.

[11] Siegel, H. J. and McMillen, R. J.
Using the Augmented Data Manipulator Network in PASM.
*Computer* , February, 1981.

[12] Siegel, H. J. and McMillen, R. J.
The Multistage Cube: A Versatile Interconnection Network.
*Computer* :65-75, December, 1981.

[13] Smith, S. D., Siegel, H. J., McMillen, R. J., Adams, G. B.
Use of the Augmented Data Manipulator Multistage Network for SIMD Machines.
*1980 International Conference on Parallel Processing* :75-78, August, 1980.

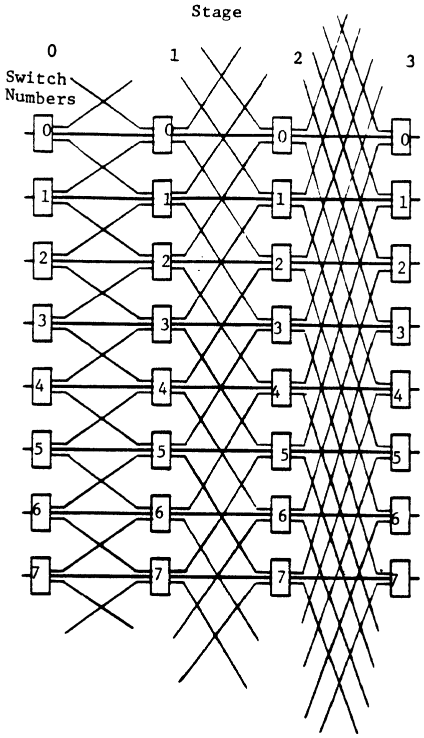


Fig. 1 An (8x8) IADM

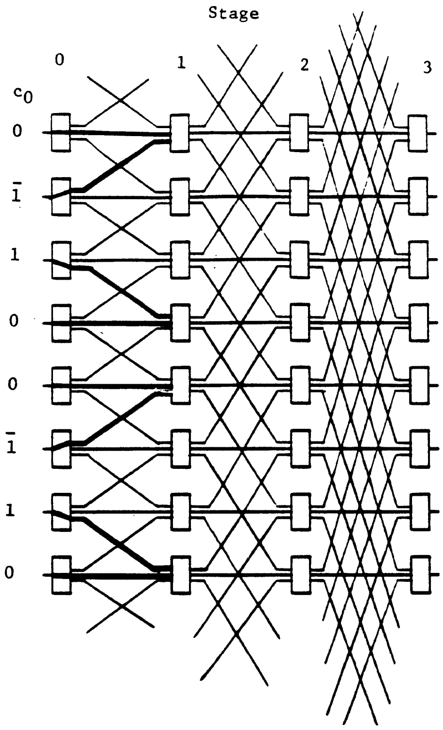
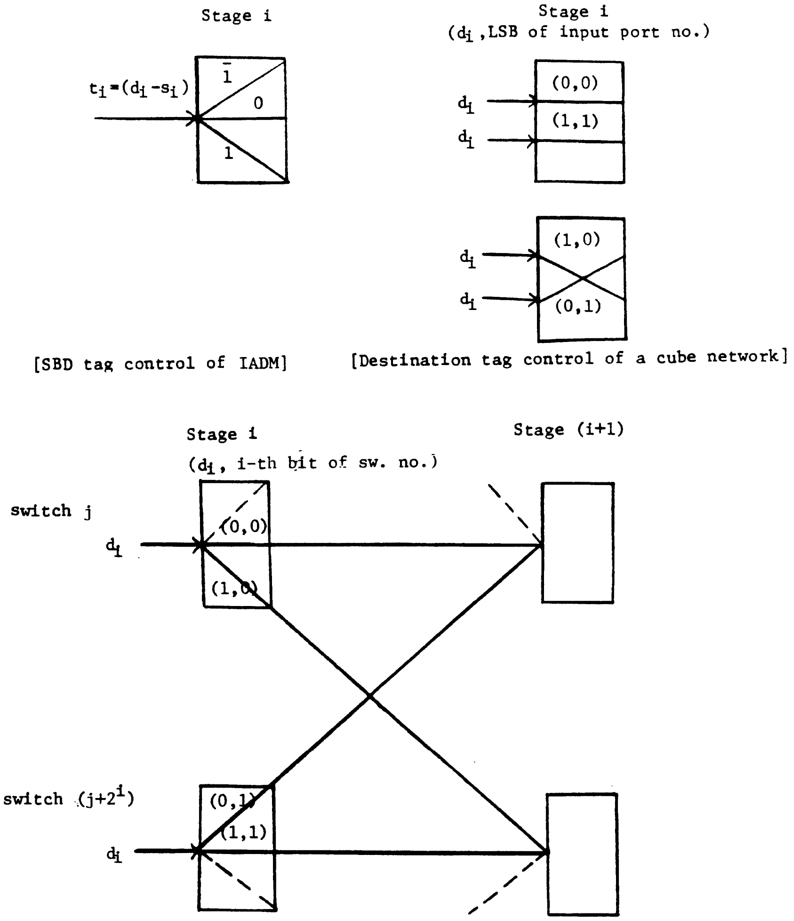


Fig. 4 Conflicts in stage 0 for the unshuffle permutation by the SBD tag control.



[SBD tag control of IADM]

[Destination tag control of a cube network]

[Destination tag control of the IADM]

Fig. 2 The SBD tag control of the IADM, the destination tag control of the IADM, and the destination tag control of a cube network.

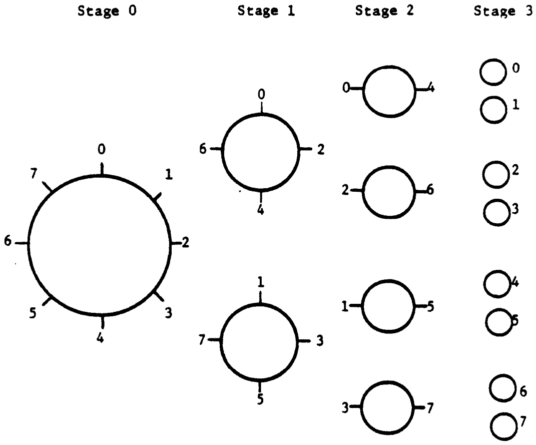
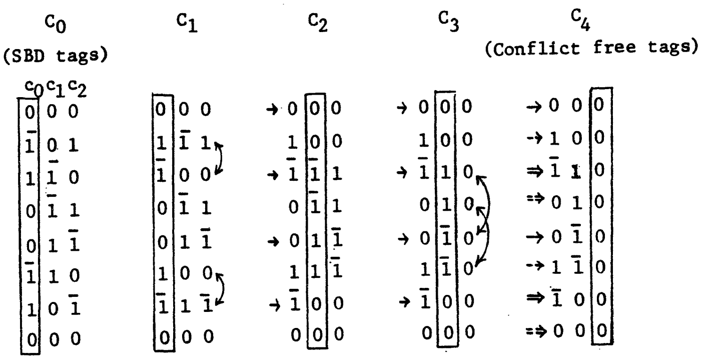


Fig. 3 Partitions of switches at each stage of the (8x8) IADM. (Numbers around each circle are switch numbers.)



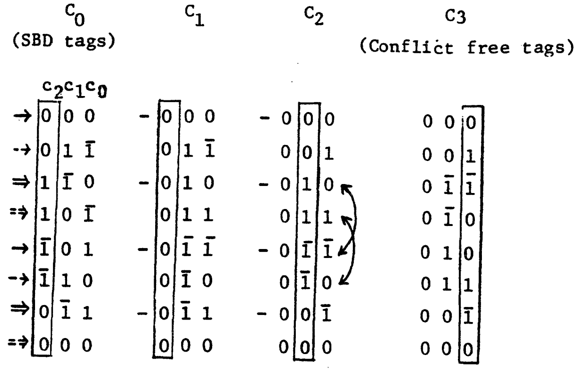


Fig. 5 Generation of conflict free tags by the IADM-global-control for the unshuffle permutation.

Fig. 7 Generation of conflict free tags by the ADM-global-control for the shuffle permutation.

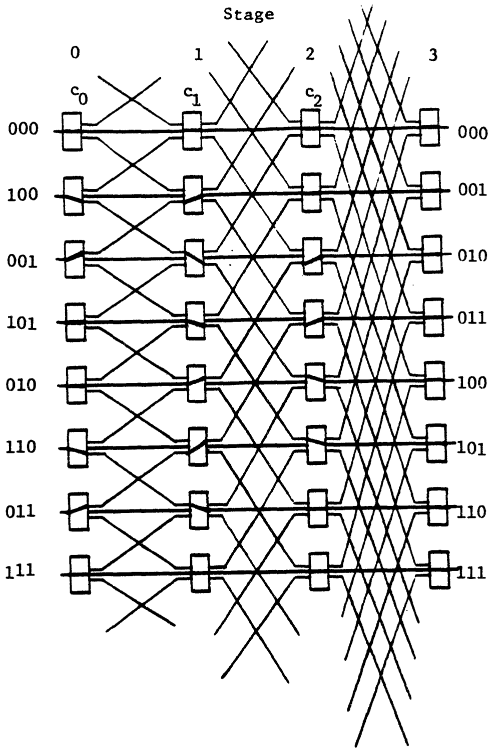


Fig. 6 The unshuffle permutation on the (8x8) IADM by the IADM-global-control.

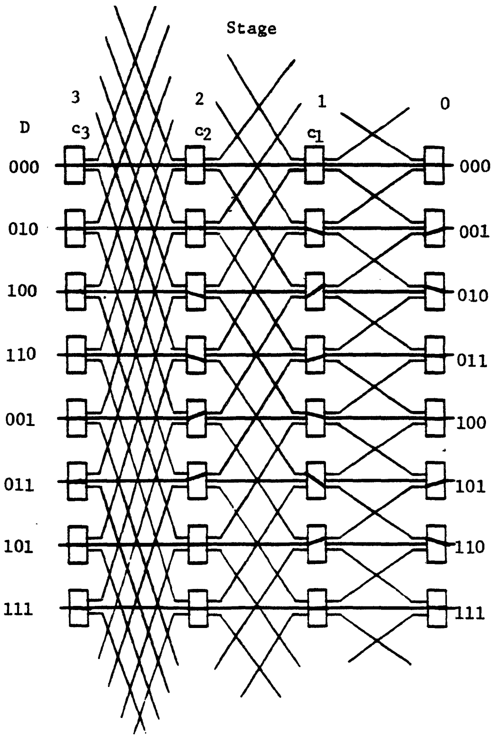


Fig. 8 The shuffle permutation on the ADM by the ADM-global-control.

# Optimistic vs Pessimistic Concurrency Control Algorithms: A Comparative Study

M.H. Nagi
Dept. of Computer Science
Alexandria University
Alexandria, Egypt

A.A. Helal and A.K. Elmagarmid
Computer Engineering Program
Dept. of Electrical Engineering
Pennsylvania State University
University Park, PA 16802

## ABSTRACT

The Performance evaluation of concurrency control algorithms (CCA) has been extensively studied in both the centralized and decentralized environments (GARE79, HSIA81, SHMU82, CHES83, SHUM81, AGRA85, MITR84). However, differing performance models, assumptions and parameters studied have led to contradictory results [AGRA85,HELA86].

In this paper the behaviour and performance of two fundamentally different CCAs in single site databases has been investigated. These are the dynamic two-phase locking (2PL) and the commit-time validation (CTV). 2PL represents a pessimistic approach to concurrency control whereas CTV is an optimistic approach [KUNG81]. For each algorithm a performance model was constructed and the simulation technique was used to perform the study. Three parameters affecting data contention are studied in this paper. These parameters are: the degree of multiprogramming (the load effect), the read/write mix (ratio of query to update) and the database granularity. Unlike previous studies, in this paper we studied the three aforementioned parameters altogether; thus, providing insight into the composite effect of these parameters.

## Introduction

Although, the parallel processing of transactions enhances the system's performance, data misuse may result in the absence of concurrency control mechanisms. There are three generically different approaches that can be used to design concurrency control algorithms. These are: locking, timestamping, and the optimistic approach [BERN81]. In this paper we study and compare the performance of two of these three generically different approaches. These are the Dynamic Two-Phase Locking (Locking approach) [Mena78, Rose 78] and the Commit-Time Validation algorithms (Optimistic Approach) [Bada79, Baye80, Kung81].

Dynamic Two-phase Locking synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. It regulates data access as follows. Before reading a data item X, a transaction must own a readlock on X. Before writing into X, it must own a writelock on X. The ownership of locks is governed by two rules: (1) A transaction may own a readlock on an item x as long as no other transaction owns a writelock on X, and a transaction may own a writelock on an item X if no other transaction owns a readlock or a writelock on the same item X. (2) Once a transaction surrenders ownership of a lock it may never obtain any additional locks. The 2PL controller may block a transaction by causing it to wait for unavailable locks. Due to this blocking, deadlocks may result. Deadlocks could either be prevented or detected and then resolved [ELMA85]. In this paper, 2PL uses a deadlock detection scheme.

In CTV (also called serial validation), the concurrency controller keeps track of the write-sets of recently commited transactions. Transactions run freely until commit time, at which point each transaction is submitted to a validity test to see if commiting it will leave the database in a consistent state. For a committing transaction $T_i$, the test considers all recently committed transactions $T_{rc}$, where a recently committed transaction is one that has committed since $T_i$ started running. The test results in $T_i$ being commited iff:

$$\text{Readset}(T_i) \cap \text{Writeset}(T_{rc}) = \phi$$

for all $T_{rc}$ and being restarted otherwise.

Two-phase commit, a recovery protocal normally used in distributed database systems, is utilized both by the 2PL and the CTV algorithms (in single site database) in order to preserve transaction atomicity and database consistency in presence of system failures. There is no logging, shadowing, nor differential files used in this work. In two-phase commit (2PC), transactions write their updates in the main memory. When a transaction ends, its updates are committed in two-phases. In the first phase, the updates in the main memory are copied to a secure storage. The second phase starts by copying each single update to its place in the stored database. If a failure occurs before the second phase starts, then the database is still consistent. If a failure occurs during the second phase, the database may become inconsistent but this inconsitency can be rectified using the secure storage.

In this study it is assumed that the transaction size is large. This gives worst case results for a system with transactions of mixed sizes. In the next two sections, the simulation performance model is explained for each of the 2PL and the CTV algorithms. Section 4 lists the

0190-3918/86/0000/0131 $01.00 © 1986 IEEE

unified model assumptions. Section 5 explains the experiments and the results. Finally, the concluding remarks are given in section 6.

## 2. Simulation of the Two-Phase Locking algorithm

A fixed number of transactions are continuously cycled around the model shown in figure 1. Initially, all transactions are put on the lock request queue. A transaction then goes through the following stages.

(1) A transaction requests the next needed lock from the lock manager which in turn tries to get this lock. If the lock manager succeeds, the transaction is granted the lock. If the lock granted was a read lock, the transaction is put on the low priority disk queue in order to read the item needed. When the read operation ends, the transaction is put back on the lock request queue to request the next needed lock when the lock manager is free. If the lock granted was a write lock, no disk write operation takes place (as the two-phase commit is employed, all writes take place at commit time) and again, the transaction is put back on the lock request queue to request the next needed lock when the lock manager is free.

(2) If the lock manager fails to get the lock on the item needed, because other transaction(s) hold(s) a lock on that item, the lock manager tries to let the requesting transaction wait for the unavailable lock until it is released. If this waiting would result in a deadlock, the lock manager then restarts the requesting transaction. Otherwise, the requesting transaction is put on one of the block queues (virtually, each database item has a block queue) in which case the transaction is said to be blocked.

(3) When the restarter restarts a transaction, it releases all locks held by that transaction. Consequently, some blocked transactions should become unblocked and they are put on the front of the lock request queue. The workspace of the restarted transaction is reset and it reexecutes from the beginning.

(4) When a transaction completes, it is put on the high priority disk queue and all its writes are committed following the two-phase commit protocol. When a transaction is committed, another transaction arrives instantaneously at the system.

The concurrency control is modeled by two servers: The lock manager and the restarter. The lock manager provides two kinds of services. The first is the "Try to lock" service for a requesting transaction. This service takes a time delay $\Delta_{ttl}$ to be completed. $\Delta_{ttl}$ is found by evaluating the size of actually executed code for this service. Size of code for this service is found to be around 100 instructions. Assuming one MIPS processor, this service takes about 0.0001 seconds to be completed. The second service provided by the lock manager is the "Try to block" service in which the lock manager performs several deadlock detection tests to see if blocking a transaction would result in a deadlock or not. This service takes a time delay $\Delta_{ttb}$ to be completed. Size of code for this service is not estimated but rather an actual deadlock detection algorithm is coded and a counter is used to count the number of actually executed instructions while calling this code. The restarter server releases locks held by an aborted transaction and advances the block queues. This service takes a time delay $\Delta_r$ to be completed. Size of actually executed code for this service is found to be about 5000 Instructions. Assuming one MIPS processor, this service takes about 0.005 seconds to be completed. It should be noted that modeling of the concurrency controller as a set of servers allows the utilization of multiprocessing whenever possible. In case of one processor computer, at most one server is functioning at any time.

The parameters which control and drive the simulation are the following.

1- Number of database granules (items): M

2- Number of transactions in the system (the degree of multiprogramming or load): T

3- Number of granules accessed by a

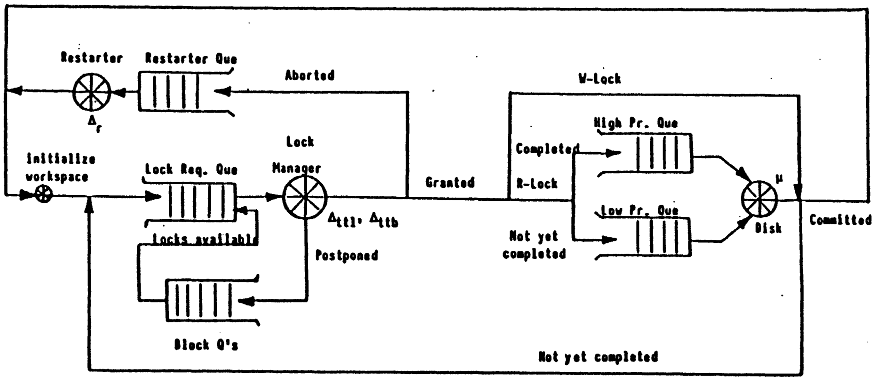


Figure 1 Performance Model of the 2PL Algorithm

transaction: I
4- Read/Write ratio: THRESHOLD
5- Disk average service rate: $\mu$
6- Lock Management CPU time delay: $\Delta_{ttl}$
7- Restart time delay: $\Delta_r$
8- Processor Power in MIPS

The following performance indices are measured for each simulation run (experiment).

1- Average transaction response time (RT) : seconds
2- System throughput (TP) : Transaction/Second
3- Degree of concurrency (DC)
4- Deadlock rate (DR) : deadlock/second
5- Conflict rate (CR) : conflicts/second
6- Maximum number of times a transaction was restarted (MXRST)

The degree of concurrency is defined as follows. Let $T_n$ be the duration of time through which the number of active transactions in the system was n. The degree of concurrency is then given by:

$$DC = 1/(T*T_s) * \sum_{n=1}^{T} n*T_n$$

where $T_s$ is the system time and T is the number of transactions in the system. Clearly, $0 < DC \leq 1$.

## 3. Simulation of the Commit-Time Validation

A fixed number of transactions are continuously cycled around the model shown in figure 2. Initially, all transactions are put on the low priority disk queue. A transaction then goes through the following stages.

(1) Each time a transaction submits a read operation, it is put on the low priority disk queue. No disk write operation takes place when a transaction submits a write operation (as the two-phase commit is employed, all writes take place at commit time). When a read or a write operation is complete, the transaction immediately submits its next read or write operation, if any.

(2) When the transaction completes, it is validated to see whether its writes could be committed or not. The transaction is put on the validator queue. When the transaction is considered by the validator, a validity test is performed.

(3) If the transaction is successfully validated, it is put on the committer queue in order to commit its writes. If the transaction is a pure query (read only), no commitment occurs and the transaction finishes and it terminates.

(4) If the validation test fails, the transaction is restarted by the restarter. The transaction workspace is reset and it reexecutes from the beginning.

(5) When a transaction is committed, it finishes execution and it then terminates. When a transaction terminates, another one instantaneously arrives at the system.

The concurrency control is modeled by a set of servers. These are the validator, the committer, and the restarter. The validator tests the completed transaction against all recently committed transactions. If the readset of the former intersects with the writeset of at least one recently committed transaction, the test fails, otherwise it succeeds. The validation service takes a time delay $\Delta_v$ to be completed. The committer inserts information about the committing transactions into the tables of the recently committed transactions and it also maintains these tables by deleting obsolete entries (entries with timestamps smaller (older) than the smallest timestamp of all active transactions are deleted). The commitment service takes a time delay $\Delta_c$ to be completed. The restarter resets the workspace of the unsuccessfully validated transaction and inserts it back into the system to reexecute from the beginning. The restarter service takes a time delay $\Delta_r$ to be completed. $\Delta_v$, $\Delta_c$ and $\Delta_r$ are estimated in a similar way as $\Delta_{ttl}$ and $\Delta_r$ of the 2PL. $\Delta_v$, $\Delta_c$, and $\Delta_r$ are all estimated to be around 0.005 seconds. Again, the concurrency

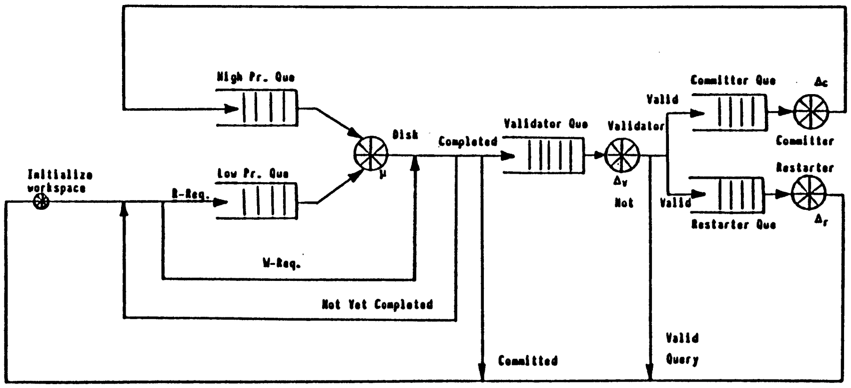


Figure 2 Performance Model of the CTV Algorithm

controller is modeled as a set of servers so that multiprocessing may be utilized.

The parameters which control and drive the simulation are the following.

1- Number of database granules (items): M
2- Number of transactions in the system (the degree of multiprogramming or load): T
3- Number of granules accessed by a transaction: I
4- Read/Write ratio: THRESHOLD
5- Disk average service rate: $\mu$
6- Validation test time delay: $\Delta_v$
7- Commitment time delay: $\Delta_c$
8- Restart time delay: $\Delta_r$
9- Processor Power in MIPS

The following quantities are measured for each simulation run (experiment).

1- Average transaction response time (RT) : Seconds
2- System Throughput (TP) : Trans./Sec.
3- Conflict rate (CR) : Conflict/Second
4- Maximum number of times a transaction was restarted (MXRST).

## 4. Unified Model Assumptions

Following is the set of model assumptions made in constructing the performance model of both the 2PL and the CTV algorithms.

(1) Two-Phase commit is incorporated with the 2PL(CTV) algorithm, thus a write operation is treated as a non-disk operation, and on transaction completion, two-phase commit begins by issuing 1 + |Writeset| disk operations(one in the first phase and |Writeset| in the second phase).

(2) Resources are finite: the physical database is stored in one disk unit with a given service rate. The case of more than one disk unit can equivalently be studied by one disk unit with a higher service rate. Also, there is only one CPU.

(3) Elements of the readset and the writeset of a transaction are distinct.

(4) Computation time needed by each transaction is negligible. This is justifiable because most transactions are I/O-bound processes.

(5) Transaction size is large. This gives a worst case results for a system with mixed sized transactions. How large a transaction is, is explained in the next section.

(6) A Transaction workspace consists of two parts: the transaction definition and the transaction data section. When a transaction is restarted, only its data section (or part of it) is reset.

## 5. Experiments and Results

In all experiments, a fixed number of transactions is cycled around the 2PL(CTV) simulator. This number represents the system load. In 2PL, a given transaction j may be inactive (blocked) or actively running. An actively running transaction may be taking a service at some server (lock manager or restarter) or else be submitting a new lock request to the simulator. This lock request is denoted $L_{ijk}$ (transaction j requests a lock on item i in mode k $\varepsilon$ [Read,Write]). In CTV, a transaction j is always actively running taking a service at some server (validator, committer or restarter) or submitting a new access request to the simulator. This access request is denoted $R_{ijk}$ (transaction j submits an access request on item i in mode k $\varepsilon$ [Read,Write]). The process of submitting an $L_{ijk}$ ($R_{ijk}$) is repeated for an "Observation Interval" number of times. Observation interval ranges from 2000 to 40,000 lock (access) requests. In the following set of experiments, the database size (M), was assumed to be a total of 640 items. The effect of granularity on the performance was investigated by studying granularities of 80, 160, 320, and 640 granules.

According to Yao [Yao77], the value of the effective transaction size changes with varying the granularity. In these experiments, the effective transaction size is almost invariant to changing granularity over [80,160,320,640] for a database of 640 items and for transactions of 15 items size.

The size of each transaction, I, was assumed to be large (15 granule lock (access) requests). This amounts to about 2.5% of the whole database (640 granules). Usually, a transaction accesses less than 1% of a database, so the 2.5% figure represents a large size transactions. This gives a worst case study of a database system with transactions of mixed sizes.

The effect of the Read/Write mix associated with the transactions was investigated by conducting experiments for mixes of 100/0, 90/10, 80/20, . . ., and 0/100.

The hardware parameters were assumed to be a disk average service rate of 20 access requests/second and one MIPS processor. These figures were fixed in all the experiments but were doubled and reduced to a half in one experiement in order to understand its effect on the performance.

In 2PL(CTV) an actively running transaction j submits an $L_{ijk}$ ($R_{ijk}$) by uniformally selecting an item i$\varepsilon$ [1 ... M] from the set of items not yet accessed by j and by selecting k$\varepsilon$ [ Read , Write ] according to a predefined Read/Write ratio (K is selected to be a "Read" with Probability less than Threshold and "Write" otherwise).

The domain of settings for all experiments conducted is listed below.

1- Number of transactions (T): [ 5,10,20,30,40 ]
2- Number of Database granules (M): [ 80,160,320,640 ]
3- Transaction size in granules (I): [ 15 ]
4- Read/Write mix (THRESHOLD): [ 100/0, 90/10,...,0/100 ]
5- Disk average service rate (μ): [ 20,40 ] access/second
6- Processor Power in MIPS : [ 1, 2 ]

### 5.1 Response time and Throughput

Response time and throughput of the 2PL and the CTV are studied under variation of three parameters. These are the system load, the transaction Read/Write mix and the granularity. In these experiments, Response time and throughput are plotted for various Read/Write mixes, while the degree of multiprogramming is fixed. This process is repeated for various degrees of multiprogramming (loads) and the resulting curves are grouped into a frame. The frame is repeated for various granularities. Thus the effect of the Read/Write mix, the effect of granularity, and the load effect are all incorporated into detailed design curves. Only response time curves are included (Fig 3).

### 5.1.1 Load Effect

In both the 2PL and CTV, increasing the system load (T) results in the following:

1- Response time linearly increases
2- Throughput linearly decreases.

This effect is shown in figure 4.

### 5.1.2 Effect of the Read/Write mix:

In both the 2PL and the CTV, the R/W mix affects the response time as follows:

1- The R/W mix has a great impact on the RT, especially when the system is loaded (T=30,40).

2- For small loads (T=10), R/W mix has a negligable effect on RT.

3- Starting from the 100% read (query) case, increasing the write percentile first impairs the RT till R/W mixes of 70/30-50/50, after which the RT improves, and in the case of the 2PL it reaches its minimum value at the pure update case.

Explanation:

The Third observation about Response Time is explained below.

(a) For query-dominant transactions (R>50, W<50), unresolvably conflicting transactions are restarted after accessing most of their requests. Restarting a mostly finished transaction contributes to an increase in the useless I/O percentile.

(b) For update-dominant transactions (R<50,W>50), since two-phase commit is incorporated with the 2PL and the CTV, no write disk operation takes place during trasaction execution and hence most of the transactions disk access operations will not be done by the time it is restarted because of an unresolvable conflict. Thus, we have the case of restarting a transaction that accessed a small fraction of its requests. This results in small useless I/O percentile. On the other hand, for update-dominant transactions, unresolvable conflict rate increases resulting in so many restarts. However, the restart operation neither contributes to high useless I/O percentile nor incurs high setup cost.

### 5.1.3 Granularity Effect

In both the 2PL and the CTV, granularity affects the response time as follows:

1- For small loads (T=10), granularities of 160,320, and 640 have almost the same RT. Granularity of 80 incurs higher resnonse time. This is further illustrated in figure 5.

2- Granularities of 80 and 160 highly impairs the RT, especially when the system is loaded (T=30,40).

3- For the 2PL, the effect of doubling the granularity from 320 to 640 (that is from M/2 to M, where M is the database size) is almost negligible. This is true for the CTV only for small loads (T=10).

### 5.1.4 Effect of Hardware Parameters

The effect of doubling the disk average service rate and the Processor speed was presented on the response time only. The observation is shown in figure 6. It is observed that doubling the disk power results approximately in reducing the response time by half, whereas doubling the processor speed results in a trivial improvement in the response time.

### 5.2 The Comparison

In this section, we compare the response time incurred by the 2PL and the CTV algorithms. In this comparison, a load of 20 transactions is fixed. However, conclusion of this comparison is similar to that for all other loads. Figure 7 depicts the response time of both the 2PL and the CTV. From these curves the following are observed:

1- CTV is better than the 2PL for granularity of 80, whereas 2PL is better than the CTV for granularities of 320 and 640. For

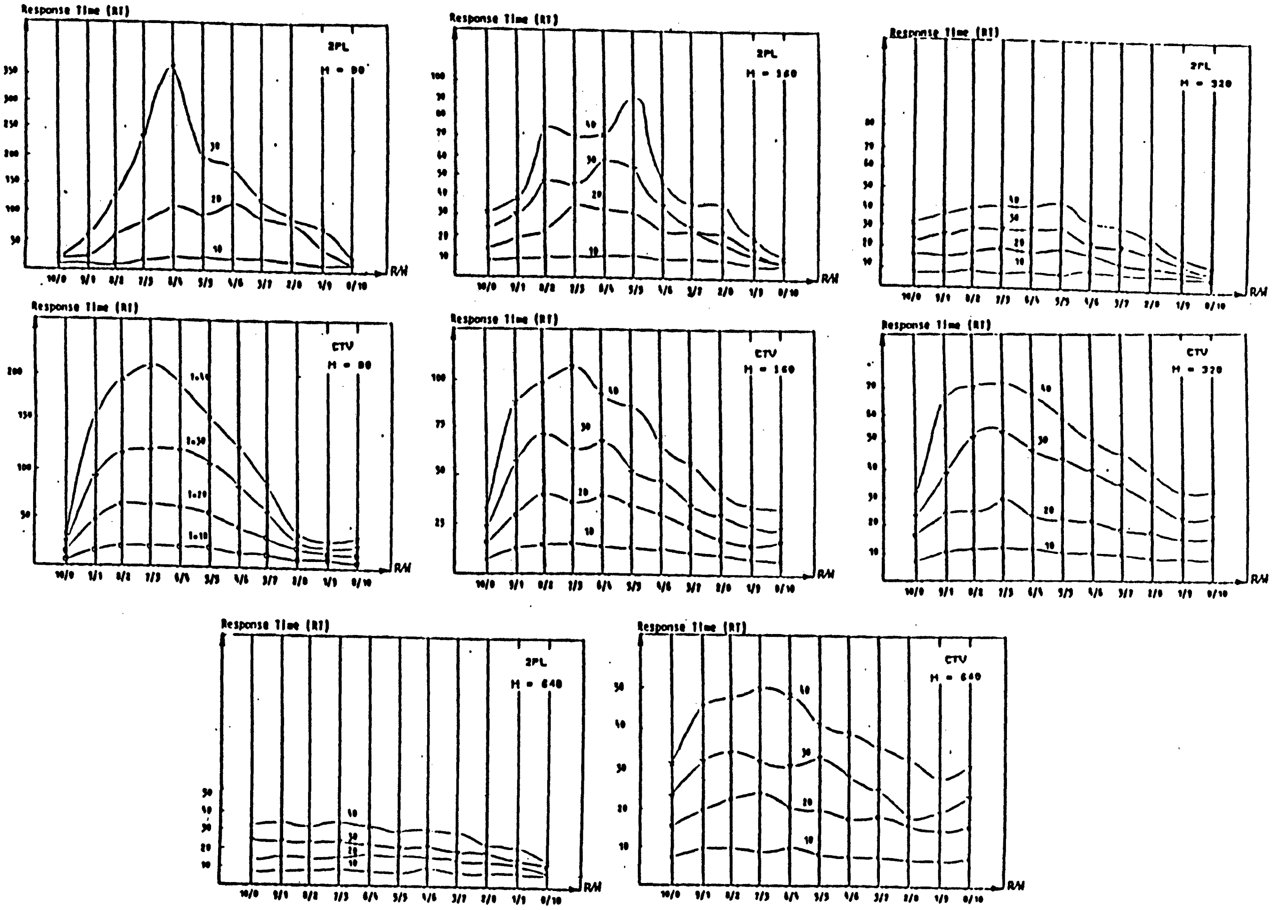


Figure 3 Response Time Design Curves

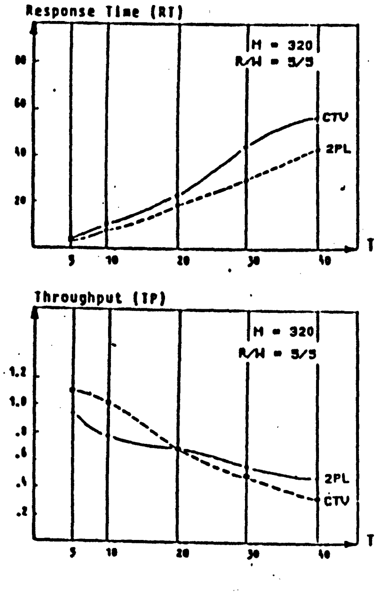


Figure 4 Load Effect

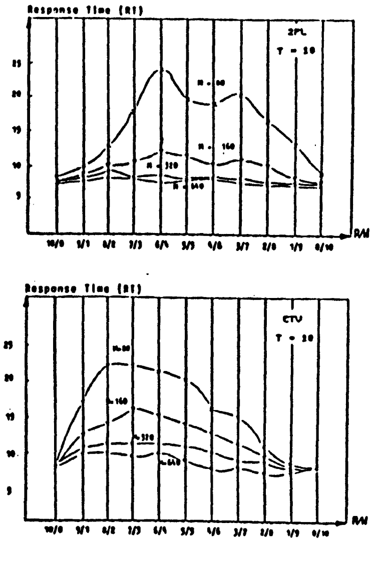


Figure 5 Small Loads

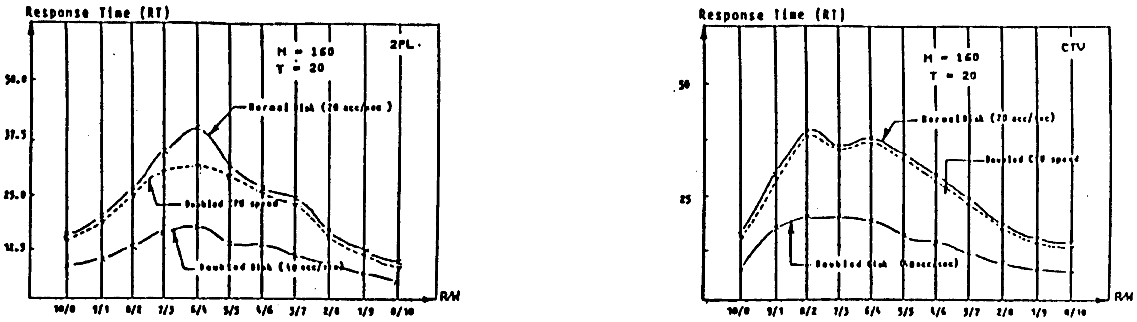


Figure 6 Effect of Hardware Parameters

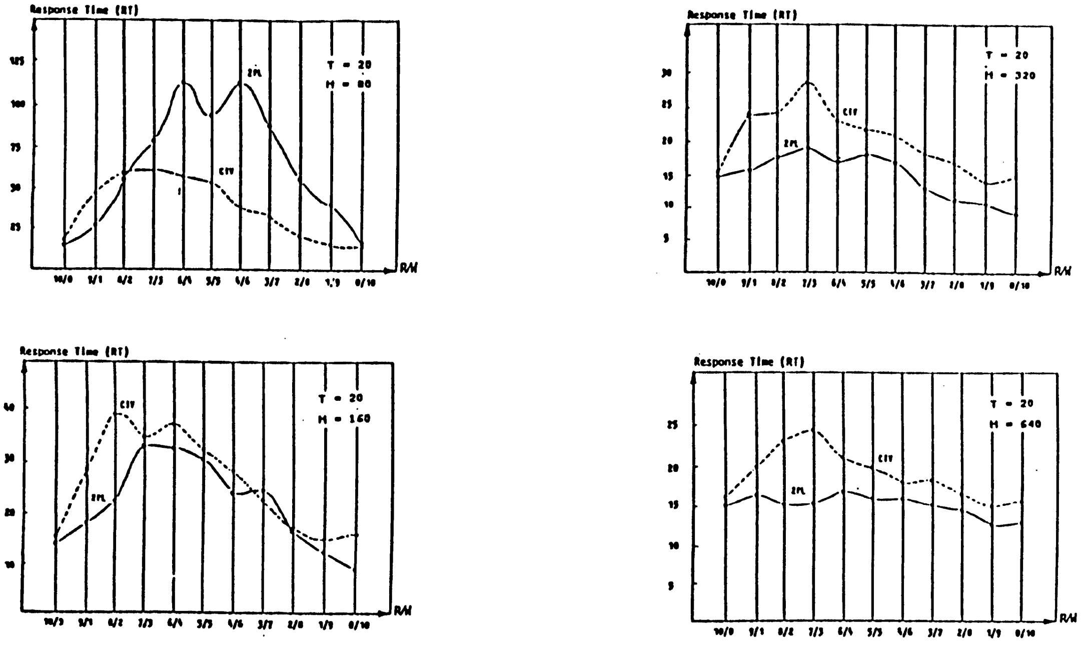


Figure 7 Comparison between the 2PL and the CTV

granularity of 160, 2PL is better for high read percentile of the R/W mix, and for write percentiles greater than 50%, the two algorithms incur approximately the same response time.

2- For granularity of 80, 2PL incurs unacceptable response time.

3- In the 2PL with granularity of 640, the R/W mix has almost no effect on the response time.

4- For high granularities, and for high read percentiles of the R/W mix, the two algorithms incur reasonable (acceptable) response time.

## 6. Conclusion

We have presented two simulation models for each of the dynamic two-phase locking and the commit-time validation. Model assumptions were unified and a comparison was made. We concluded that CTV can work better than 2PL under very coarse granularity whereas 2PL performance is not acceptable for the same very coarse granularity. On the other hand, 2PL outperforms the CTV for granularities of 320, and 640. Also, we concluded that for query dominant transactions, the two algorithms incur the same performance.

In 2PL, we have found that for large size transactions, the response time incured by a granularity M is almost the same of that incured by a granularity M/2, where M is the database size.

Under the assumption that two-phase commit is incorporated with the 2PL and the CTV, the Read/Write mix associated with transactions was found to affect the performance as follows: query dominant and update dominant transactions result in better performance than transactions with nearly equal read and write percentiles.

## Acknowledgements

We would like to thank Prof. Yahia El-hakim, Chairman of the Computer Science Department at Alexandria University, Prof. T.M. Ozsu from the University of Alberta, Dr. Jai Menon of IBM Almaden Research Center and Mr. D. Mannai for their careful reading of the manuscript and many valuable suggestions. They have improved the contents and presentation of this paper, and to them we are grateful.

## References

[AGRA 85] Agrawal, R., Carey, M.J. and Livny, M., "Models for Studying Concurrency control performance: Alternatives and implections", Proc. ACM-SIGMOD 1985, Int'l Conf. on Management of data, May 1985.

[BADA 79] Badal, D., Correctness of Concurrency Control and Implications in Distributed Database. Proc. of the COMPSAC '79, Chicago, IL, 1979.

[BAYE 80] Bayer, R., Heller, H., and Reiser, A., Parallelism and recovery in database systems, ACM trans. on DBS June, 1980.

[BERN 85] Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Database Systems", Computing Surveys, ACM, June 1981.

[CARE 83] Carey, M., "An Abstract Model of Database Concurrency Control Algorithms", ACM SIGMOD Record's Proceeding of annual meeting, 1983.

[CHES 83] Chesnais, A., Gelenbe, E. and Mitrani, I., "On the Modeling of Parallel Access to shared Data", Comm. ACM, March 1983.

[ELMA 85] Elmagarmid, A.K., "Deadlock Detection and Resoltuion in Distributed Processing Systems" Ph.D. Thesis, Dept. of CIS. The Ohio State University. Columbus, Ohio 1985.

[GARC 79] Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database", Ph.D. Dissertation, Dept. of Computer Science, Stanford U., Stanford, Calif. 1979.

[HELA 85] Helal, A.A., "Performance Analysis of Concurrency Control Algorithms in Database Systems", M.S. Thesis, Dept. of Computer Science, Alexandria University, Egypt, July 1985.

[HELA 86] Helal, A.E., Elmagarmid, A., and Hurson, A., "A Paradigm for Concurrency Control Performance Evaluation", Future Directions in Computer Architecture and Software Workshop , S. Carolina, May 1986.

[HSIA 81] Hsiao, D.K., and Ozsu, T.M., "A Survey of Concurrency Control Mechanisms for Centralized and Distributed Databases", 1981, Tech. Report OSU-CISRC-TR-81-1, Dept. of Computer and Info. Sciences, The Ohio State University.

[KUNG 81] Kung, H.T., and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM trans. on database systems, June 1981.

[MENA 78] Menasce, D., and Muntz. R., Locking and Deadlock Detection in Distributed Database Proc. of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks. August 1978.

[MITR 84] Mitra, D., and Weinberger, P.J., "Probalistic Models of Database Locking: Solution, Computational Algorithms and Asymptotics", Journal, ACM, Oct. 1984.

[RIES 77] Ries, D.R., and Stonebraker, M., "Effects of Locking Granularity in Database Management Systems", ACM trans. on database systems, Sep. 1977.

[RIES 79] Ries, D.R. and Stonebraker, M., "Locking Granularity, Revisited", ACM Trans. on Database System, June 1979.

[ROSE 78] Rosenkrantz, D., Stearns, R., and Lewis, P., System Level Concurrency Control for Distributed Database Systems. ACM Transactions on DBS, June 1978.

[SHUM 85] Shmueli, O., Spirakis, P., and Goodman, N., "A Methodology for Concurrency Control Performance Evaluation", Aiken Computation Lab., Harvard University, Aug. 1982.

[SHUM 81] Shum, A.W., and Spirakis, P.G., "Performance Analysis of Concurrency Control Methods in Database Systems.", Performance 81, North-holand Publishing Co., 1981.

[YAO 77] Yao, S. B., Approximately Block Accesses in Database Organization. Comm. ACM, April, 1977.

# INCREASING PROCESSOR UTILIZATION DURING PARALLEL COMPUTATION RUNDOWN

William H. Jones

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Abstract -- Some parallel processing environments provide for asynchronous execution and completion of general purpose parallel computations from a single computational phase. When all the computations from such a phase are complete, a new parallel computational phase is begun. Depending upon the granularity of the parallel computations to be performed, there may be a shortage of available work as a particular computational phase draws to a close (computational rundown). This can result in the waste of computing resources and the delay of the overall problem.

In many practical instances, strict sequential ordering of phases of parallel computation is not totally required. In such cases, the "beginning" of one phase can be correctly computed before the "end" of a previous phase is completed. This allows additional work to be generated somewhat earlier to keep computing resources busy during each computational rundown. This paper identifies the conditions under which this can occur, reports the frequency of occurrence of such overlapping in an actual parallel Navier-Stokes code, suggests a language construct, and discusses possible control strategies for the management of such computational phase overlapping.

## Introduction

General purpose parallel computations are usually divided into phases that must execute sequentially in order to guarantee algorithmic integrity. For instance, the checkerboard approach to the successive over-relaxation solution of the potential field problem divides into two such phases: the "odd" locations phase and the "even" locations phase. On the parallel phase level, the iterated values of the previous phase must be complete before the new values of the next phase can be correctly computed.

In the checkerboard algorithm, the execution time of each location is definite (nominally, the time for four additions and a divide). Thus, the distribution of work among processors can be accurately planned. Under ideal conditions (involving the number of checkerboard locations in comparison to the number of processors), the distribution of work can be arranged so that each processor shares an exactly even portion of the work and, as a consequence, each processor completes its work at exactly the same time. Perfect computation resource utilization is realized (at least in a practical sense) since the next computational phase can begin immediately.

Unfortunately, ideal conditions are infrequently found in real applications. Continuing with the checkerboard algorithm, consider the situation when the potential grid is 1024 points on a side (2**20 grid points) and 1000 processors are available. Each computational phase will provide 524, 288 individual computations, or 524 computations for each of the 1000 processors; however, 288 computations will be left over for distribution among the 1000 processors. This will leave 712 processors with nothing to do while the final 288 computations are carried out.

The burden of experience gained by the author suggests that even this example is optimistic. Most computations carried out by the author's parallel Navier-Stokes solver (the Combined Aerodynamic and Structural Dynamic Problem Emulating Routines or CASPER [1] which was controlled by the Parallel, Asynchronous Executive or PAX [2]) could not even be ascribed with definite execution times. In some instances, whether or not the computation was even to be carried out in a particular instance was a conditional part of the algorithm. No control over the computation-count-to-processor ratio was attempted -- processors were allocated as they became available on a the-more-the-merrier basis. Also, shared information access times were unpredictable and unrepeatable from instance to instance. As a result, there was no assurance that individual processors could be kept busy as a particular computational phase drew to a close.

The PAX/CASPER project provided the experience base cited later in this paper. PAX/CASPER was focused on a parallel, general purpose, Navier-Stokes solver. Thus, this experience base is presented not as a grand generalization for all of parallel processing, but as a specific example in practical parallel processing.

Certain other situations that might seem of interest in the overlapping of computational phases (for instance, the possibilities for overlapping in a tight iterative loop) are not treated for the simple reason that they did not occur in the PAX/CASPER project. PAX/CASPER was not so much a research project in parallel processing as an exploratory development of a far-term aerodynamic tool. Thus, the motivation was to solve the problems that occurred rather than to solve the problems that one could imagine.

It has been suggested that scheduling and overhead problems will be a particular problem in PAX/CASPER. So far, this has not been the case. Operational experience shows that the ratio of computation to management has been running at something in the neighborhood of 200. This paper is an effort to chart a method of improving upon this situation so as to stave off any backsliding that might occur as the ratio of computational to management resources increases. There are additional strategies which have been identified for development. These include a middle management scheme to parallelize the serial management function, a direct worker-to-worker lateral communication scheme, and a data-proximity work assignment algorithm. These strategies combined with the overlapping of computational phases should enhance the management overhead situation.

Various solutions to the computational rundown problem may be acceptable. Some parallel processing schemes for general purpose computation may choose simply to accept the lower processor utilization as a minor design flaw. Another alternative is to create a multi-parallel-job-stream environment that allows computational work of one job stream to fill in when another job stream enters a computational rundown situation. This will bring processor utilization up; however, it should be recognized that the primary goal of parallel processing is to reduce elapsed wall-clock time for a given job. The introduction of such a "batch" environment will inevitably distribute processor resources among the several job streams and, thus, reduce the total processing power on any particular job and lengthen its elapsed wall-clock time.

### Overlapping Computational Phases

The goal then is to find more ready-to-compute work from the parallel algorithm that is being computed. As mentioned previously, this is not possible at the parallel phase level: each phase must be completed before the next is begun in order to guarantee algorithmic integrity; however, if an examination is made at a deeper (sub-phase or, in the terminology of the author, task) level, it is frequently discovered that the completion of portions (tasks) of one phase will allow the correct computation of portions (tasks) of the succeeding phase.

Consider again the checkerboard algorithm. If all the "odd" locations adjacent to a particular "even" location have been updated with new values from the current computational phase, then the new value for that particular "even" location for the next computational phase can be correctly computed. Additionally, since all the computations requiring as an input the current value of that particular "even" location have been completed, the value for that "even" location can be updated without affecting the results of the current computational phase.

At this point, it is necessary to make certain assumptions (or, alternatively, set certain system design constraints) about the nature of computational phase rundown. Two basic situations arise: one in which task assignments and releases are statically determined and one in which such matters are dynamically determined.

The static situation is much simpler from the standpoint of next-phase task release timing since everything is determined ahead of time. In this case, it can be acceptable for computational rundown to begin almost immediately since the scheduling of the next-phase task has already been statically determined. No completion processing of current-phase tasks is required to schedule the release of the next-phase task. (In fact, work in this area for the purposes of real-time simulation has been conducted for some years at NASA Lewis [3, 4]).

The dynamic scheduling situation is substantially more interesting. Some time delay must be available between the completion of the first current-phase tasks and the onset of computational rundown. This delay is needed to provide time to process the completion of the early current-phase tasks and, in so doing, schedule the next-phase tasks that are thus enabled. During this delay, there must be enough current-phase tasks to keep the processing resources busy in order to avoid a computational load dip while the next-phase tasks are scheduled.

In the dynamic scheduling situation, enablement relationships between the current-phase tasks and the next-phase tasks (i.e., the relationship that enables a next-phase task based upon the completion of a current-phase task) may be either static or dynamic. That is, the completion of a particular current-phase task may always enable the same next-phase task (the static enablement case) or it may enable some next-phase task that can only be identified at the time of execution (the dynamic enablement case). The nature of the enablement relationships is important because it is involved in setting the time delay from the completion of the first current-phase tasks to the availability of the first enabled next-phase tasks.

Considering these characteristics of the dynamic scheduling situation (i.e., the time to process current-phase task completion, the time to recognize enablement relationships, and the time to schedule enabled next-phase tasks), it can be observed that the number of tasks should substantially outnumber the number of processors. Certainly, there should be at the outset of the current-phase work at least two tasks for each processor so that at least one task execution time will be available to process the completion of the first task assigned to the processor and to schedule the enabled next-phase task. This presumes that completion processing and task scheduling time is small with respect to task execution time. In particular, it assumes that one such completion, enablement, and scheduling cycle for each of the processors in the system can be completed in a single task execution time. (The author's experience with PAX suggests that this is reasonable even for dynamic managerial style parallel processing systems. Systems that use hardware-level synchronization primitives presumably would be at even greater advantage in this area.)

The conditions under which this overlapping of computational phases can correctly occur are the same as those that allow parallel computations within a particular phase. Let the logical predicate PARALLEL(x,y) return the condition TRUE when x and y are such that parallel computations are allowed. Clearly, PARALLEL(n,m) must always be TRUE if n and m are distinct computational granules of the same parallel computational phase. Let q be an uncompleted granule of the current phase and r be a granule of the next phase that has been enabled by some completed granule, p, of the current phase. If PARALLEL(q,r) necessarily returns the value TRUE, then the current-phase and next-phase can be correctly overlapped.

The exact nature of the logical predicate PARALLEL(x,y) is, of course, of substantial practical interest; however, it has no direct impact upon the ability to overlap phases as outlined above. Different parallel systems may identify different logical predicates.

### Identifying Enabled Granules

The first challenge to be met is to find a way of identifying enabled next-phase granules for overlapping. It is easy to postulate that some mapping function exists either to map from the set of completed granules, p, to the set of enabled granules, r, or to map from the set of uncompleted granules, q, to the set of enabled granules, r. It is very difficult to establish what this mapping function might be in any general way. Fortunately, this mapping function is much more easily identified when each concrete situation is faced.

First, consider the simplest imaginable case as represented by the following Fortran code segment:

```
      ...
      DO 100 I=1,N  ! First computational phase
      B(I)=A(I)     !
  100 CONTINUE      !
      DO 200 I=1,N  ! Second computational phase
      D(I)=C(I)     !
  200 CONTINUE      !
      ...
```

Assuming that there are not shared output area constraints, it can be observed that these two parallel computational phases can be computed in parallel with each other. This represents what might be called a universal mapping function wherein any granule of the second computational phase is enabled by any granule or set of granules (including the null set) of the first computational phase.

PAX/CASPER experience shows that 6 out of 22 (or 27 percent) of the parallel computational phases allow universal mapping enablement of the succeeding phases. This represents 266 out of 1188 lines (or 22 percent) of the code that is executed in parallel in PAX/CASPER.

This universal mapping usually occurs in PAX/CASPER when the nature of the larger computational process is changing. For instance, the change over from power of compression computations to interpolator matrix generation is one such character change. The two computations do not involve shared information of any kind and, thus, they can be entirely overlapped. Of course, the two phases could be merged into one by a preprocessor of the parallel control stream; however, since the mechanisms necessary to handle this case would be a subset of those needed for the following case, it might well be simpler to support this enablement mapping.

For the next case, consider the following Fortran fragment that is to be computed in parallel as two succeeding computational phases:

```
      ...
      DO 100 I=1,N  ! First computational phase
      B(I)=A(I)     !
  100 CONTINUE      !
      DO 200 I=1,N  ! Second computational phase
      C(I)=B(I)     !
  200 CONTINUE      !
      ...
```

Again assuming that there are not shared output area constraints, it can be observed by inspection that the identity mapping function (I = I) maps from completed granules, p, to enabled granules, r. This is also a simple and easily identified mapping.

PAX/CASPER experience indicates that it applies in 9 out of 22 (or 41 percent) of the parallel computational phases (representing 551 of 1188 code lines, or about 46 percent of the parallel code in PAX/CASPER). Combining this direct mapping with the simpler universal mapping above indicates that (at least in PAX/CASPER experience) 68 percent of the parallel computational phases and 68 percent of the code executed in parallel can be easily overlapped to defeat computational rundown. These two enablement mapping possibilities are the most frequently occurring situations in PAX/CASPER experience.

The next most frequently occurring enablement mapping in PAX/CASPER experience is what could be called null mapping, that is, the situation in which no overlapping is possible. This occurs in 4 out of 22 (or 18 percent) of the computational phases and represents 262 out of 1188 (or 22 percent) of the lines of code executing in parallel. In all cases the cause was not that such an overlapping did not exist between the parallel computations but was, in fact, that serial actions and decisions had to occur between the phases. This is important since it allows one to assess how often the extra effort of supporting overlapping features will be entirely defeated, regardless of the sophistication of the overlapping phase support features.

Another enablement mapping occurring in PAX/CASPER experience is a reverse indirect mapping. Consider the following Fortran fragment:

```
      ...
      DO 10 I=1,N         ! Set up source mapping
      DO 10 J=1,10        !
      IMAP(J,I)=IRAND()   ! IRAND produces an integer
   10 CONTINUE            !   in the range 1 to N
      DO 100 I=1,N        ! First computational
      A(I)=FUNC(I)        !  phase generates some
                          !  number in A(x)
  100 CONTINUE            !
      DO 200 I=1,N        ! Second computational
      DO 200 J=1,10       !  phase sums subsets of
      B(I)=A(IMAP)(J,I))! the results of the
                          !  first computational
                          !  phase
  200 CONTINUE            !
      . . .               !
```

Clearly, this computation can be overlapped; however, determining the enablement mapping is very difficult. This is because knowing that a particular first phase granule is complete does not directly identify any distinct second phase granule as computable; however, a reverse mapping from desired second phase granule to required first phase granules is possible.

In PAX/CASPER experience, this situation occurs in 2 of 22 (or 9 percent) of the computational phases representing 78 out of 1188 (or 7 percent) of the lines of code executing in parallel. While this is not a frequently occurring situation in PAX/CASPER experience, it cannot be ignored out of hand. Some engineering judgement must be made to weigh the cost (in terms of management overhead, computational resource transferred from workers to management, etc.) of some reverse enablement mapping solution against the cost of computational rundown in 9 percent of the parallel computational phases.

Certainly, a solution exists for the reverse, indirect enablement mapping. Once the values of the information selection map (represented in the code fragment by the array IMAP) have been determined, it is a simple matter to produce a composite map of first phase granules that must be completed in order to enable a particular second phase granule. The executive can then use this map upon each first phase granule completion to determine the computability of particular second phase granules. This map could also be used to direct a preferred order of first phase granule dispatching so as to enable a known second phase granule as early as possible.

Two important facts about this reverse enablement mapping must be included. First, both occurrences of this situation involved a dynamically generated information selection map. Thus, the composite granule map would have to be generated by the executive at or after first phase initiation but before any second phase enablements. Second, the impact of executive computation must be considered. In the PAX/CASPER UNIVAC 1100 test bed, executive computation was done at the direct expense of worker computation. Thus, extensive composite granule map generation could be self defeating. Some real parallel machines may provide separate executive computing resources, in which case the generation and use of composite granule maps would not be out of the question.

A final enablement form was observed in PAX/CASPER that could be characterized as a forward, indirect mapped situation. Consider the following Fortran fragment:

```
      DO 10 I=1,M           ! Generate forward
      IMAP(I)=IRAND()       !  map
   10 CONTINUE              !
      DO 100 I=1,M          ! Use forward map
                            !  to operate on a
      B(IMAP(I))=A(IMAP(I)) !  subset of the
  100 CONTINUE              !  arrays
      DO 200 I=1,N          ! Perform some further
      C(I)=B(I)             !  further opera-
  200 CONTINUE              !  tion on the
                               complete arrays
      ...
```

This situation is somewhat easier than the reverse, indirect mapping in that the identification of a particular granule in the first phase can be directly mapped to an enabled granule in the successor phase; however, much of the complication of a mapped enablement remains. This form was the least frequently occurring situation in PAX/CASPER showing up only once (5 percent of the phases) and accounting for only 31 of 1188 lines of code executed in parallel.

No other forms of enablement mapping were observed in PAX/CASPER. Certainly, extensions of the forms already presented can be imagined. Additionally, a seam mapping problem (such as would be appropriate for the checkerboard approach to the successive over-relaxation problem) can be foreseen. These other forms are beyond the scope of the present paper.

## Language Construction

The developing PAX/CASPER language is simple and requires the user to make specific statements concerning choices for the management of each parallel computational phase. Statements involving the enablement of a succeeding phase could be made at two times: during the definition of a computational phase to the management system and during the invocation of the phase for actual computations. The difficulty to be faced is that the statements no longer apply solely to the phase being referenced, but rely also on the characteristics of the succeeding phase.

The simplest approach is to require the user to specify the appropriate enablement mapping method when the phase in invoked. It might appear as in the following PAX parallel language fragment:

```
...
DISPATCH  phase-name
          ...
          ENABLE/MAPPING=option
          ...
```

This is simple and explicit; however, it leaves the door wide open to user mistakes. There is no interlock between this phase and the next that can be verified by the executive. A simple solution to this would be to identify the name of the enabled next phase so that the executive system (or language processor) can verify that, in fact, that phase is following. This might appear as follows:

```
...
DISPATCH  phase-name
          ...
          ENABLE [phase-name/MAPPING=option]
          ...
...
```

This allows the desired verification, but also brings up a new possibility. Occasionally, a conditional branch that is not dependent on the computational phase separates that phase from two or more succeeding phases, each of which may (or may not) be overlappable. If each of these phases were identified in the above construct, the executive could preprocess the branch and overlap the appropriate phase. This could look as follows:

```
...
DISPATCH phase-name
         ...
         ENABLE/BRANCHINDEPENDENT
          [phase-name-1/MAPPING=option
            phase-name-2/MAPPING=option]
         ...
IF       (IMOD(LOOPCOUNTER,10).NE.0)
THEN     GO TO branch-target
DISPATCH phase-name-1
         ...
GO TO    rejoin
branch-target:
DISPATCH phase-name-2
rejoin:
...
```

Finally, the matching of mapping selections and phases and the invocation of the appropriate overlapping services is something that could be done when the parallel phase is defined to the system; however, it would still be necessary to identify pre-processable branches at the computation invocation site. This could appear as follows:

```
DEFINE   PHASE phase-name
         ...
         ENABLE [
                phase-name-1/MAPPING=option
                phase-name-2/MAPPING=option
                phase-name-3/MAPPING=option
                ]
...
DISPATCH phase-name
         ...
         ENABLE/BRANCHDEPENDENT
...
```

The ENABLE/BRANCHINDEPENDENT would be deleted when branch pre-processing was either not appropriate or not needed. The executive system could perform the appropriate lookahead to see whether any of the named succeeding phases was actually following and apply, as appropriate, the specified enablement mapping.

## Control Strategies

Control strategies for enabling and scheduling overlapped parallel computational phases are, of course, highly dependent upon the overall parallel processing strategies. As alluded to earlier, some approaches to parallel processing may do all of this before any computations are begun. Indeed, the entire process may be done manually by a human being when the pattern of parallel processing is fixed for the life of the system.

Within the PAX system, the opposite is true: the identification and scheduling of computable granules is entirely automatic. A scheduling mechanism for enabled computational granules already exists within the PAX system. It was developed to schedule dynamically created computations that conflicted (usually in terms of shared data access) with pre-existing computational granules.

Within PAX, each internal description of one (or more) computational granules included a queue head for a double circularly-linked list of computable but conflicting computational granules. Upon completion of the described computation, all the queued conflicting computations became unconditionally computable and were placed in the waiting computation queue. The waiting computation queue was kept in a known order and, for the purposes of the conflicting computation problem, it was determined that such conflicting computations would be placed ahead of the normal computations in the queue and, thus, given higher priority.

The scheduling of universally mapped successor phases within this system is very easy indeed. At the time of phase initiation, the successor phase is also initiated and the resulting computation description placed in the waiting computation queue behind the current phase description.

The scheduling of directly enabled successor phases is similarly easy at first sight. At the time of phase initiation, the successor phase is also initiated and the resulting computation description placed in the conflicted computation queue of the current phase description. Thus, when the current phase computation is completed, the now-enabled successor computation will be placed in the waiting computation queue to be considered for scheduling.

The above approach for directly enabled successor phases is fine if each indivisible granule of computation is described separately. Unfortunately, this is usually not economical (in terms of storage space and task search times, among other things) and was not the choice taken in PAX design. Computations were, instead, described as large, contiguous collections of granules. The descriptions were split apart as necessary to produce conveniently sized tasks for workers and then merged back into single descriptions when the work was completed. This splitting of descriptions reqiuires that queued computation descriptions also be split so that each queued description will accurately reflect the enablement relationship between the computation and its queued successor computation.

While this is certainly possible, it forces a further design decision for the executive software. PAX computation splitting was demand-driven by the presence of an idle worker. It was felt that the delay while splitting a task description was acceptable; however, the additional delays of splitting queued successor computation descriptions may represent an unacceptable situation. Two possible solutions exist. One possibility is to pre-split the tasks before idle workers present themselves to the executive. This would allow the executive to work ahead in otherwise idle time. Alternatively, the splitting of a computation could generate a successor-splitting task that could be quickly queued for later attention when the executive would again be idle.

The successor computation description could be removed from the current computation description and included in the successor-splitting task information. When the successor-splitting task is executed the successor computation could be split and requeued to the appropriate current computation descriptions.

Management of indirectly (both forward and reverse) mapped successor computations is a good deal more interesting. The description of the successor computation cannot simply be queued to the description of the current computation since there is no guarantee of the enablement relationship. Additionally, it would seem wise to get the current phase into execution without the delay of constructing the necessary information for enabling successor computations. Both forward and reverse indirection would seem well handled by much the same mechanisms since the only significant difference is the direction of the indirection. Each leads naturally to a list of current phase granules that must be completed to enable a particular successor phase granule.

It would seem appropriate to identify a subset group of successor-phase granules that are to be the subject of the enablement operation so as to avoid solving an unnecessarily large enablement problem. Once this subset has been identified, the current-phase granules that enable the successor subset can be identified. Since these are not necessarily the current phase granules that would be naturally selected by the scheduling mechanism, they should be split into individual descriptions and placed in the waiting computation queue in such a manner as to elevate their computational priority.

It is important to note that the description of the successor subset cannot simply be queued to any one of the identified current-phase granules since it is enabled not by the completion of any one such granule but by the completion of all the identified granules. This enablement on completion of all identified current-phase granules can be handled by any number of simple mechanisms. For instance, during completion processing, a status bit (set when the current-phase granules were identified and split into individual descriptions) can be checked and, if it is set, an enablement counter decremented. When the enablement counter reaches zero, it can be taken as a signal that the successor-phase granules are computable.

## Concluding Remarks

This paper has discussed the possibilities for overlapping parallel computations in a general purpose parallel-computation environment so as to minimize loss of computational resources. Practical experience with PAX/CASPER, a parallel Navier-Stokes solver, suggests that simple and plausible steps could provide such overlapping in 68 percent of the computational phases and that, with extended effort, more than 90 percent of the computational phases are amenable to some form of phase overlapping.

## References

[1] W.H. Jones, "Combined Aerodynamic and Structural Dynamic Problem Emulating Routines (CASPER): Theory and Implementation, NASA TP-2418, 1985.

[2] W.J. Jones, "Parallel, Asynchronous Executive (PAX): System Concepts, Facilities, and Architecture," NASA TP-2179, 1983.

[3] D.J. Arpasi, "Real-Time Multiprocessor Programming Language, (RTMPL) User's Manual," NASA TP-2422, 1985.

[4] D.J. Arpasi and E.J. Milner, "Partitioning and Packing Mathematical Simulation Models for Calculation on Parallel Computers," NASA TM-87170, 1986.

# A SYSTEM FOR COMPUTING THE SPEEDUP OF PARALLEL PROGRAMS

**Bruce P. Lester**

Department of Computer Science
Maharishi International University
Fairfield, Iowa 52556

## ABSTRACT

New analytical techniques are presented for probabilistic systems with concurrency and applied to computing the expected execution time and speedup of parallel programs. These new techniques are formulated for a program description language called PEL (Performance Evaluation Language), which allows essential control and timing characteristics of parallel programs to be expressed in a precise manner. The analysis techniques for PEL are an extension of standard linear system techniques to the realm of concurrency, based on the development of a new mathematical operator called the "join product." Efficient approximation techniques are also introduced for estimating the execution time and speedup of parallel programs.

## 1. INTRODUCTION

Since the main purpose of concurrency is improved performance, it is important to have analysis tools for evaluating the expected performance of various concurrent algorithms with respect to their sequential counterparts. One practical analysis technique is to simply write the program and run it on the target computer. However, the execution time for different input data may vary considerably, and it may be difficult to generalize the results from a small number of data sets. Also, the execution time will be highly dependent on uncontrollable environmental factors such as system load and placement of data files on the disk. Another common analysis technique is to do a direct complexity analysis of the abstract algorithm itself. This method has proved useful in computing overall relative complexity of algorithms in terms of "big O" properties, but is not exact enough for performance prediction of real programs. Also, this type of complexity analysis is difficult for large programs, especially those involving disk access.

This paper presents an alternative analysis technique for concurrent programs which is somewhere in between these two techniques. The actual program is written out in a special language which allows the programmer to specify the estimated execution time of primitive operations, and to model the effect of differing input data with conditional branching probabilities. One of the earliest efforts to use probabilistic analysis techniques for computer programs is found in a paper by Ramamoorthy [1], in which traditional flow graph techniques for Markov processes are used to estimate the execution time of programs expressed as flow charts, with conditional branches represented by probabilities. Using traditional z-transform techniques, the complete probability distribution for the execution time of the whole program is calculated, from which important statistics such as average execution time and variance are easily computed. Deo [2] gives an efficient technique for finding the expected execution time using a stochastic model of a program in conjunction with the standard mean first passage time techniques used for Markov processes [3]. Similar techniques are also presented by Trivedi [4]. However, none of these sequential program techniques is directly applicable to programs with concurrency.

With the increasing importance of parallel computation, there is a growing body of research on performance evaluation of parallel systems. Most of this research uses timed or stochastic Petri nets as the underlying model for the parallel system. Research results are encouraging and show that analytical or simulation techniques can be used to analyze the behavior of parallel systems represented as a Petri net with timed transitions and probabilistic choices. Zuberek [5] and Molloy [6] present techniques and examples of application to communication protocols, and Marsan [7] applies Petri net techniques to performance evaluation of multiprocessor systems. The disadvantage of these techniques is they are based on generating the reachability graph of all possible states of the Petri net, which may be exponential in the size of the net. Thus, these techniques are not practical for analyzing concurrent computer programs, typically involving large numbers of potentially concurrent operations. Sahner and Trivedi [8] describe a tool for performance evaluation of concurrent systems called SPADE, which uses an acyclic directed graph to model precedence constraints among events. SPADE is useful for analyzing concurrent program task structures, but is not general enough for detailed internal analysis of concurrent programs.

## 2. PERFORMANCE EVALUATION LANGUAGE

For the purpose of performance evaluation of computer programs, a simple language called PEL (Performance Evaluation Language) is used to represent important characteristics of the program, including the basic control structure of the program, concurrency, operation execution times, conditional branching probabilities, and loop repetition counts. A description of a program in PEL may be written directly by the programmer, or created by a compiler from the source code of the original program. If the compiler technique is used, then the programmer must supply additional information about branching probabilities and loop repetition counts in the program. Individual operation execution times may be already known to the compiler or may also be specified by the programmer.

0190-3918/86/0000/0145 $01.00 © 1986 IEEE

A similar technique is used by the Parafrase system [9] at the University of Illinois to measure program execution times for the purpose of analyzing speedups resulting from various types of program restructuring techniques that create concurrency. In the Parafrase system, the user may specify conditional branching probabilities in Fortran programs with special "assertions". If no assertion is made concerning a given branch, then Parafrase assumes that all branches have equal probability. Parafrase also estimates the number of iterations of each DO-loop, with the help of user assertions and user-definable global default values. Estimates for operation execution times such as arithmetic operations, logical operations, subscript calculations, store operations, and memory fetches are built into Parafrase, and thus not specified by the user.

The syntax of PEL is quite simple with five basic types of statements: *delay, if-then-else, sequence, loop,* and *fork-join.* The three statement types *if-then-else, sequence,* and *loop* correspond to the traditional control structures found in any block-structured language like PASCAL. The *delay* statement is used to model the execution time delays associated with primitive operations in the program such as arithmetic or assignment operations. The *fork-join* statement is used as the basic control structure for concurrency. PEL is used to create a probabilistic model of the program control structure and timing delays, in order to evaluate the overall execution time of the whole program. PEL programs have no input data, and their execution is based entirely on probabilities, which are supposed to represent average values for typical input data sets of the original program. Figure 1 shows an example of a simple PEL program.

```
PROGRAM;
  IF RANDOM < .3
    THEN  BEGIN  DELAY(5); DELAY(2); DELAY(7)  END
    ELSE
      FORK
        IF RANDOM < .5 THEN DELAY(20)
                 ELSE DELAY(35);
        LOOP 10  DO  DELAY(3);
        IF RANDOM < .4 THEN DELAY(5)
                 ELSE DELAY(10)
      JOIN;
END.
```

**Figure 1 - A Sample PEL Program**

Each time an if-then-else statement in PEL is executed a new random number between 0 and 1 is generated (represented by the word "RANDOM") and then compared with a constant, which represents the probability of choosing the "THEN" clause. In the above program, the outer "IF" chooses the "THEN" clause with probability .3 and the "ELSE" clause with probability .7 . The "THEN" clause consists of a sequence of three "DELAY" statements whose execution time is given in parenthesis. The "ELSE" clause contains a fork-join statement which executes three separate statements concurrently. The loop statement executes its body the number of times specified by the fixed constant loop count (in this example the loop count is 10). Statements may be nested to an arbitrary depth as in ordinary block-structured languages like PASCAL.

The use of fork-join primitives for specifying concurrency in programming languages originated with Dennis and VanHorn [10], and has become a standard which is found in many languages and operating systems today [11]. There are three types of fork-join statements in PEL to represent some of the major types of concurrency primitives used in existing programming languages: *fork-list, fork-n,* and *fork-gen.* The *fork-list* statement is simply a list of statements which are all executed concurrently — this is similar to the "cobegin-coend" concurrency primitive found in languages such as Concurrent Pascal [12], or the "PAR" constructor in the parallel programming language OCCAM [13]. The *fork-n* statement has the following form:

FORK(n):  statement ; JOIN   (where n is any positive integer)

The *fork-n* statement creates n concurrent executions of its statement body, and is used to model the "forall" statement which is very common among concurrent programming languages such as SISAL [14], ARGUS [15], and OCCAM [13]. The major use of this "forall" construct is as a parallel form of a DO-loop for doing highly parallel operations on vectors and arrays, and is found in many parallel array languages [16].

The *fork-gen* statement in PEL consists of a list of labeled statements along with a partial ordering giving the execution precedence constraints. Such partial orderings result when flow analysis is performed on ordinary sequential programs by a compiler to automatically locate potential parallelism (see Kuck [17]). Figure 2 shows an example of a *fork-gen* statement:

```
FORK
  1: LOOP 5  DO DELAY(4);
  2: IF RANDOM < .6  THEN DELAY(23)
               ELSE DELAY(10);
  3: DELAY(5);
  4: DELAY(3);
  5: IF RANDOM < .7  THEN DELAY(15)
               ELSE DELAY(5)
PREC
  1 < 4 ; 2 < 3 ; 2 < 5 ; 3 < 4
JOIN
```

**Figure 2 - A Fork-gen Statement**

The statement labels are used in the "PREC" section to specify the precedence constraints for executing the statements. The symbol "<" is used to represent an execution precedence constraint and may be interpreted as "is an immediate predecessor of." For example, in the above *fork-gen,* statement 1 and 2 may be executed concurrently, but statement 4 may begin only after 1 is completed and statement 2 must precede both 3 and 5. The PREC list for this *fork-gen* statement defines the execution precedence graph shown in figure 3.

## 3. EXECUTION TIME OF PROGRAMS

A system has been developed to compute the probability distribution for the overall program execution time of any PEL program, including the average execution time and the speedup resulting from concurrency. In PEL, all of the execution time delays in the program are assumed to be represented by the delay statements. The control operations such as IF branching, LOOP, FORK, and JOIN are assumed to involve no time delays. However, if it is desired for the PEL program to model time delays in these statements, then delay statements may be added to account for actual execution times of these control operations.

The program execution time is represented by a probability mass function $f(t)$, which gives the probability that the program executes in exactly $t$ time units. For simplicity only integral values of $t$ are permitted. (Fractional values of $t$ can be accomodated by using a smaller time unit.) The function $f$ satisfies the usual rule for probability distributions that the sum of all probabilities gives 1:

$$\sum_{t=0}^{\infty} f(t) = 1$$

For the purposes of computing the overall program execution time, the probability mass functions will be represented by their geometric transform:

$$f^g(z) = \sum_{t=0}^{\infty} f(t)z^t$$

In the literature on probability theory, this is sometimes called the z-transform, discrete transform, or generating function. Transform techniques are useful in the analysis of all stochastic processes, especially Markov processes. The value of the transform is that it turns a stochastic process into a linear system, which can then be analyzed using a powerful set of linear system techniques such as flow graph analysis [3]. When parallelism is allowed, however, then the system is no longer strictly *linear*, and so the standard linear flow graph techniques are not directly applicable. To analyze PEL programs, we have developed new flow graph techniques which can be used to analyze probabilistic systems with parallelism.

For any PEL statement $S$, $T(S)$ is used to denote the transform of the probability mass function for the execution time of $S$. The transform for any PEL statement may be computed recursively from its component statements according to the following rules:

1. If S is a delay statement of the form $DELAY(t)$, then $T(S) = z^t$.
2. If S is a sequence statement of the form $BEGIN\ S_1;S_2;...;S_n\ END$, then $T(S) = T(S_1)T(S_2)...T(S_n)$.
3. If S is an if-then-else of the form $IF\ RANDOM<p\ THEN\ S_T\ ELSE\ S_F$, then $T(S) = p\ T(S_T) + (1-p)\ T(S_F)$.
4. If S is a loop statement of the form $LOOP\ n\ DO\ S_1$, then $T(S) = [T(S_1)]^n$.

These rules for the sequential portions of the program are derived from the standard properties of geometric transforms. Similar formulas are given in Appendix E of ref. [4] for the Laplace transforms of structured program control statements. However, in order to compute the transform for the concurrent fork-join statements, an entirely new operation on transforms is needed called a "join product."

*Definition*: The *join product* (denoted $\circ$) is a binary infix operator with the following properties:

A. Let $F^g$ denote the set of all geometric transforms. $\circ : F^g \times F^g \to F^g$

B. $\circ$ is commutative and associative

C. $\circ$ distributes over addition

D. Let $a,b,x,y$ be any real numbers. Then $az^x \circ bz^y = abz^{max(x,y)}$

Properties A-C of the join product are identical to multiplication. The only difference is in property D where the maximum of exponents is used instead of the sum. This simple join product operation is used to help compute the execution time transform for fork-join statements according to the following rules:

5. If S is a fork-list statement of the form $FORK\ S_1;S_2;...;S_n\ JOIN$, then $T(S) = T(S_1)\circ T(S_2)\circ \cdots \circ T(S_n)$
6. If S is a fork-n statement of the form $FORK(n)$: $S_1$; $JOIN$, then $T(S) = \underbrace{T(S_1)\circ T(S_1)\circ \cdots \circ T(S_1)}_{\text{n repetitions}}$

For a more complete presentation of all of these rules including proofs of their validity, the reader is referred to Lester [18]. Applying rules 1-4 to the program of figure 1 gives the following transforms for the execution time of each of the three statements inside the fork-join:

$$.5z^{20}+.5z^{35}$$
$$(z^3)^{10}=z^{30}$$
$$.4z^5+.6z^{10}$$

Performing a join product on these three transforms according to rule 5 gives the transform for the whole FORK-JOIN as follows:

$$\begin{aligned}&(.5z^{20}+.5z^{35})\circ z^{30}\circ(.4z^5+.6z^{10})\\ &= (.5z^{20}\circ z^{30}+.5z^{35}\circ z^{30})\circ(.4z^5+.6z^{10})\\ &= (.5z^{30}+.5z^{35})\circ(.4z^5+.6z^{10})\\ &= .5z^{30}\circ .4z^5+.5z^{30}\circ .6z^{10}+.5z^{35}\circ .4z^5+.5z^{35}\circ .6z^{10}\\ &= .2z^{30}+.3z^{30}+.2z^{35}+.3z^{35} = .5z^{30}+.5z^{35}\end{aligned}$$

For the outer IF of the program, the true branch has transform $z^5z^2z^7 = z^{14}$ according to rules 1 and 2. Combining this with the FORK transform from the false branch using rule 3 gives the following:

$$.3z^{14}+.7(.5z^{30}+.5z^{35}) = .3z^{14}+.35z^{30}+.35z^{35}$$

This is the transform for the execution time of the whole program. The expected (average) execution time of the program can be computed directly from the transform to be 26.95 time units.

The transform computation rule for *fork-gen* statements is more complex and is only summarized here. The reader is referred to Lester [18] for a more complete detailed presentation. The transform for a *fork-gen* is computed recursively from the transform for each of the component statements $S_i$ having the following form:

$$T(S_i) = \sum_{t=0}^{\infty} f_i(t)z^t$$

In a transform, the coefficient of each power of z is a real number representing a probability. For the purposes of computing the overall transform of a *fork-gen* statement, these coefficients are represented symbolically as a character string called an *f-symbol* of the form $f_i(t)$, where i and t are positive integers. This symbolic form of the execution time transform is called the ***transmission***. For example, the transmission for statement 2 in the *fork-gen* of figure 2 is $f_2(10)z^{10} + f_2(23)z^{23}$. There is a special multiplication rule for these *f-symbols* as follows:

Let $f_i(t_a)$ and $f_i(t_b)$ be any *f-symbols*, then

$$f_i(t_a)\, f_i(t_b) = \begin{cases} f_i(t_a) & \text{if } t_a = t_b \\ 0 & \textit{otherwise} \end{cases}$$

Using the execution precedence graph for the *fork-gen* statement, the ***flow*** is computed for each arc in the graph according to the following rules:

1. The flow on the input arc to the *fork-gen* is 1.
2. The flow on each output arc of a statement is the join product of all its input flows multiplied by the statement transmission.

The execution time transform for the whole *fork-gen* is found by computing the flow on the output arc of the execution precedence graph and then substituting the corresponding numerical values for the *f-symbols*. For example, the above flow computation rules can be used on the *fork-gen* of figure 3 to compute the following flow for the output arc:

$$f_2(10)\, f_5(5)z^{23} + f_2(10)\, f_5(15)z^{25} +$$
$$f_2(23)\, f_5(5)z^{31} + f_2(23)\, f_5(15)z^{38}$$

Substituting the specific values of the f-symbols gives the transform for the whole fork-join:

$$.12z^{23} + .28z^{25} + .18z^{31} + .42z^{38}$$

The expected execution time for the whole *fork-gen* can easily be computed from this transform as 31.3 time units.

## 4. PARALLEL SPEEDUP FACTOR

For large programs, the computation of the complete transform may be very time consuming since the number of terms in the transforms may grow quite large, especially if there are lots of complex loops with high repetitions. For such programs it is necessary to have an efficient approximation technique for estimating bounds on the expected execution time. Since the branching probabilities, delay times, and loop repetition counts are usually only approximations themselves, a good approximation technique for analyzing the whole program may be sufficient in many cases. The parallel speedup factor may be computed exactly using transforms or approximated efficiently with one of these approximation techniques.

The recursive definition of PEL statements allows a simple recursive computation rule for the lower and upper bounds on the expected execution time: the bounds for each statement are computed from the bounds of its component statements. The lower bound will be calculated by using expectations or averages, and the upper bound by using maximums. For most programs, the lower bound will be much tighter than the upper — in fact, for purely sequential programs with no parallelism, the lower bound will be exactly the expected execution time. However, we shall see later that for certain programs with a high degree of concurrency, the actual expected execution time may be nearer to the upper bound. Similar approximation techniques were introduced for performance evaluation of asynchronous systems in Lester [19].

*Definition*: Let $S$ denote any PEL statement. The ***average approximation*** (denoted $AVE(S)$) is a nonnegative number defined recursively as follows:

(a) If S is a delay statement of the form $DELAY(t)$, then $AVE(S) = t$.

(b) If S is a sequence statement of the form *BEGIN* $S_1;S_2;...;S_n$ *END*, then $AVE(S) = AVE(S_1) + AVE(S_2) + \cdots + AVE(S_n)$.

(c) If S is a loop statement of the form *LOOP* $n$ *DO* $S_1$, then $AVE(S) = n\ AVE(S_1)$.

(d) If S is an if statement of the form *IF RANDOM*$<p$ *THEN* $S_T$ *ELSE* $S_F$, then $AVE(S) = p\ AVE(S_T) + (1-p)\ AVE(S_F)$.

(e) If S is a fork-join statement with component statements $S_1, S_2, ..., S_n$, then $AVE(S)$ is length of the longest path through $S$ (critical path), where path length is the sum of $AVE(S_i)$ for all statements $S_i$ encountered on the path.

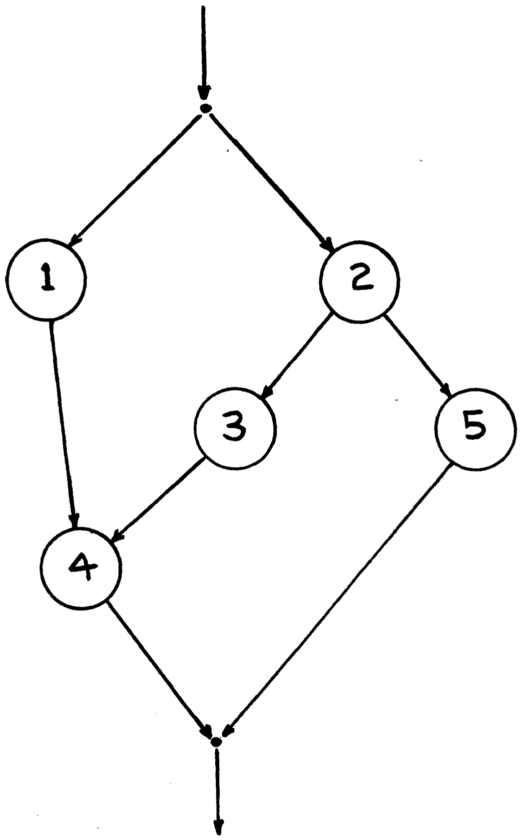


Figure 3 - Execution Precedence Graph

For any PEL statement S, the *maximum approximation* (denoted $MAX(S)$) is defined according to the same rules as $AVE$, except for rule (d) where the maximum of the true and false branches is used instead of the average. It is possible to show that for any PEL statement $S$, $AVE(S)$ and $MAX(S)$ represent lower and upper bounds on the expected execution time $E(S)$:

$$AVE(S) \leq E(S) \leq MAX(S)$$

In fact, $MAX(S)$ is also an upper bound on the execution time itself, not just the expectation. The definition of $AVE$ and $MAX$ provides a linear time recursive algorithm for computing an upper and lower bound on the expected execution time of any PEL program. Recall for the program of figures 1, the exact expected execution time was calculated earlier as 26.95 time units. For this program, $AVE$ is 25.2 and $MAX$ is 35. The reason that $AVE$ is so close to the actual expected time is that the program has a relatively low level of parallelism. For the ***fork-gen*** statement of figure 2, the exact flow methods and ***f-symbol*** technique were used to calculate the expected execution time as 31.3 time units. Using the approximation techniques, $AVE$ is computed as 29.8, which compares favorably to the exact expected time, and $MAX$ is 38.

For any PEL statement S, the *sequential execution time* (denoted $SEQ(S)$) is defined according to the same rules as $AVE$, except for rule (e) where the sum of the times for all the component statements of each fork-join is used instead of the critical path. It is easily shown that $SEQ$ corresponds exactly to the expected execution time assuming that fork-join statements are executed sequentially rather than concurrently. So far in our computation of the execution time, we have assumed an unlimited number of processors so that all the concurrency represented in the fork-join statements may be exploited. $SEQ$ is the expected execution time assuming the program is executed on a one processor system with no concurrency. Just as for the $AVE$ and $MAX$ approximations, $SEQ$ can also be calculated in time which is linear in the number of statements in the program.

For any PEL program, the ***parallel speedup factor*** is defined as the expected sequential execution time divided by the expected execution time. That is, the parallel speedup factor is the expected execution time assuming one processor divided by the expected time assuming an unlimited number of processors. The expected parallel execution time can be calculated exactly by using the transform method, or it can be approximated using the $AVE$ or $MAX$ approximations. The parallel speedup factor can then easily be calculated by dividing into $SEQ$. For the example program of figure 1, $SEQ$ is 50.05 and the parallel speedup factor is therefore (1.86). For the ***fork-gen*** of figure 3, the sequential execution time ($SEQ$) is 57.8 and the parallel speedup factor is (1.85).

## 5. NORMAL APPROXIMATION TECHNIQUE

The methods of section 4 provide a linear time algorithm to compute upper and lower bounds on the expected execution time of any PEL program. For programs with a relatively low level of parallelism, the lower bound ($AVE$) is a good approximation to the actual expected execution time; and for purely sequential programs, $AVE$ is the exact value of the expected time. However, programs with moderate or high parallelism may have expected execution times that differ significantly from $AVE$, and in some cases even approaches the $MAX$ approximation. This section introduces an additional approximation technique called the "normal approximation," which uses the degree of parallelism to compute a more exact approximation to the expected execution time somewhere between the upper and lower bounds provided by $MAX$ and $AVE$.

The normal approximation improves significantly on the $AVE$ approximation by considering both the expectation and the variance of the execution time for each statement. There is a simple recursive computation rule for the normal approximation that is similar to the $AVE$ approximation, except that two numbers are computed for each statement: the $NORM$ (an approximation to the expectation) and the $VAR$ (an approximation to the variance). It is interesting that for purely sequential programs, the variance does not effect the expected execution time, but for highly parallel programs, changes in the variance may significantly effect the overall expectation. Thus, by considering the variance, the normal approximation is much more accurate than the AVE or MAX approximations, and yet is still efficiently computable.

For delay, sequence, loop, and if statements in PEL, standard probabilistic techniques provide simple formulas for computing the expectation ($NORM$) and the variance ($VAR$) from the component statements. For the fork-join statements, a new technique has been developed which assumes a normally distributed execution time for each component statement and then uses the exact transform techniques outlined in section 3 to compute the overall expectation and variance. To reduce the complexity of the transform computation, a discrete approximation to a normal distribution is used for each component statement. This discrete approximation has only seven nonzero terms computed directly from the expectation and variance of the component statement, thus insuring that the transform techniques are fast and efficient.

*Definition*: Let S be any PEL statement. The ***normal approximation*** is a pair of numbers $NORM(S)$ and $VAR(S)$ defined recursively by the following rules:

(a) If S is a delay statement of the form $DELAY(t)$, then $NORM(S) = t$ and $VAR(S) = 0$.

(b) If S is a sequence statement of the form $BEGIN\ S_1;S_2;...;S_n\ END$, then $NORM(S)=NORM(S_1)+NORM(S_2)+ \cdots +NORM(S_n)$ and $VAR(S) = VAR(S_1)+VAR(S_2)+ \cdots + VAR(S_n)$.

(c) If S is a loop statement of the form $LOOP\ n\ DO\ S_1$, then $NORM(S) = n\ NORM(S_1)$ and $VAR(S) = n\ VAR(S_1)$.

(d) If S is an if statement of the form $IF\ RANDOM{<}p\ THEN\ S_T\ ELSE\ S_F$, then $NORM(S) = p\ NORM(S_T) + (1-p)\ NORM(S_F)$ and $VAR(S) = p(NORM(S_T)^2 + VAR(S_T)) + (1-p)(NORM(S_F)^2 + VAR(S_F)) - NORM(S)^2$

(e) If S is a fork-join statement, then for each component statement $S_i$, let $N$ denote $NORM(S_i)$ and $D$ denote $(VAR(S_i))^{1/2}$. Create the following discrete normal approximation to the transform of $S_i$:
$T(S_i)=.00621z^{N-3D}+.0606z^{N-2D}+.2417z^{N-D}+ .3829z^{N}+.2417z^{N+D}+.0606z^{N+2D}+.00621z^{N+3D}$
(where the exponents of z in $T(S_i)$ are rounded to the nearest whole number). The usual transform techniques

of section 3 are applied to these approximate transforms to compute an approximate transform $T(S)$ for the whole fork-join. $NORM(S)$ is defined as the expectation for $T(S)$, and $VAR(S)$ is defined as the variance for $T(S)$.

The accuracy of this normal approximation technique depends on how closely the actual execution time distributions of the individual statements follow a normal distribution. There is an important result in standard probability theory called the *central limit theorem*, which states that the probability distribution for a sum of n independent random variables approaches a normal distribution as n grows large. Thus, if the individual statements in the fork-join contain a loop or have many component statements, then chances are good that their execution time will indeed begin to approach a normal distribution. We have found that loop statements with anything in excess of a few repetitions, can be approximated by normal distributions with an accuracy of 1-2% . Our empirical studies of a wide range of parallel programs have shown that the normal approximation is accurate within 1-5%, representing a significant improvement over the $AVE$ and $MAX$ approximations, and yet still efficiently computable even for large programs.

## 6. IMPLEMENTATION

For research purposes, a prototype system has been developed for analyzing PEL programs using the exact transform techniques and the various approximation techniques presented in this paper. The system is written in LISP and is currently running on a VAX 11/780 under the VMS operating system. The PEL program is entered into the system as a simple text file in a format similar to that shown in the figures of this paper. To illustrate the practical use of this system in designing and analyzing real parallel programs, we give two simple examples here: highly parallel vector division and parallel file processing with double buffering. For both these examples, the process of formulating the algorithm and creating the PEL program are illustrated, then the actual results from the prototype PEL analysis system are given.

The first example is a simple vector division function with a special rule for handling division by zero: if the divisor and dividend are both zero, then the quotient is one, but if the dividend is nonzero, then the quotient is positive or negative infinity (represented by the maximum size number for the computer). Using the standard "forall" primitive to represent the parallel form of a loop, the function to compute a vector C by dividing the sixteen component vector A by vector B is as follows:

```
FORALL i := 1 TO 16 PARDO
  IF B[i]<>0 THEN C[i] := A[i]/B[i]
    ELSE IF A[i] := 0 THEN C[i] := 1
                  ELSE C[i] := MAXNUM * SIGN(A[i])
```

In the above statement, there are sixteen concurrent activations of the body of the "forall", one for each value of the index i. "MAXNUM" is a predefined constant, and "SIGN" is a predefined system function. To translate this statement into PEL, some estimates are needed for the relative execution time of the primitive operations in the language. Let us assume that assignment requires 5 time units, array subscripting and the forall setup each require 10 time units, the SIGN function requires 40 time units, and division, multiplication, and boolean comparison each require 10 time units each. Also, assume that 80% of the vector entries are nonzero in a typical input data set. The "forall" primitive can be represented in PEL by a fork-n statement, and the above statement is translated into the following PEL program:

```
PROGRAM;
  BEGIN
    DELAY(10);
    FORK(16):
      DELAY(25);
      IF RANDOM > .8 THEN DELAY(50)
                ELSE BEGIN
                  DELAY(25);
                  IF RANDOM > .2 THEN DELAY(15)
                            ELSE DELAY(80)
  END.
```

For this program, the PEL analysis system computed the following results:

Expected Execution Time: 136.6
Sequential Execution Time: 1494.4
Parallel Speedup Factor: 10.93

In a second example, we consider a file processing program which reads a sequence of student records from a disk file, computes the grade point average of each student, and stores the results as a new sequential disk file. The program uses double buffering of records for both files so that all three activities of reading, computing, and writing can occurr concurrently. The overall flow of activity is shown in figure 4. It seems from this diagram that the parallel speedup factor will be a maximum of three since there are three concurrent activities. However, a more detailed analysis reveals that the parallel speedup factor is highly dependent on file storage techniques, disk access times, and execution times for language primitives.

Let us assume that the Student File is sequential and contains three student records per disk block, so that a disk access to read in a new block is required with probability (.25) for each student record on the average. Also, assume that the G.P.A. file is sequential with a much smaller record size for which 19 records can be stored in each disk block. Thus, the probability that writing a single G.P.A. record will require a disk access is (.05) on the average. When the program issues an operating system call to read the next record for a sequential disk file, let us assume that the execution time is 2000 time units if a disk access is required, but only 500 time units if the needed block is already in memory and no new disk access is required. Thus the PEL statement for reading each student record is as follows:

```
IF RANDOM < .25 THEN DELAY(2000) ELSE DELAY(500)
```

The PEL statement for writing each G.P.A. record is as follows:

```
IF RANDOM < .05 THEN DELAY(2000) ELSE DELAY(500)
```

The execution time for computing the G.P.A. of each student will depend on some timing assumptions for the primitives of the source programming language and the size of the student record. Assuming that each student record contains an average of 40 course grades, and the primitive arithmetic and other operations of the language vary from 8 to 12 time units, a simple looping program for the "Compute G.P.A." activity shown in figure 4 translates into the following PEL statement:

```
BEGIN
  DELAY(8);
  LOOP 40  DO BEGIN  DELAY(12); DELAY(28)  END;
  DELAY(8);
  DELAY(8)
END;
```

To execute the Read, Compute, and Write activities concurrently, a fork-list statement is needed in PEL, and this statement will be executed once for each student record. Assuming there are 1000 student records in the file, there will be an outer loop with 1000 repetitions. Finally, some timing assumptions are needed regarding file and buffer initialization times: opening a sequential file for input requires 2500 time units, opening a new sequential file for output requires 600 time units, and initializing buffers requires 40 time units. From all of the above assumptions, the following PEL program results:

```
PROGRAM;
 BEGIN
   DELAY(2500); DELAY(600); DELAY(40); DELAY(40);
     LOOP 1000 DO
       BEGIN
         DELAY(12);
         DELAY(12);
         FORK  DELAY(8); DELAY(8)  JOIN;
         DELAY(12);

         FORK
           IF RANDOM < .25 THEN DELAY(2000)
                          ELSE DELAY(500);

           BEGIN
             DELAY(8);
             LOOP 40 DO
               BEGIN
                 DELAY(12);
                 DELAY(28);
               END;
             DELAY(8);
             DELAY(8)
           END;

           IF RANDOM < .05 THEN DELAY(2000)
                       ELSE DELAY(500);
         JOIN
       END;
 END.
```

Using the prototype PEL analysis system for this program results in the following output:

```
Expected Execution Time:  17.79 × 10^5
Sequential Execution Time: 31.18 × 10^5
Parallel Speedup Factor: 1.75
```

Thus, it is seen that waiting time and lack of balance between the execution time of the three concurrent activities in figure 4 has reduced the apparent concurrency factor of 3 to an actual parallel speedup factor of only (1.75). If the time unit used in making the initial estimates for the basic operations is 10 microseconds, then the expected execution time for the program is 17.8 seconds.

## 7. CONCLUSIONS

The PEL language is a formal system for describing the overall control structure and timing delays in a parallel program, as an aid in computing the parallel speedup factor. Real parallel programs are sufficiently large and complex that such a formal description language is useful as a supplement to standard algorithm analysis techniques. The transform techniques and efficient approximation techniques presented in this paper for PEL programs can then be used to compute the expected execution time and parallel speedup factor for the program. The mathematical complexities introduced by program parallelism make formal analysis techniques necessary even for small programs.

There is a large body of standard analysis techniques for sequential probabilistic systems using z-transforms and linear flow graph reduction. The major contribution of this paper is the extension of these techniques to parallel systems in the form of PEL programs. This paper also presents efficient approximation techniques for estimating the expected execution time and parallel speedup factor for large programs. Future work in this area will focus on the development of compilers to translate parallel programs directly into PEL for analysis. Also, future research is needed to extend the PEL primitives to allow message passing between parallel processes, a feature which is contained in many concurrent programming languages.

## REFERENCES

[1] C.V. Ramamoorthy, "Discrete Markov Analysis of Computer Programs," *Proc. ACM 20th National Conference*, 1965, pp. 386-392.

[2] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, N.J., 1974, pp. 444-448.

[3] R.A. Howard, *Dynamic Probabilistic Systems, Vol. 1: Markov Models*, Wiley, New York, 1971.

[4] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[5] W.M. Zuberek, "Timed Petri Nets and Preliminary Performance Evaluation," *Proc. IEEE 7th Annual Symposium on Computer Architecture*, 1980, pp. 89-96.

[6] M.K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers*, Vol. C-31, No. 9, September 1982, pp. 913-917.

[7] M.A. Marsan, G. Conte, and G. Balboa "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Transactions on Computer Systems*, Vol. 2, No. 2, May 1984, pp. 93-122.

[8] R.A. Sahner and K.S. Trivedi, "SPADE: A Tool for Performance and Reliability Evaluation," International Conference on Modelling Techniques and Tools for Performance Analysis, France, June 1985.

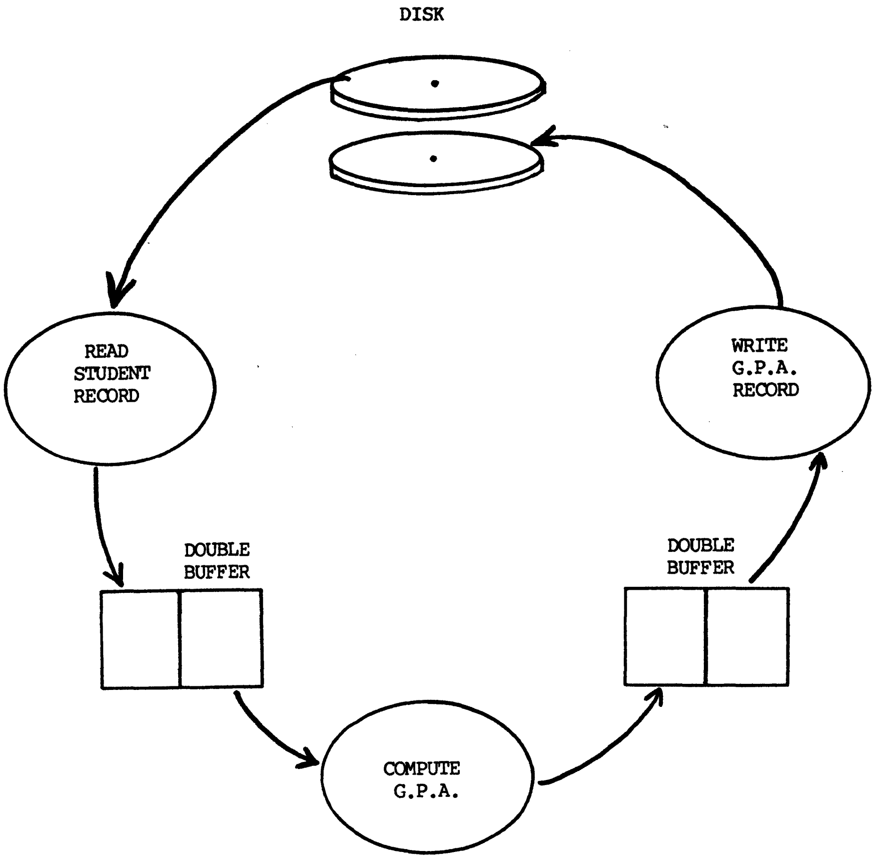


Figure 4 - Parallel File Processing Program

[9] D.J. Kuck, et.al., "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," *Proc. 1984 International Conference on Parallel Processing*, IEEE Press, August 1984, pp. 129-138.

[10] J.B. Dennis and E. VanHorn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, Vol. 9, No. 3, March 1966, pp. 143-155.

[11] A.C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974, pp. 55-58.

[12] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transcations on Software Engineering*, Vol. 1, No.2, June 1975, pp. 199-207.

[13] *OCCAM Reference Manual*, INMOS Corporation, Colorado Springs, Colorado.

[14] J. McGraw, et. al., "SISAL: Streams and Iteration in a Single Assignment Language," Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory, Livermore, California, March 1985.

[15] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions of Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 381-404.

[16] R.H. Perrot, "A Language for Array and Vector Processors," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 2, October 1979, pp. 177-195.

[17] D.J. Kuck, *The Structure of Computers and Computations*, Wiley, New York, 1978, pp. 135-155.

[18] B.P. Lester, "Performance Evaluation of Structured Concurrent Programs," submitted for publication, 1985.

[19] B.P. Lester, "Performance Evaluation of Asynchronous, Modular Systems," *Computers and Electrical Engineering*, Vol. 8, No. 3, September 1981, pp. 207-222.

# PARALLEL STRUCTURING OF CONTROL AND RESOURCES MANAGEMENT SYSTEMS FOR PARALLEL PROGRAMS

Robert O'Dell[1] and J. C. Browne
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

## Introduction

Execution of parallel programs on parallel computer architectures requires scheduling, coordination, and resource management. These control activities themselves consume computer resources. Centralized resource serial control will become the performance bottleneck when large numbers of processes must be scheduled and managed.

This paper studies parallel structuring of process control and resource management. The environment for the study is the Run-time System of the Computation Structures Language (CSL) parallel programming system. The approach has been to design and evaluate a parallel version of the control and resource management functions of the CSL run-time system. The performance characteristics of the parallel version are predicted and an algorithm for determining the optimal degree of parallelism for the run-time system is given.

Parallel structuring of resource management is essential in order to alleviate the well-known serialization bottleneck that occurs whenever a large number of parallel processes must access a shared resource through a serial monitor [MAD74, CEZ83].

The serialization bottleneck of a centralized resource management system can be demonstrated by considering the fact that the resource manager is a shared resource of the tasks. The resource manager can impose a constant upper bound on the speedup obtainable in the system even if the tasks themselves run completely in parallel without interdependencies and have a linear speedup potential.

The Computation Structures Language (CSL) is a language for specifying the structure of parallel computations and controlling the execution of these structures. (The next section gives an introduction to and examples of CSL programs.) The CSL Run-time System (RTS) thus forms a serialization bottleneck when the execution of a complex computation structure must be managed, and limits the maximum speedup obtainable in parallel programs in the CSL system. The design of a Parallel Run-time System (PARTS) consists of a method of parallelizing the resource management functions of the original Run-time System. It consists of a restructuring of the system tables such that they support multiple copies of the original RTS along with specifications for these access procedures.

A simulation model for PARTS executing on a certain class of CSL programs was developed. The data used to parameterize the modes is taken from a working RTS on a real system where appropriate. The performance of the model under varying conditions is shown.

An analytical equation that closely approximates the performance of the simulation is presented. The equation is used to predict the performance characteristics of PARTS under variations of degree of parallelism and to develop a schedule for creating multiple copies of the RTS to execute in parallel that maximizes the speedup under the condition that all tasks have equal execution times. The restriction to tasks with equal execution times is not as limiting as it might seem at first glance. We have found that execution of CSL programs is often dominated by execution of replicated arrays of processes which will have closely similar execution times.

The results reported here should have at least conceptual application to any system for control and source management of complex dynamic parallel computation structures.

## Parallelizing the Centralized RTS

The Computation Structures Language (CSL) is a language for expressing the synchronization and communication requirements of a set of tasks [Browne 82]. CSL specifies the traversal of a computation graph. The CSL program does not, however, perform any of the computations associated with the parallel program. A simple CSL program for the producer/consumer problem is shown below.

```
JOB Sample-CSL;

CONSTRUCT
    TASKS
        T[i] : File1;  RANGE i=1 to 2;
        T3   : File2;

    CHANNELS
        Ch[i] : DATACHANNEL from T[i] to T3;
                          RANGE i=1 to 2;

END; (* Construct *)

BEGIN   (* Executable Code *)

  COBEGIN
      // EXECUTE T[1];
         SEND X to Ch[1];
         RECEIVE Y from Ch[1];

      // EXECUTE T[2];
         SEND Y to Ch[2];
         RECEIVE Y from Ch[2];
   COEND;
   EXECUTE T3;

END. (* CSL Program *)
```

[1] The first author is now employed by Advanced Micro Devices, 5900 East Ben White Blvd., Austin, Texas 78741.

In this program it is specified that T[1] and T[2] may execute in parallel. The three statements following each "//" comprise a parallel stream. Each statement within a stream is executed serially. After T3 receives both X and Y, the Cobegin completes and T3 may execute.

The CSL RTS is the resource manager for a running parallel program. It can be thought of as defining a local operating system for the parallel program. When a Cobegin is encountered, the RTS sends messages to tasks telling them to begin (Initializing), waits for completion, and determines status information through a message sent back from the completed task (Finishing). The RTS also coordinates the passing of data between tasks. The result of any action by the RTS is a modification of the global state of the parallel computation. The global state of the computation is recorded in a set of system tables maintained by the run-time system.

By definition of CSL, tasks only communicate with each other before or after their execution. Thus the computation granularity of the tasks depends on the frequency of communication among the tasks. In programs written in the CSL environment, this has varied from 2 MS to 1000 MS.

The set of statements managed within the Cobegin at a given time are serviced in a Round-robin fashion with time being allocated on the basis of what work needs to be done. Thus when a statement is ready to be Initialized or Finished, it may have to wait for service. This causes a serialization bottleneck to develop within the system.

The PARTS system partitions the Cobegin into groups of statements, each managed by a separate RTS. The partitioning process attempts to balance the workload between RTS's. When a Cobegin is encountered, the current RTS "Initializes" the other RTS's telling each to commence execution. This is an overhead factor in the PARTS system. As soon as the RTS's are Initialized, they begin managing their portion of the Cobegin statements. Since a global state is preserved, the system tables are made available for access by all RTS's in the system. Since RTS's modify the global state, the system tables must be accessed serially which may introduce wait time into RTS execution.

In summary, the serialization bottleneck of Initializing and Finishing statements in the Cobegin has been parallelized in PARTS, but has been compromised somewhat by the time to Initialize and Finish the new RTS's plus the additional wait time due to shared memory contention for system table access. However, the first-order serialization point has been removed.

## Simulation Model

The simulation model developed for PARTS is an extension of a model of a single RTS managing a given number of streams. Since the Cobegin is the major feature of the language, and the schema by which parallel computation may take place in CSL, the simulation models just this activity. (Often an entire parallel program structure can be described by a series of Cobegin statements.)

The simulation was constructed using the Performance Analyst's Workbench System [IRA84], an event driven simulation language especially designed for modelling computer and information systems. The operation of PAWS is described here through Information Processing Graphs (IPG's).

Figure 1 shows the IPG of a single RTS managing 3 computation streams comprised of a single statement. This, for instance, would model the following:

```
COBEGIN
      // EXECUTE T1;
      // EXECUTE T2;
      // EXECUTE T3;
COEND;
```

The result of a Cobegin being encountered is a transaction leaving Source and proceeding to Fork. There, 3 new transactions are generated with phases 1, 2, and 3. After waiting in series in the Queue IANDF for Initialization, the transactions cause the operation of the stream to commence. E1 through E3 model the execution times of each of the three streams. When a stream completes, a transaction passes through the Compute node for routing purposes, enters the IANDF Queue to be Finished, and then waits at Join for the rest of the streams to complete and be Finished. When all 3 transactions enter Join, the simulation completes. For large numbers of streams IANDF will be nonempty for the duration of the simulation.

An implicit assumption made within the model is that the streams execute independently. This was done deliberately to expose the severe performance degradation experienced by parallel computations without sufficient parallel resource management, even when the underlying computation is experiencing optimal performance. Any communication between streams would increase the workload of the RTS's, increasing the serialization bottleneck.

Figure 2 shows the extension of the model to a PARTS configuration with 3 RTS's each managing their portion of a partitioning of the streams in a Cobegin. In this model, RTS1 Initializes RTS2 and RTS3 before proceeding on to manage its streams. Since RTS2 and RTS3 are tasks themselves, they must pass through the IANDF node of RTS1 to be Finished. When all three RTS's have completed, the Cobegin is complete.

An allowance for the effect of shared memory contention between RTS's is made by introducing a delay node representing the time taken by each RTS to acquire control of the shared tables. In PARTS this is nearly constant for any given stream.

The parameters of the model, then, are:

1. X - the number of streams in the Cobegin
2. Y - the number of RTS's defined to execute in parallel
3. I - the task Initializing time
4. F - the task Finishing time
5. RTSi - the RTS Initializing time
6. RTSf - the RTS Finishing time
7. E - the execution time of a given stream (task)
8. SM - the shared memory access time

## An Analytical Model

An analytical model is shown that describes the behavior of the PARTS system when all streams have equal execution times. CSL allows for convenient methods of executing tasks in parallel that are replicas of each other. This feature has been used extensively in CSL programs written to date.

Speedup in PARTS is defined as:

$$(I + E + F) \ / \ f(X, Y, SM)$$

where "I + E + F" is the time for a single RTS to execute 1 stream of execution time E. The function "f" is the completion time of a PARTS configuration (for a given Cobegin) normalized to that of a single stream.

$$\begin{aligned} & f(X, Y, SM) = (A + B + C + D) \ / \ X \\ \text{where:} & \\ A &= (X/Y) \ (I + F) \\ B &= (Y-1) \ (RTSi + RTSf) \\ C &= 2 \ (X/Y) \ SM \ (Y-1) \ p \\ & p = MIN \ [((y-1)SM) \ / \ (SM + I) \ , 1 \ ] \\ D &= MAX \ [ \ 0 \ , \ E - \ ((X/Y) - 1)F \ ] \end{aligned}$$

"A" represents the savings achieved by parallelizing the resource management functions of Initializing and Finishing tasks. "B" is derived from the fact that (Y-1) RTS's must be Initialized and Finished by the master RTS. "C" models the added time due to the serialization of shared memory access. When p=1, the queue for shared memory access is saturated. D models the idle wait time an RTS spends waiting for work before the first task completes, but after all streams have been initialized. If an RTS is kept completely busy, then D=0.

## Results

Simulation runs were made for many values of the parameters. Space precludes detailed presentation of results. Additional results may be obtained from the authors by request.

Figure 3 shows the speedups for tasks with a typical execution length (25 MS) observed in the CSL environment. The values for I, F, RTSi, and RTSf were determined to be 4 +- 1.25 MS on a Dual Cyber 170/750 system. This time was comprised mainly of Cyber operating system overhead in the message passing system. The value of SM was given the pessimistic value of 0.5 MS.

If the right choice is made for the number of RTS's to be used in a given configuration, PARTS is an effective way to increase the speedup potential. This scheduling choice can be easily accomplished by differentiating the speedup equations with respect to Y. This gives the schedule of RTS's for maximum speedup as:

$$Y = SQRT \ ( \ X \ \ (I - 2 \ SM \ p)/ \ (RTSi + RTSf))$$

With this choice of schedule, the speedup can be shown to be proportional to SQRT(X). This holds even though the performance curve of any given RTS reaches an upperbound. If the execution time of each task is the same (as we have assumed above), the maximum speedup is independent of the execution time. Of course, the speedup using that schedule may vary widely

An arena in which PARTS turns out not to be so promising is when increasing numbers of tasks compute over a constant workload (decreasing granularity of units of computation). (In the previous examples, increasing X implied increasing the problem size. Each task continued to have the same execution time regardless of the number of tasks.) In this case, we define the sum of the execution times of all tasks to be constant.

A nonintuitive result is shown in Figure 4. In this case the simulation of Figure 5 is modified by allowing the execution times of the tasks to vary in a hyperexponential distribution around a mean of 25 MS. An increase in the variance-to-mean ratio to 0.4 caused the results shown. For a certain range of X, the speedup is actually better for a larger variance!

This is explained in the following way. The initial performance decrease is due to the fact that the increased variance causes some streams to finish much later than others. Time is lost while an RTS waits for it to complete. When more streams are introduced, however, the sheer number of streams causes each RTS to have no idle wait time in finishing streams. The higher variance, then means that the RTS has a higher probability of "starting" the finishing work earlier and thus "completing" it earlier. This results in greater parallelism among RTS's and less contention for system tables.

## Conclusion

This paper has presented a study of the effect of parallel structuring of control and resource management algorithms upon the execution of parallel computations. A simple analytic model of general parallel computation incorporating overhead effects was developed and applied to this example. Speed-up shows a square root dependency on the degree of parallelism in the control and resource management algorithms.

Variance in processing time for the tasks being controlled was observed to be beneficial over certain ranges of execution to the parallelism obtainable by the control system.

## Acknowledgements

This work was supported in part by a grant from the Department of Energy, Grant Number DE-AS05-81-ER10987, and by the Control Data Corporation, Subcontract Number 4502-001285.

## REFERENCES

[BRO81] Browne, J.C., Tripathi, et al, "A Language for Definition and Control of Reconfigurable Parallel Computation Structures," *Proc. of Int. Conf. on Parallel Processing*, 1981.

[MAD74] Madnick, S.E. and Donovan, J.J., *Computer Science Series: Operating Systems*, McGraw-Hill, Inc., 1974.

[CEZ83] Cezzar, K. and Klappholz, D., "Process Management in a Speedup-Oriented MIMD System," *Proc. of Int. Conf. on Parallel Processing*, 1983.

[FLA84] Flatt, H.P., "Parallel Processing - Can you eat your cake and have it too?" preprint, IBM Palo Alto Scientific Center.

[IRA84] *Performance Analysts Workbench System (PAWS)*, Information Research Associates, Austin, Texas, 1984.

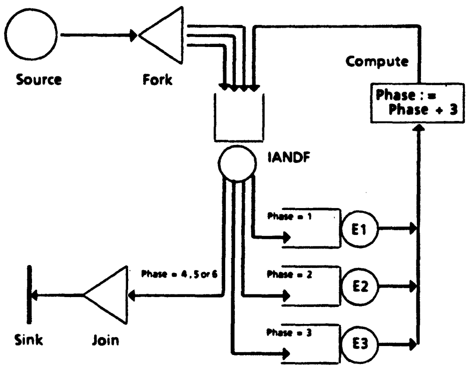


FIGURE 1. A Single RTS Managing 3 Tasks

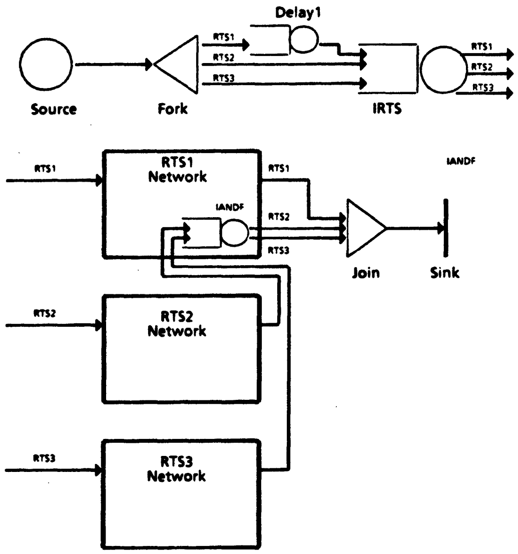


FIGURE 2. A PARTS Execution with 3 RTS's

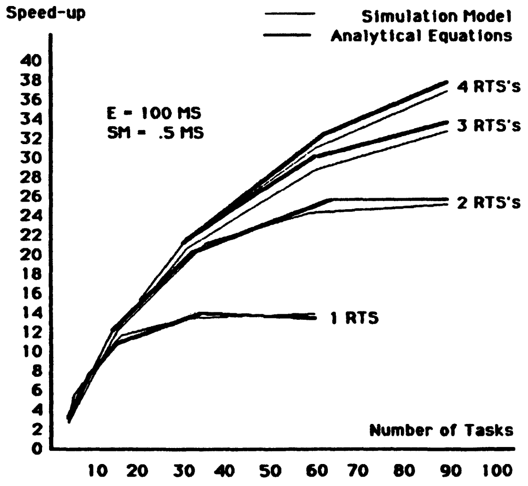


FIGURE 3. Speed-ups for 100 MS Tasks in PARTS

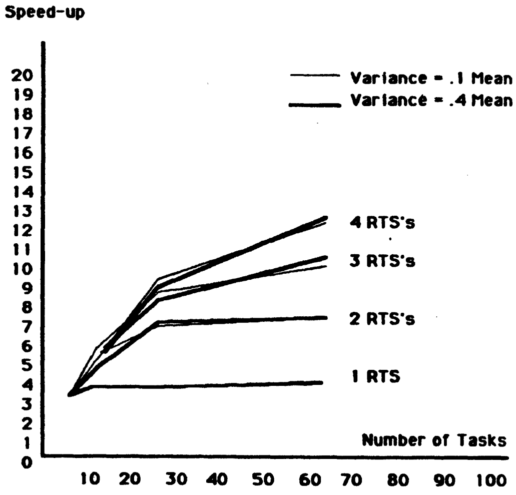


FIGURE 4. Speed-ups for Tasks with varying Execution times

# ESTIMATING THE SPEEDUP IN PARALLEL PARSING *

Dilip Sarkar
Narsingh Deo

Computer Science Department
Washington State University
Pullman, WA 99164-1210

Abstract. A method for estimating the speedup for asynchronous bottom–up parallel parsing has been presented. Two models for bottom–up parallel parsing are proposed, and the speedup for each of the two models is estimated using the technique developed here. The speedup obtained for one model is very close to the simulation result already available in literature. The second model shows a greater speedup than the first.

Index Terms: compilers, parallelism, parsing

## I. INTRODUCTION

The term parallel computer means different things to different people. In this paper it is an MIMD machine – a general–purpose, tightly–coupled, collection of a fixed number of identical processors, working asynchronously under one operating system to solve a single computational problem. Such parallel machines have been in existence for some time. Two primary types of such machines seem to be emerging (i) the fixed–connection model, such as Intel's iPSC family and (ii) the shared–memory models, such as the HEP computer. When a large number of processors are to be connected together, the former has an advantage from the hardware point of view, but the latter is more convenient to construct an algorithm on.

It is too early to tell which of the two organizations is going to be the dominant commercial machine. For a good introduction to various architectural issues see [11, 13]. For this paper either of two models can be assumed.

Although a good deal of work has been reported on parallel algorithms in various application areas, the amount of work done on compilation in parallel has been relatively meager. From the results of the earlier work it is clear that parallelism in compilation is promising. But the questions of speedup, efficiency, and so forth are yet to be answered satisfactorily. In this paper we consider one such aspect of parallel compilation.

The compilation can be divided, broadly, into three tasks – lexical analysis, parsing, and code generation [1,2]. Lexical analysis and code generation appear more parallelizable than parsing [7,8,12]. Therefore, we deal only with parsing in this paper.

Cohen and Kolodner [5] proposed an asynchronous bottom–up parallel parsing model and estimated, by simulation, the speedup obtainable for Pascal–like languages. If the processors are connected as a linear array (as they have implied), their model may not always parse a string successfully, because a processor goes into inactive mode when its stack becomes empty, and a processor can directly communicate only to its adjacent processors. For example, let $P_1, P_2$, and $P_3$ be three processors, arranged in that order, parsing a string. If the stack of $P_2$ becomes empty before those of $P_1$ and $P_3$ then the parsing cannot be completed, because $P_1$ and $P_3$ can no longer communicate.

We modify the model proposed in [5], to run on a linearly connected array of processors, as well as propose a new model that yields higher speedup. Both these models can be realized on a shared–memory machine as well as on a fixed–connection machine.

In Section II related previous work is surveyed briefly. In Section III two parallel parsing models are proposed. A technique to estimate speedup in asynchronous bottom–up parallel parsing is presented in Section IV, while in Section V we estimate the speedup that can be obtained with the two models proposed in Section III. It is shown that the new model gives a greater speedup than the model presented in [5]. Our result also shows that the conjecture made about the average speedup for bottom–up parallel parsing in [4] is true for one of the models proposed here.

## II. PREVIOUS WORK

For parallel computer architecture, and programming the readers are referred to Hwang and Brigs [11], and Kuck [13]. Ellis [9] presented two algorithms for compilation which can be implemented on a parallel machine with ILLIAC IV–type cellular architecture. Various techniques for lexical analysis and parsing, using the CDC–STAR–100 vector instruction set, were introduced by Donegan and Katzke [8]. Krohn [13] also exploited the vector processing capability of CDC–STAR–100 to generate object code in parallel for Fortran–like languages. The program slicing mechanism discussed in [16] can also be used for parallel compilation.

Baer and Ellis [3] have shown that by modelling an existing sequential compiler we get an understanding of modifications necessary to transform the sequential structure into a pipeline of processes. They have evaluated a pipelined compiler through measurement and simulation. But the pipeline of processes has a limitation that only a fixed and small number of processors can be used, and that the speedup obtainable is not large.

Dekel and Sahni [7] considered the translation of infix arithmetic expressions into their postfix or syntax tree

* This work was supported by the U.S. Army Research Office under grant DAAG29-82-K-0107.

0190-3918/86/0000/0157 $01.00 © 1986 IEEE

form using synchronous parallel processors, sharing common global memory. The speedup obtainable is high. Mickunas and Schell [15] extended the LR parsing technique [1,2 ] for a multiprocessor environment so that many processors can be used to parse a given string starting at different tokens in the string. Fischer [10] has proposed algorithms for bottom–up synchronous parallel parsing and has shown through simulation that appreciable speedups can be achieved with these algorithms. Ligett et al. [14] extended the LR technique to allow arbitrarily many processors to build parse tree simultanteously and measured the performance of the algorithm experimentally.

Cohen and Hickey [4] made the first attempt to compute upper and lower bounds for the speed- up that can be obtained by asynchronous, bottom–up, non–backtracking parsing of strings generated by a context–free grammar. They analytically found upper and lower bounds for speedup in parallel parsing. To develop these bounds they ignored the processor coordination and communication time. A practical asynchronous parallel parsing model to parse Pascal–like languages was proposed in [5]. The simulation result presented in [5] was compared with the bounds in [4]. The comparison showed a wide gap between the two.

## III. MODELS

In this section two models for asynchronous bottom–up parallel parsing are presented. These models are used in later sections to estimate speedup.

As pointed out in Section I the asynchronous parallel parsing model proposed in [5] may not, under certain situations, complete parsing successfully. We modify this model to run on a linearly connected processor array and call it Model A, to distinguish it from a second model, called Model B, presented later in this section.

Model A: Let there be $q$ processors $P_1, P_2, \ldots, P_q$ arranged in a linear array, such that processor $P_i$ is directly connected with processor $P_{i+1}$ and $P_{i-1}$ (see Fig. 1(a)). A processor $P_j$ is called a predecessor of the processor $P_k$ if $j < k$ and processor $P_k$ is referred to as a successor of $P_j$. Thus, processor $P_i$ has $(i-1)$ predecessors and $(q-i)$ successors. Processor $P_1$ has no predecessor and processor $P_q$ has no successor. Every processor $P_i$ has a stack, which is referred as $STK_i$.

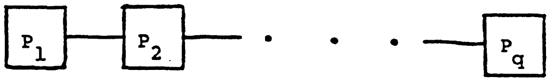


Fig. 1(a) A Linear Array of Processors

The given input string of length $L$ is divided into approximately $q$ equal parts. The ith processor $P_i$ starting at token $\lfloor (i-1)L/q \rfloor$ scans to the right for the next synchronizing token (e.g., semi–colon, end, etc.) and initiates parsing from the next token. A processor can be in one of the four states – active, wait, merge–only, and inactive.

A processor remains in the active state if it is able to perform either of the two parse steps – namely, shift or reduce; or it is performing the stack–merge operation. Stack–merge is the process in which a processor $P_i$ transfers the contents of its stack, from bottom, to the stack of another processor $P_j$ until $P_i$ encounters a stack–separator or its stack becomes empty. (Stack–separator is a special symbol used as a marker to separate the content of the stack of a processor.) When a processor cannot reduce due to insufficient information in its stack, but has received the next synchronizing token, it places a stack separator on the top of the stack and continues parsing. By placing a stack separator a new stack is simulated.

When the end processor $P_q$ has completed parsing its part and is left with a nonempty stack, it enters into merge–only state. When any other processor $P_i, 1 \leq i < q$, has completed parsing its part and is left with nonempty stack, it requests a merge to its successor $P_{i+1}$ and enters into wait state. In the wait state processor $P_i$ may be acknowledged by the processor $P_{i+1}$ or may get merge request from processor $P_{i-1}$. In the former case the state of $P_i$ is changed to active and $P_i$ receives tokens from $P_{i+1}$, while in the latter case processor $P_i$ cancels its merge request to the processor $P_{i+1}$. If a processor $P_i$ is not in wait state and receives a merge request from $P_{i-1}$, then $P_i$ sends acknowledgement to $P_{i-1}$. After sending acknowledgement processor $P_i$ starts stack–merge with $P_{i-1}$. Processor $P_1, 1 < 1 < q$, with nonempty stack goes to merge–only state when its successor processor $P_{i+1}$ is inactive. In merge–only state a processor $P_i$ waits for a merge request from its predecessor $P_{i-1}$. Processor $P_i$ becomes inactive if its stack is empty and $P_{i+1}$ is inactive. A processor $P_i$ in wait state with empty stack does not acknowledge merge request immediately but waits for the contents of the stack of $P_{i+1}$ and transfers them to $P_{i-1}$.

Model B: In this model every processor can communicate directly with every other processor. As expected, the extra cost of interconnection provides an enhancement in parsing speed by reducing the interprocessor coordination and communication time. In such a completely connected system, although there is no predecessors and successors of a processor in strict sense, to identify each processor and its substring we number them as in Model A and use the same terms predecessor and successor. The processors of this model also have four states (as in Model A), but the state transition is different in a few cases, as discussed below.

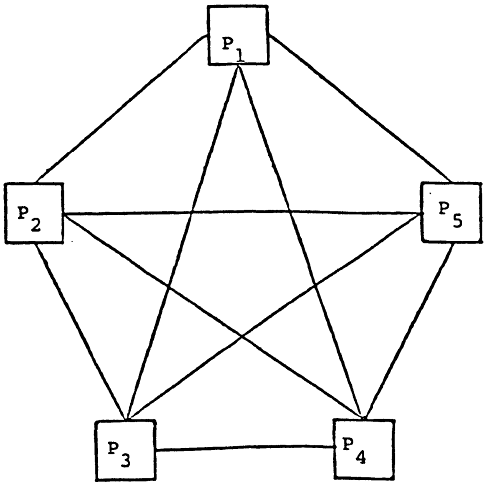


Fig. 1(b) A Completely Connected Five Processor System.

Processor $P_{i+1}$ is called the immediate successor of $P_i$. As soon as the stack of a processor becomes empty, it enters into inactive state irrespective of the state of its immediate successor. A processor $P_i$ knows which processor $P_k$ is its immediate active successor. Before a processor $P_i$ enters into inactive state it informs its immediate active predecessor $P_j$ the index of its immediate active successor $P_k$. For example, let the immediate active successor of $P_3$ be $P_5$ and that of $P_5$ be $P_{10}$. Consider the situation where $P_5$ becomes inactive before $P_3$ and $P_{10}$. In this situation, before $P_5$ becomes inactive it informs $P_3$ that $P_{10}$ is henceforth the immediate active successor of $P_3$.

A method to estimate the speedup for asynchronous bottom–up parallel parsing is presented in the next section.

## IV. ESTIMATING THE SPEEDUP

In this section we develop a technique to estimate speedup for parallel parsing. We divide the parsing time into three parts as in [5].

Let $T_{pq}$ be the total time of parsing a string of tokens in parallel using $q$ processors. In parallel parsing, time is spent for processor coordination and communication, besides the reduction time $T_{rq}$ and shift time $T_{sq}$. Let $T_{cq}$ be the time spent for processor coordination and communication. (The second suffix $q$ is the number of processors used.) We assume that the parse tree has a level $h$ such that for levels 1 to $h$ the number of nodes at each level is smaller than the number of processors, and all processors are not being utilized. The nodes at levels $(h+1)$ and below are sufficient so that all $q$ processors work simultaneously and independently with negligible coordination and communication. From level $h$ to level 1, processor coordination and communication time is significant. Therefore, we consider coordination and communication time, $T_{pq}$ for this part only.

The total time to parse in parallel, $T_{pq}$ can be expressed as

$$T_{pq} = T_{sq} + T_{rq} + T_{cq}$$

Since the coordination and communication time is zero in case of a single processor, the parse time $T_{P1}$ with a single processor is

$$T_{p1} = T_{r1} + T_{s1}$$

Therefore, the speedup obtained with $q$ processors is

$$SP(q) = \frac{T_{r1} + T_{s1}}{T_{rq} + T_{sq} + T_{cq}} \qquad (1)$$

Now, let $L$ be the length of the input, the estimated number of nodes in its parse tree be $N$ and the number of internal nodes in levels 1 to $h$ be $N_c$. The number of shift operations is the length $L$ of the string, which is also the number of leaves in the parse tree. The number of reduce operations is the number of internal nodes in the parse tree. If $t_r$ and $t_s$ be the average reduce and shift time, respectively, (for one operation) then we can express the total parse time $T_{p1}$ with a single processor as

$$T_{p1} = (N - L) \cdot t_r + L \cdot t_s$$

In parallel parsing shift operations are executed in parallel. The reduction operations corresponding to the internal nodes below level $h$ are executed almost independently and in parallel (as the number of nodes at each level exceeds the number of processors). The internal nodes at levels 1 to $h$ require $h$ units of reduction time [4], apart from the processor coordination and communication time. Therefore,

$$T_{pq} \simeq \frac{(N - L - N_c) \cdot t_r}{q} + \frac{L \cdot t_s}{q} + h \cdot t_r + T_{cq}$$

which gives the speedup with $q$ processors as

$$SP(q,L) \simeq \frac{(N - L) \cdot t_r + L \cdot t_s}{((N - L - N_c) \cdot t_r + L \cdot t_s)/q + h \cdot t_r + T_{cq}} \qquad (2)$$

The number of nodes, $N$, depends on the length of the string and the language. The level $h$ depends on the number of processors, and the language. The processor coordination and communication time, $T_{cq}$, depends on the number of processors as well as on the connection topology. First we determine $N$, $N_c$, and $h$. Then $T_{cq}$ will be estimated for both models in the next section.

Consider a deterministic, context–free language with $m$ production rules and $v$ nonterminals. Let the production rules be numbered as $1, 2, \ldots, m$. In derivation of a string of length $L$, let the ith production rule be used $r_i$

times. Then we can express the number of internal nodes in the parse tree as,

$$N - L = \sum_{i=1}^{m} r_i \tag{3}$$

In [6] a method was developed for evaluating $r_i$ in terms of occurrences of $(m-v)$ terminals $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_\alpha$, where $\alpha = (m - v)$. Using this method, the number of uses of each production in terms of the occurrences of the terminals if, else, case, while, repeat, ;, +, *, > and () for a Pascal–like language (as in [5]) is shown in Table 1, where $n_{\Upsilon_i}$ is the number of times terminal $\Upsilon_i$ occurs.

| Rule No. | Rule | Number of uses in a successful parse |
|---|---|---|
| 1 | P := S | 1 |
| 2 | S := S; I | $n_;$ |
| 3 | S := I | $n_{if} + n_{else} + n_{while} + n_{repeat} + n_{case} + 1$ |
| 4 | I := id ← E | $n_; + n_{else} + 1$ |
| 5 | I := if B then S fi | $n_{if} - n_{else}$ |
| 6 | I := if B then S | |
| | else S fi | $n_{else}$ |
| 7 | I := while B do S od | $n_{while}$ |
| 8 | I := repeat S until B | $n_{repeat}$ |
| 9 | I := case E of S end | $n_{case}$ |
| 10 | E := E + T | $n_+$ |
| 11 | E := T | $n_; + n_{else} + n_{case} + n_{()} + 2n_> + 1$ |
| 12 | T := T * F | $n_*$ |
| 13 | T := F | $n_+ + n_{()} + n_; + n_{else} + n_{case} + 2n_> + 1$ |
| 14 | F := id | $n_* + n_+ + n_; + n_{else} + n_{case} + 2n_> + 1$ |
| 15 | f := (E) | $n_{()}$ |
| 16 | B := E [> / ≥ / < / ≤ / = / ≠]$E$ | $n_>$ |

TABLE 1 Syntax of Pascal–Like Language and Count Relations Between Terminals

If the frequency of occurrence of the terminals is known, we can estimate $N - L$. Two methods are discussed in [6] to determine the average frequency of the terminals. Using these techniques we can approximate the number of internal nodes as a fraction of $L$, as $N - L = k' \cdot L$, where $k'$ depends on the language. Therefore, we can write

$$N = k \cdot L, \quad \text{where} \quad k = k' + 1 \tag{4}$$

The average number of sons of an internal node is

$$d = \frac{N-1}{N-L} = \frac{k - 1/L}{k-1}$$

The average number of sons for internal nodes at levels 1 to $h$ is also assumed to be $d$. Assuming that at level $h$ there are exactly $q$ internal nodes, we can write

$$q = d^{h-1}$$

From which we get

$$h = log_d(q) + 1 \tag{5}$$

Similarly, $N_c$, the number of nodes at levels 1 to $h$, can be expressed in terms of $q$ and $d$ as follows:

$$N_c = \frac{d^h - 1}{d - 1} \quad \text{i.e.,}$$

$$N_c = \frac{d \cdot q - 1}{d - 1} \tag{6}$$

Substitutions of (4), (5), and (6) in (2) gives,

$$SP(L,q) \simeq \frac{k' \cdot L \cdot t_r + L \cdot t_s}{((k' \cdot L - (d \cdot q - 1)/(d-1))t_r + L \cdot t_s)/q + (log_d(q) + 1)t_r + T_{cq}} \tag{7}$$

In the next section we estimate the coordination and communication time $T_{cq}$ for Model A and Model B and derive expressions for speedup for a Pascal–like language.

## V. SPEEDUP FOR MODELS A AND B

In [4] it was shown that the upper bound for speedup for parallel parsing increases monotonically with the number of processors and reaches a limiting value. Beyond this critical number of processors no further speedup is obtained. In obtaining this result Cohen and Hickey [4] neglected the processor coordination and communication time. Furthermore, they conjectured that the average speedup curve for strings of a given length would be of the same shape as their maximum speedup curve.

In this section we show that the Cohen–Hickey conjecture holds for Model B; but for Model A we get an expression for speedup which is close to the simulation result obtained in [5], but quite different from the speedup conjectured in [4].

The processor coordination and communication time depends on the model for the parallel parsing. We determine the value of $T_{cq}$ for each of the two models and then substitute them to get the expression for the speedup.

Model A: In Model A the average coordination and communication time $T_{cq}$ is determined by the average number of tokens left in $STK_q$ and the number of processors, $q$. For a merge request to travel to $P_q$ from $P_1$, it takes $(q-1)$ units of time, and for the first token to reach $P_1$ takes $(q-1)$ units of time, where the unit of time is the period required by two adjacent processors to exchange a merge or datum. If $k_1$ is the average number of tokens in processor $P_q$'s stack (when it enters into merge–only state) then next $(k_1 - 1)$ tokens can be passed to $P_1$ in next $(k_1 - 1)$ units of time using pipeling. This gives

$$T_{cq} = 2(q-1) + (k_1 - 1)$$

Model B: In this model every processor can communicate with every other processor directly. Hence, to collect all irreducible tokens from $P_q$, in $h$ reduction steps, $P_1$ may need $(h-1)$ request and $k_1$ data transfers. Hence,

$$T_{cq} = (h-1) + k_1$$
$$\text{or,} \quad T_{cq} = log_d(q) + k_1$$

Next we estimate $k_1$, the average number of unparsed tokens on a stack.

Estimating $k_1$, the Average Number of Tokens: Let $\sigma_i$ be the number of tokens left on $STK_i$ when $P_i$ completes parsing its part of the input. We define,

$$\sigma = Max(\sigma_i,\ i \in \{1,2,\ldots,q\})$$
$$\text{Then,} \quad k_1 = \sum_{i=1}^{\sigma} ipr(i) \tag{8}$$

Where $pr(i)$ is the probability that at least one stack has $i$ tokens and no stack has more than $i$ tokens.

Assuming that a processor has any number of tokens between one and $\sigma$ with equal probability of $\frac{1}{\sigma}$, we derive an expression for $pr(i)$.

Probability that $STK_1$ has $\sigma$ tokens when $P_1$ completes parsing the input to it is $\frac{1}{\sigma}$. Probability that $STK_1$ has fewerer than $\sigma$ tokens but $STK_2$ has $\sigma$ tokens is $(1-\frac{1}{\sigma}) \cdot \frac{1}{\sigma}$. Similarly, the probability that $(q-1)$ stacks have fewer that $\sigma$ tokens and $STK_q$ has $\sigma$ tokens is given by,

$$(1 - \frac{1}{\sigma})^{q-1} \cdot \frac{1}{\sigma}$$

The probability that at least one stack has $\sigma$ tokens,

$$\text{i.e.,} \quad pr(\sigma) = \frac{1}{\sigma} + (1-\frac{1}{\sigma}) \cdot \frac{1}{\sigma} + \cdots + (1-\frac{1}{\sigma})^{q-1} \cdot \frac{1}{\sigma}$$
$$= 1 - (1-\frac{1}{\sigma})^q$$

Similarly we get,

$$pr(\sigma - 1) = (1-\frac{1}{\sigma})^q (1 - (1-\frac{1}{\sigma})^q),$$

In general,

$$pr(\sigma - i) = (1-\frac{1}{\sigma})^{i \cdot q}(1 - (1-\frac{1}{\sigma})^q),$$
$$\text{for} \quad i \in \{1,2,\ldots,\sigma-1\}$$

and

$$pr(1) = 1 - \sum_{i=2}^{\sigma} pr(i) = (1-\frac{1}{\sigma})^{(\sigma-1)\cdot q}$$

Substitution of $pr(i)$ in equation (8) gives

$$k_1 = \sum_{i=0}^{\sigma-2}(\sigma - i)(1-\frac{1}{\sigma})^{i\cdot q}(1-(1-\frac{1}{\sigma})^q) + (1-\frac{1}{\sigma})^{(\sigma-1)\cdot q}$$

$$= (1-(1-\frac{1}{\sigma})^q)\sum_{i=0}^{\sigma-2}(\sigma-i)(1-\frac{1}{\sigma})^{i\cdot q} + (1-\frac{1}{\sigma})^{(\sigma-1)\cdot q}$$

$$= \frac{\sigma - (\sigma+1)(1-\frac{1}{\sigma})^q - 2(\sigma-2)(1-\frac{1}{\sigma})^{(\sigma-1)\cdot q} + (2\sigma-3)(1-\frac{1}{\sigma})^{\sigma\cdot q}}{1-(1-\frac{1}{\sigma})^q}$$

In practical situations $\sigma \geq 2$ and if the number of processors is large then we can approximate $k_1$ by

$$k_1 \simeq \sigma - \frac{(1-\frac{1}{\sigma})^q}{1-(1-\frac{1}{\sigma})^q}$$

When $\sigma = \frac{L}{q}$, then we get $k_1 \simeq \frac{L}{q} - \frac{L}{q^2}$.

In the rest of the paper this expression for $k_1$ is used.

Nature of Speedup Function: The expression for average speedup for Model A is

$$SP1(L,q) \simeq \frac{k' \cdot L \cdot t_r + L \cdot t_s}{((k' \cdot L - (d \cdot q - 1)/(d-1))t_r + L \cdot t_s)/q + (log_d(q) + 1)t_r + 2(q-1) + \frac{L}{q} - \frac{L}{q^2} - 1} \tag{9}$$

The general shape of the speedup curve for a given length of the string with varying number of processors can be obtained as follows.

The numerator in $SP1(L,q)$ does not depend on the number of processors. Hence, we consider only the denominator. Let the denominator be denoted by $DSP1$. Taking the first derivative of $DSP1$ with respect to $q$ and equating this to zero, (after removing those terms that asymptotically go to zero), the value of $q$ is

$$q = ((k' \cdot t_r + t_s + 1)L/2)^{1/2} = q_0$$

Substituting $q_0$ in the second derivative of $DSP1$ (with respect to $q$) an expression with positive value is obtained. Therefore, $q_0$ is the number of processors which parse a string of length $L$ in a minimum time. The speedup increases with the number of processor up to a maximum and then decreases.

Similarly, the expression for speedup for Model B is given by

$$SP2(L,q) \simeq \frac{k' \cdot L \cdot t_r + L \cdot t_s}{((k' \cdot L - (d \cdot q - 1)/(d-1))t_r + L \cdot t_s)/q + (log_d(q) + 1) \cdot t_r + log_d(q) + \frac{L}{q} - \frac{L}{q^2}} \tag{10}$$

It can be shown that $SP2(L,q)$ increases to a maximum value monotonically, and then it remains constant. Unlike $SP1(L,q)$, $SP2(L,q)$ does not decrease as the number of processors increases beyond the critical number.

Speedup for Pascal–like Languages: To find the speedup for Pascal–like languages we calculate $N$, estimated number of nodes and $d$, estimated degree in the parse tree as follows.

From Table 1 we get,

$$N - L = \sum_{i=1}^{16} r_i$$

$$=(5n_{;} + 2n_{if} + 5n_{else} + 2n_{while} + 2n_{repeat} + 5n_{case}$$

$$+ 3n_{+} + 4n_{()} + 7n_{>} + 2n_{*} + 6) \qquad (11)$$

| Terminals | Frequency of occurrence (every 100 terminals) |
|---|---|
| id | 60 |
| ← | 6 |
| if | 2 |
| else | 0.9 |
| while | 0.1 |
| repeat | 0.05 |
| case | 0.15 |
| () | 6.6 |
| ; | 12 |
| + | 4.6 |
| * | 4.6 |
| > | 4.6 |

TABLE 2 Average occurrence of some terminals in Pascal like languages Ref. [5]

Using expression (11) and the frequency of occurrence of each terminal shown in Table 2 we get

$$N = 2.4175L$$

$$d = 1.70547$$

Substituting these value of $N$ and $d$ in (9) and (10) we get

$$SP1(L,q) \simeq \frac{1.4175L \cdot t_r + L \cdot t_s}{((1.4175L - (1.7047q - 1)/0.7047)t_r + Lt_s)/q + (log_{1.7047}(q))t_r + t_r + 2(q-1) + \frac{L}{q} - \frac{L}{q^2}}$$

and,

$$SP2(L,q) \simeq \frac{1.4175L \cdot t_r + L \cdot t_s}{((1.4175L - (1.7047q - 1)/0.7047)t_r + L \cdot t_s)/q + (log_{1.7047}(q) + 1)t_r + log_{1.7047}(q) + \frac{L}{q} - \frac{L}{q^2}}$$

In Fig. 2 the speedup curves are given with $t_r = t_s = 1$ and $L = 1000$. The dotted curve presents the speedup obtained by simulation in [5].

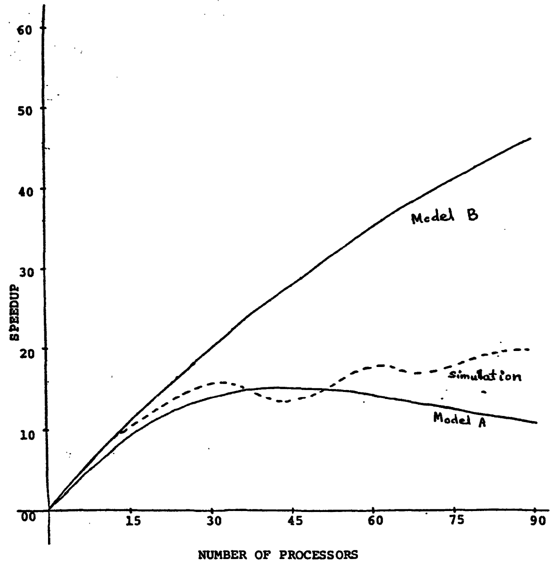


Fig. 2. Number of Processors vs Speedup Curves for Pascal–like Languages with $k_1 = \frac{L}{q} - \frac{L}{q^2}$

## VI. CONCLUSION

We have presented a method for estimating the speedup for asynchronous, bottom–up, parallel parsing. To develop this result we have made a few simplifying assumptions about the nature of the parse tree. Hence the expressions may not give exact speedup, but the closeness of the estimated speedup using the method developed here and the simulation result in [5] indicates that the assumptions are realistic, and that the significant parameters have been taken into account.

Next, it may be useful to investigate how the structure of a language determines the speedup in parallel compilation. Parallelization of the existing sequential code–generation techniques may be useful for speedup and machine utilization.

The Model A is restrictive in that a processor can directly communicate with its immediate left and immediate right neighbors only. This increases the processor coordination and interprocessor communication time. On the other hand, Model B is expensive to construct if the number of processors is large. A model which is a combination of the two, for example $Cm^*$ [11], should also be investigated.

Acknowledgement: We would like to thank Sajal Kumar Das for his careful reading of the manuscript.

## REFERENCES

[1] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing,* Englewood Cliffs, NJ: Prentice–Hall, 1972.

[2] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Reading, MA: Addison–Wesley, 1977.

[3] J. -L. Baer and C. S. Ellis, "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment," *IEEE Trans. on Soft. Engg.,* Vol. SE–3, No. 5, Nov. 1977.

[4] J. Cohen and T. Hickey, "Upper Bounds for Speedup in Parallel Parsing," *JACM,* Vol. 29, No. 2, April 1982.

[5] J. Cohen and S. Kolodner, "Estimating the Speedup in Parallel Parsing," *IEEE Trans. on Soft. Engg.,* Vol. SE–11, No. 1, Jan. 1985.

[6] J. Cohen and M. S. Roth, "Analyses of Deterministic Parsing Algorithms," *CACM,* Vol. 21, No. 6, June 1978.

[7] E. Dekel and S. Sahni, "Parallel Generation of Postfix and Tree–forms," *ACM Trans. Prog. Lang. Syst.,* Vol. 5, No. 3, July 1983.

[8] M. K. Donegan and S. W. Katzke, "Lexical Analysis and Parsing Techniques for a Vector Machine," *Proc. Conf. Prog. Lang. and Compilers for Parallel and Vector Machines, ACM-SIGPLAN Notices,* Vol. 10, March 1975.

[9] C. A. Ellis, "Parallel Compiling Techniques," *Proc. ACM 26th Nat. Conf.,* 1971.

[10] C. N. Fischer, "On Parsing Context–free Languages in Parallel Environments," Ph.D. Dissertation, Dept. Comput. Science., Cornell Univ., Ithaca, NY, April 1975.

[11] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw–Hill, 1984.

[12] H. E. Krohn, "A Parallel Approach to Code Generation for Fortran–like Compilers," *Proc. Conf. Prog. Lang. and Compilers for Parallel and Vector Machines, ACM-SIGPLAN Notices,* Vol. 10, March 1975.

[13] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Comput. Surveys,* Vol. 9, March 1977.

[14] D. Ligett, G. McCluskey and W. M. McKeeman, "Parallel LR Parsing," Wang Institute of Graduate Studies, School of Information Technology, Technical Report TR-82-03, July 1982.

[15] M. D. Mickunas and R. M. Schell, "Parallel Compilation in a Multiprocessor Environment," *Proc. ACM* 1978.

[16] M. Weiser, "Program Slicing," *IEEE Trans. on Soft. Engg.,* Vol. SE–10, No. 4, July 1984.

# PTOOL:

## A Semi-automatic Parallel Programming Assistant [†]

*Randy Allen*[‡]
*Donn Bäumgartner*
*Ken Kennedy*
*Allan Porterfield*

Department of Computer Science
Rice University
Houston, Texas 77251

## Abstract

We present a new parallel programming assistant, called PTOOL, that identifies loops suitable for parallel execution. It employs a theory of dependence to determine which loops in a FORTRAN program will produce the same answers when iterations are scheduled in an arbitrary order. When a loop fails the test, an explanation is provided in the form of a report of the dependences that can be violated by some schedule. PTOOL also provides assistance in the allocation of variables between local and shared global memory. Interprocedural data flow analysis techniques extend its scope of applicability to whole programs. PTOOL, which is in use at Los Alamos National Laboratory, represents a first step toward a sophisticated interactive programming system based on incremental dependence analysis.

## 1. Introduction

Humans and machines bring very different strengths to bear on the problem-solving process. Humans make effective use of abstraction to develop broad strategies and are adept at simplification of complex situations and at generalization from simple principles. Unfortunately, they are not good at exhaustive search and seem unable pay rigorous attention to details—areas where the computer excels. As a result, programmers often fail to precisely specify the details of their intended solutions to the machine, inevitably leading to "bugs" in their programs.

The advent of parallel machine architectures has added a new level of complexity to the debugging process. Parallelism implies nondeterminism; nondeterminism, in turn, implies nonrepeatability. In other words, program sections that are executed in parallel do not necessarily follow the same execution path every time the program is run—even when the program is run on the same data. This additional complexity requires additional precision from the programmer. Not only must he ensure that the computations performed by the sequential sections are those that he intended, but he must also ensure that the computations indicated in parallel regions perform the desired calculations regardless of the order in which they are executed. This increased complexity permits some very elusive bugs.

Given the difficulties that can arise in parallel programming, it seems clear that new methods and tools will be needed [18]. There are several possible lines of attack. At one extreme, a parallel program might be repeatedly run on the same test data to ensure that it computes the same answers on each execution. Although this approach is easy to implement, it is inelegant and probably would not be effective, since the conditions required to produce an error need not arise during the test. At the opposite extreme, formal correctness proofs could be applied to a parallel program. In spite of significant levels of activity on verification over the past decade, complete automatic proofs for realistic programs appear impractical for the time being.

A third approach would be to develop an automatic system to convert implicit parallelism within a sequential program into an explicit form. The advantage of this method results from the ability to debug the program using standard techniques on sequential machines prior to parallel conversion. Since the conversion involves provably correct transformations, the parallel version should execute correctly, thereby avoiding the need for a complex parallel debugging phase. Additionally, this approach provides a mechanism for converting "dusty deck" programs.

In spite of these advantages, fully automatic conversion to parallelism is not practical using current language processing technology. Although many promising techniques have been developed [10,22,21,6], the programmer is still an essential ingredient in the exploitation of parallelism. There are several reasons for this. First, most programs contain a large number of potential parallel regions. Searching all regions for parallelism can be an extremely expensive task. In addition, synchronization overhead on many machines can easily overwhelm the advantages of parallel execution [12], unless the granularity of the parallel regions is very large. Often, this determination depends upon values that are known only at run-time. Hence, the programmer's judgement is an important part of the parallel programming process.

This paper describes a new parallel programming assistant, called PTOOL, that combines the advantages of the manual and automatic approaches. When using PTOOL, a programmer develops and debugs a program on a sequential machine using traditional techniques. Once the program is debugged, the programmer identifies the regions, usually separate iterations of loops, that should be executed in parallel. PTOOL then reports whether the results of execution ***depend in any way on the order of execution of the regions.*** If not, the regions can be safely executed in parallel. Otherwise, the programmer can ask PTOOL for diagnostic information to help identify problems. This information is presented as reports on potential conflicts in the use of variables shared by different parallel regions.

The discussion of PTOOL begins in Section 2 with an overview of the parallel programming approach it supports. Section 3 gives an introductory treatment of dependence theory, the primary tool for analyzing programs. The method used by PTOOL to determine loops that can be executed in parallel is described in Section 4, while details of the system design are presented in Section 5. The user interface is discussed in Section 6. Finally, Section 7 summarizes the work and suggests future directions.

## 2. Parallel Programming Process

This section introduces a model for the parallel programming process that PTOOL is intended to support. Fundamentally, PTOOL is a system designed to help programmers transform sequential loops in FORTRAN to parallel loops. Although this model is not universal, it covers many interesting cases.

Multiprocessors are the most efficient when each processor is able to compute at full speed, without requiring synchronization with other processors. To achieve this ideal, we not only must identify a collection of program fragments that can be executed in parallel, but we must also schedule the computation in a manner that balances the execution load across processors. In FORTRAN, the lingua franca of scientific computation, the construct most likely to give rise to a large number of parallel regions with comparable execution times is the DO loop. There

[†] This research was supported by IBM corporation.

[‡] Author's address is: Dana Group, 550 Del Rey Ave., Sunnyvale, Ca. 94086.

0190-3918/86/0000/0164 $01.00 © 1986 IEEE

are two reasons why. First, if they can be made to execute correctly in parallel, the separate iterations of a DO loop can provide enough computation to keep a significant number of processors busy. Second, since each iteration of a DO loop should take roughly the same time to execute, the load will be automatically balanced across the processors. Experience with actual multiprocessors strongly supports the importance of loops in the parallel programming [19].

In the light of these observations, we have designed PTOOL to determine the conditions that inhibit parallel execution of loops. In doing so, we have been careful not to restrict the analysis to DO loops, because iterative regions in FORTRAN are often coded using backward GOTO statements.

In deciding whether or not the iterations of a given loop can be executed in parallel, it is important to decide which variables mentioned in the loop body must be shared by parallel tasks and which can be allocated to storage local to the processor. This decision is important because shared variables can be accessed in different orders by different processors. If we wish to compute the same results every time the program is run, all possible schedules for a set of parallel tasks must lead to a functionally equivalent access sequence for shared variables. For example, it should not be possible to have two schedules in which the order of a load and store of a shared variable are reversed. If we are to avoid the insertion of explicit synchronization between loop iterations, we must establish that, without synchronization, such a reversal can never happen. In performing the requisite analysis, we need not concern ourselves with local variables, since the relevant loads and stores happen on the same processor.

Since PTOOL was developed to support parallel programming research at Los Alamos National Laboratory, where FORTRAN code was being converted to the Delnelcor HEP and the Cray X-MP[1], the methods for expressing parallelism on those machines strongly influenced its design. The standard method on those machines is to express a parallel body of code as a subroutine, because that expression provides a natural encapsulation of the code to be broadcast to the different processors. Furthermore, reentrant code is straightforward to generate.

In the subroutine model, the loop body is replaced by a parallel subroutine invocation that takes as parameters an indication of the loop iteration to execute and variables that have to be globally available to all processors. PTOOL assumes that subroutine parameters and variables in COMMON are shared, but all other variables used in the loop body are assigned to storage local to each processor. As we shall see, PTOOL offers a suggestion, based upon dependence analysis, of the variables which should be parameters. The programmer is free to ignore this suggestion and select his own parameters. PTOOL will accept this modification and tailor its advice to the set of shared variables specified by the user.

Under the model we have introduced, the principal requirement that must be preserved when converting a program for parallel execution is the access order to shared variables. The main tool for analyzing patterns of loads and stores in a program is a powerful theory of dependence, the subject of the next section.

## 3. Data Dependence

In a sequential language such as Fortran, the execution order of statements is well defined. Therefore, the behavior of the program under sequential execution order can be used as a basis for evaluating other execution orders. Specifically, it is possible to examine alternative execution orders to see whether they will also produce the same results. Parallel execution implies that the statements in a collection of parallel regions (iterations of a loop, for our purposes) can be executed in any order so long as the statements within a particular region are executed in sequence. Hence, for a collection of parallel regions to be suitable for parallel execution, there must be no dependences between statements of different parallel regions. In our model, there must be no dependences that cross loop iterations.

To determine whether this condition holds, we must identify the pairs of statements whose relative execution order *must* be preserved under any program transformation if the results are to be preserved. This relationship is represented by a collection of ***dependences*** among the statements of the program. A statement $S_2$ is said to ***depend upon*** a statement $S_1$ if $S_2$ follows $S_1$ in dynamic execution of the sequential program and it must follow $S_1$ in any reordering that preserves the correct results.

There are two ways for a statement $S_2$ to depend on statement $S_1$. First, $S_1$ can cause a change in the control flow that determines whether $S_2$ is executed, creating a ***control dependence*** of $S_2$ on $S_1$. Second, the two statements may access the same variable in a way that requires that their order be preserved. A dependence created to prevent incorrect access order to a variable is called a ***data dependence***. Although both types of dependence must be considered when rearranging statements, programs can always be transformed so that all control dependences become data dependences[1]. Therefore, we restrict the remainder of this discussion to data dependences.

Data dependence arises most naturally when one statement defines a variable that is later used by a second statement. However, Kuck has identified three types of data dependence [17]:

(1) *true dependence* — $S_1$ stores into a variable that $S_2$ later uses.

(2) *antidependence* — $S_1$ fetches from a variable that $S_2$ later stores into.

(3) *output dependence* — two statements both store into the same variable.

All three types of data dependences must be considered to safely reorder a program.

It is also useful to distinguish dependences that cross loop iteration boundaries from those that do not. To see this, consider the following example.

```
              DO 100 I = 1, N
S1                A(I) = ...
S2                ... = A(I)
        100   CONTINUE
```

$S_2$ quite obviously has a true dependence upon $S_1$, implying that $S_1$ must be executed before $S_2$ in order for the computation to be correct. However, this dependence in no way precludes the separate iterations of the loop from executing in parallel, because values created within the loop are used on the same iteration, and need not be saved for later iterations. Such a dependence is said to be ***loop independent*** [2]. On the other hand, consider a slightly different loop.

```
              DO 100 I = 1, N
S1                A(I) = ...
S2                ... = A(I-1)
        100   CONTINUE
```

In this case, we may ***not*** safely transform the loop to execute in parallel, because the dependence crosses loop iterations; that is, values created on one iteration are used on a later iteration. A dependence of this sort is said to be ***loop carried***, because it exists only by virtue of the iteration of the loop—if the loop body is executed only once, the dependence ceases to exist.

It is useful to observe that a loop carried dependence arises because of the iteration of a particular loop. For instance, in the following nest of loops

[1] HEP is a trademark of Denelcor Corporation and X-MP is a trademark of Cray Research.

```
                DO 200 I = 1, M
                      DO 100 J = 1, N
S1                         A(I,J) = ...
S2                         ...  = A(I-1,J)
          100         CONTINUE
          200   CONTINUE
```

iteration of the outer loop (on I) gives rise to the dependence. So long as the outer loop is iterated sequentially, the dependence will be satisfied. In the light of this observation, we classify carried dependences by indicating the particular loop that creates the dependence.

Loop carried and loop independent dependence provide a precise characterization of the execution orders that are important within a program. So long as these dependences are preserved, the results of a computation will also be preserved. This fact is extremely important in vectorization as performed in PFC, an automatic vectorization system written at Rice [3,16]. PFC constructs a complete dependence graph for the program to which it is applied, in order to distinguish the statements that can be executed as vector operations from those that cannot. This graph can also be used to determine parallel loops, as the next section shows.

## 4. Identifying Parallel Loops

From the parallel programming paradigm presented in section 2, it should be clear that there are two tasks to be performed in converting sequential code: 1) identifying local and global variables, and 2) finding loops suitable for parallel execution. These two tasks are highly interdependent as the following loop illustrates:

```
          DO 100 I = 1, N
                 T = ...
                 ... = T
    100   CONTINUE
```

If T is a variable that can be kept in the local memory of individual processors, as above, then there is no dependence that prevents parallel execution of the loop. If, on the other hand, T must be globally available (as would happen if there were a use of the last value of T after the loop), then the loop would not execute correctly in parallel. Once T becomes a global variable, different processors may intermix intermediate computations involving T, with unpredictable results.

Identifying the variables that must be global to a parallel loop turns out to be a relatively straightforward process. Simply stated, a variable must be globally available if it holds a value that is used outside the loop, or, in the symmetric case, if a definition from outside the loop reaches into the loop. Given the dependence graph described in the previous section, recognition of variables that must be globally allocated becomes a fairly trivial task. The system need only analyze the true dependences coming into and going out of a loop; any variables that give rise to such a dependence must be made globally available. In actual practice, the problem is somewhat more complicated, because full dependence graphs are normally constructed only over loop bodies. However, definition-use chains [15] are quite commonly constructed over an entire program (as they are in PFC), and can be used in the absence of stronger dependence information.

The second aspect of parallel programming is the identification of loops suitable for parallel execution or, more correctly, determination of loops which cannot be executed in parallel. Recall that parallel execution, as used in this paper, means execution of different iterations of a loop on different processors. We can achieve this without synchronization only so long as one processor does not compute a value needed by another processor (or one processor does not store on top of a value needed by another processor, etc.). Accordingly, the key restraints to parallel execution are dependences carried by the candidate loop that involve a global variable. For instance, in the following example

```
          COMMON   A(100,80)
          DO 100 I = 1, 80
                DO 100 J = 1, 100
                      A(J,I) = ...
                      ... = A(J-1, I) + ...
    100   CONTINUE
```

the I loop carries no dependences, and may be safely executed in parallel. The J loop, however, is a different matter. If it is executed on multiple processors, each processor will create values needed by a different processor. Hence, a loop may be executed correctly in parallel if and only if that loop carries no dependences based on a global variable [4].

These two principles are the fundamental considerations in converting a sequential code for parallel execution. Because both conditions are tedious to verify, a human converting a code for parallel execution may easily overlook an important problem. More specifically, a human programmer might not notice that a variable must be globally allocated or might miss a loop carried dependence. Since the resulting faults may manifest themselves as errors only under specific schedules, they can be extremely difficult to locate.

## 5. PTOOL Design

### 5.1. Overview

PTOOL divides quite naturally into two subcomponents: one to construct the dependence graph and the other to answer queries about potential parallelism in the program. The first of these components, called PSERVE, is a modified version of PFC running on an IBM 370. When PTOOL is invoked, the user submits the program source, including all the subroutines, to PSERVE for analysis. PSERVE constructs the dependence graph and saves it in a database of two files. This database is shipped back to the user for interactive analysis.

The user then conducts an interactive dialog with the display facility, called PQUERY. PQUERY shows the actual program source on the screen and permits the user to select a loop for analysis. It uses the database constructed by PSERVE to identify the global variables in the proposed parallel region and it presents any dependences that might impede parallel execution.

While the division of PTOOL's functions into two relatively independent components prohibits incremental reanalysis of the program as the user eliminates problems, it does permit PQUERY to be run on a variety of machines. In fact, we have already implemented versions for an IBM PC and a SUN workstation.

### 5.2. PSERVE

### 5.2.1. PFC Modifications

Before computing a dependence graph, PFC employs a number of preliminary transformations to enhance the precision with which it can determine dependences. These transformations can radically change the structure of a program. Since the purpose of PTOOL was to display the dependences as they exist in the *user's* source program (and not in a transformed program), it was necessary to carefully alter the transformations so that dependences could be accurately calculated for the original program.

The two main preliminary transformations in PFC are "induction variable substitution" [24, 3] and "IF conversion" [1]. Induction variable substitution is the process of replacing auxiliary induction variables in a loop with a direct expression based on the loop induction variable. For instance, in the following example

```
        DO 100 I = 1, 100
              IX = IX - 1
              A(IX) = A(IX) - 1
100     CONTINUE
```

IX is used as an auxiliary induction variable to run the loop "backwards". The difficulty with such variables is also illustrated above—an automatic system cannot easily determine whether the references to A are independent. This difficulty can be removed as follows:

```
        DO 100 I = 1, 100
              A(IX - I) = A(IX - I) - 1
100     CONTINUE
        IX = IX - 100
```

This process, which is essentially the inverse of the classic optimization *strength reduction* [15], is normally performed in PFC so that the array dependences of A may be accurately calculated. The problem with this transformation is that it eliminates some scalar dependences that inhibit parallel execution.

In order to correctly handle induction variables, PTOOL must use multiple passes to add dependences to the graph. Dependences for induction variables must be added before induction variable substitution is performed. After induction variable substitution, array dependences can be accurately calculated and added to the graph. As PTOOL adds dependences for the induction variables, it flags the edges so that it can inform the user that these dependences can be removed by substitution.

Control dependences are transformed into data dependences by a process known as "IF conversion" [1]. The basic idea is to transform a statement whose execution is affected by a transfer of control to a conditional statement, controlled by a Boolean "guard" that exactly represents the conditions under which control flow would have reached the statement

GOTO's can be quite naturally classified into two categories: forward GOTO's and backward GOTO's. Each of these categories may possess the further property of being an *exit* branch, meaning that the branch exits one or more loops. Although non-exit forward branches cannot directly cause any problems with parallelization, they can cause indirect problems by forcing some variables under their control to be global. Thus, while not strictly necessary, forward branches are converted because of this problem.

Backward branches are converted via IF conversion into DO WHILE loops, which permit more effective analysis by PFC. Because of the conversion, the user can select a backward GOTO when analyzing the program in exactly the same manner that he selects a DO loop. IF conversion as implemented in PFC is powerful enough to convert any sequence of branches into an equivalent branchless program.

Transformations are only one of the ways in which PFC attempts to produce a precise dependence graph. Another technique that has proved extremely successful is interprocedural analysis.

### 5.2.2. Interprocedural Analysis

Most automatic vectorizers make no attempt to trace dependence edges across procedure boundaries. As a result, they either ignore loops containing procedure calls (declaring them unvectorizable), or assume that all possible variables (i.e. parameters and COMMON) are modified. PFC employs a more sophisticated approach: it uses iterative data flow analysis on the program call graph to determine the side effects of procedure calls [5]. This additional information permits a substantially more precise dependence graph in the presence of subroutines.

The most significant advantage of this approach is the reduction in size of the dependence graph. On GAMTEB, a 504 line program written at Los Alamos, the dependence graph computed without interprocedural analysis contained 22,998 edges. With the application of interprocedural analysis, the graph was reduced to only 2,605 edges. The smaller graph is beneficial to PTOOL's user both directly and indirectly. Directly, the user no longer sees spurious edges associated with variables in COMMON; hence he is able to narrow his focus to real problems. Indirectly, the user experiences a performance improvement in PQUERY because the database is substantially smaller.

A second benefit of the interprocedural information has been the identification of COMMON variables actually used in calls. A fairly standard programming practice for large programs is to have the same COMMON blocks across all subprograms, thereby avoiding the problems of determining which variables have to be passed as parameters and of determining how actual parameters line up with formal parameters. Since all COMMON variables have to be in global memory, it is possible that a temporary value placed in such a variable at the beginning of a loop iteration will prevent parallel execution. Additionally, access to global memory is usually slower than access to local memory. Because PTOOL is able to pick out precisely the variables that are used and modified across procedures, it can aid a user in moving these variables to local memory.

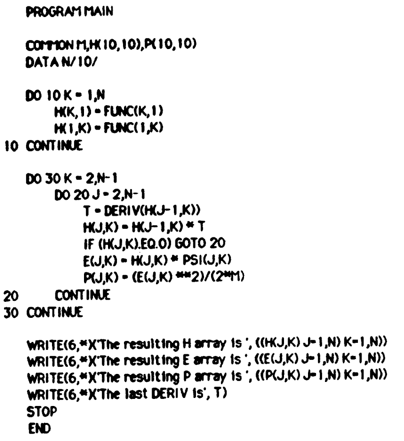
In those situations where a procedure's source code is not available, PFC assumes that all parameters and COMMON variables are both used and modified.

### 5.2.3. Constructing the Graph

Recall that the original goal of PFC was to vectorize FORTRAN programs. It proceeds by generating a dependence graph and then producing a vector FORTRAN equivalent of the input FORTRAN program. For PSERVE, PFC has been modified to execute up to the point that the dependence graph and definition-use chains are assembled. Loop independent edges are then filtered out of the dependence graph and each loop carried edge is annotated with additional information (such as nesting level, whether the variable is in COMMON, etc.) before being added to PSERVE's dependence graph file. In addition, a second file, containing information about loop structure and definition use chains, is built. This information allows the display process to discern which lines make up a loop and identify the parameters of a given loop.

### 5.3. PQUERY

The primary novelty of PTOOL is the interactive display of a dependence graph in PQUERY. This section illustrates the power of PTOOL's display through the example in Figure 1, which captures a number of important characteristics from scientific applications. The code itself can be viewed as solving a wave equation, or computing an unknown function at a mesh of points given only its partial derivative in one direction. The first



```
      PROGRAM MAIN
C
      COMMON M,H(10,10),P(10,10)
      DATA N/10/
C
      DO 10 K = 1,N
         H(K,1) = FUNC(K,1)
         H(1,K) = FUNC(1,K)
   10 CONTINUE
C
      DO 30 K = 2,N-1
         DO 20 J = 2,N-1
            T = DERIV(H(J-1,K))
            H(J,K) = H(J-1,K) * T
            IF (H(J,K).EQ.0) GOTO 20
            E(J,K) = H(J,K) * PSI(J,K)
            P(J,K) = (E(J,K) **2)/(2*M)
   20    CONTINUE
   30 CONTINUE
C
      WRITE(6,*)('The resulting H array is ', ((H(J,K) J=1,N) K=1,N))
      WRITE(6,*)('The resulting E array is ', ((E(J,K) J=1,N) K=1,N))
      WRITE(6,*)('The resulting P array is ', ((P(J,K) J=1,N) K=1,N))
      WRITE(6,*)('The last DERIV is', T)
      STOP
      END
```

Figure 1: Test Program

loop initializes boundary conditions at the border of the mesh. The next loop nest sweeps the calculation across the array. The inner loop moves up a column, calculating the value at any particular location *j* by using the value calculated at location *j*-1. It also performs some auxiliary computations, based on the value at the location. The outer loop sweeps the computation across all the columns. The calculations moving up the columns are all dependent upon previous values, and must proceed sequentially. However, nothing prevents parallel computation of different columns.

In addition to the code that is shown in Figure 1, the source for all of the function and subroutine calls was included when the code was run though PSERVE. The only important property of these procedures was the fact that they were "pure"; that is, they did no READs or WRITEs, and did not access global memory.

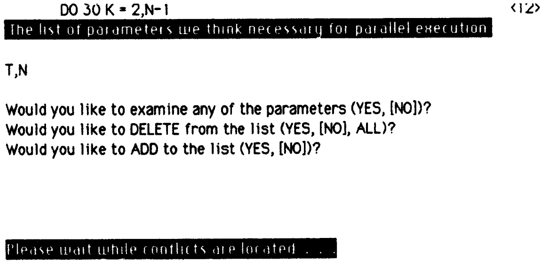
Once PSERVE has constructed the dependence database and downloaded it to the PQUERY machine, the system is ready for an interactive session. The user begins by browsing the complete FORTRAN source file. The first task is to identify loops that are likely candidates for parallelization. For example, in Figure 1, we can see that either of the two outer loops (DO 30 and DO 20) should execute correctly in parallel. A cursory examination of the code reveals no obvious problems with this conclusion.

The PQUERY browser provides a typical set of editor commands for moving about the file (eg. search, go-to-line, page-forward), The user can thus proceed to the first loop to be checked. Once the DO statement (or backwards GOTO) defining this loop is visible on the screen, the loop can be selected by placing the cursor on the appropriate line (using either a mouse or cursor keys) and hitting a selection key.

When a loop is selected, PQUERY displays a list of variables not in COMMON that need to be global to the loop. As stated earlier, these variables must be parameters to the resulting parallel subroutine call. Figure 2 presents the screen that a user would see after selecting the DO 30 loop (from the code in Figure 1) for parallel execution. PTOOL has detected two variables that must be shared—T and N. T is somewhat surprising, since it appears to be a local temporary.

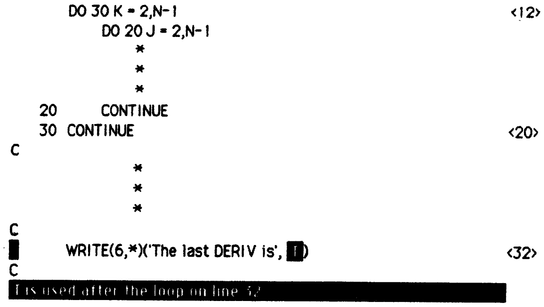
Since it is natural for the programmer to question PTOOL's decisions, we provided a simple explanation facility. For example, if the programmer questions the global allocation of T, he will be presented with the screen in Figure 3. The figure shows that T is used on line 32 after being defined in the selected loop (lines 12-20)—a use outside the loop. The variable, if present in the source (it is possible that COMMON variables will not appear at a call site), is highlighted and the reason that the variable must be global is given.

After reviewing the parameters, the user can add or delete any parameters. This permits PTOOL to solve a number of problems. For instance, on one pass over a loop, a user can allow all of the parameters to remain and discover the scope of the overall problem. Then he can delete all parameters to investigate dependences on variables in COMMON. If dependences are present, the user can attempt to eliminate them by transforming the program source. When no dependences arise from COMMON variables, the user can examine parameters singly, using transformations to eliminate each of the problems. PSERVE can then be invoked on the transformed source to verify that the process has been successful.



```
          DO 30 K = 2,N-1                                  <12>
 The list of parameters we think necessary for parallel execution

 T,N

 Would you like to examine any of the parameters (YES, [NO])?
 Would you like to DELETE from the list (YES, [NO], ALL)?
 Would you like to ADD to the list (YES, [NO])?



 Please wait while conflicts are located . . .
```

Figure 2: Global variables as displayed by PTOOL



```
      DO 30 K = 2,N-1                                  <12>
         DO 20 J = 2,N-1
             *
             *
             *
  20     CONTINUE
  30  CONTINUE                                         <20>
C
             *
             *
             *
C
█     WRITE(6,*)('The last DERIV is', T)               <32>
C
T is used after the loop on line 32
```

Figure 3: PTOOL's explanation of the parameter list

Figure 4 illustrates PTOOL displaying dependence problems after the user is satisfied with the parameter list. PTOOL has highlighted the variable T in two statements, because there is a dependence (caused by the global allocation of T) that can cause incorrect parallel execution. If the dependence can be eliminated by forward substitution or induction variable substitution, PTOOL will inform the user of that fact. In the example program, all problem dependences arise because T must be in global storage. If the programmer eliminates that requirement by changing the code (or simulates it by removing T from the parameter list), no conflicts will appear.

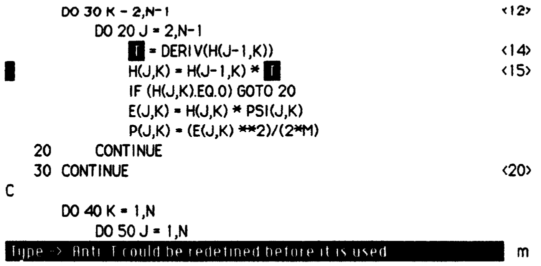
## 6. An Assessment

A number of studies have observed that there are two general classes of errors introduced in converting programs for parallel execution [20,7].

1) Errors due to unintentional data sharing or access to shared variables in an improper order.

2) Errors due to incorrect synchronization code.

PTOOL is effective in helping the programmer identify the causes of errors in the first class. By displaying dependences in an understandable manner, PTOOL can immediately pinpoint problems that might require enormous amounts of unaided human time to find. In fact, in the first demonstration it found a problem that had consumed three man-months of effort at Los Alamos. Interprocedural analysis has been an essential element in the success of PTOOL, because it eliminates most of the spurious dependences caused by large COMMON blocks. Furthermore, it makes it possible to analyze complete Fortran programs of substantial size.



```
      DO 30 K = 2,N-1                                  <12>
         DO 20 J = 2,N-1
            T = DERIV(H(J-1,K))                        <14>
█           H(J,K) = H(J-1,K) * T                      <15>
            IF (H(J,K).EQ.0) GOTO 20
            E(J,K) = H(J,K) * PSI(J,K)
            P(J,K) = (E(J,K) **2)/(2*M)
  20     CONTINUE
  30  CONTINUE                                         <20>
C
      DO 40 K = 1,N
         DO 50 J = 1,N
Type -> Anti  T could be redefined before it is used   m
```

Figure 4: An anti-dependence displayed by PTOOL

In this context, it is useful to contrast PTOOL with the DAPP (Data Flow Analysis for Parallel Programs) system of Appelbe and McDowell [7]. When completed, DAPP will accept a Fortran program in which synchronization code has already been inserted and produce a report of *parallel access anomalies*—pairs of statements that can access the same location simultaneously during a legal schedule. DAPP has the advantage over PTOOL of applicability to a variety of concurrent structures. On the other hand, it uses an analysis technique that is potentially exponential in the size of the program (the single-procedure analysis of PTOOL is proportional to the square of the procedure size and the interprocedural analysis runs in time almost linear in the size of the call graph). Another drawback of DAPP is that it presents an exhaustive report of potential anomalies. Our experience with PTOOL and the experience of other researchers [9] is that exhaustive reporting produces an enormous amount of information in which the important facts can be lost. The selective display provided by PTOOL permits the programmer to focus on one problem at a time. For these reasons, we think that PTOOL will be more useful in analyzing large programs.

On the other hand, PTOOL is of no help whatever for problems of the second class—errors in synchronization code. It is our belief that writing synchronization code is inherently difficult and that programmers should be able to deal with parallelism at a higher level of abstraction. There are several systems of language extensions, implemented through preprocessing, that provide a higher-level interface [8,14,11].

Another approach would be to extend PTOOL to support parallel programming in a more direct fashion. For example, once the programmer has dealt with all dependences that prohibit parallel execution of a given loop, he might request that the system generate the code required to initiate and synchronize the parallel execution. Such a facility would be the first step in evolving PTOOL from a debugging aid to a programming system. Dependence analysis can be a very powerful tool during the program development process. When displayed in an informative way, dependences can provide information about how effectively the program is making use of parallel hardware.

One major drawback to using PTOOL as a programming support system is that it is not sufficiently interactive, since it cannot redisplay dependences immediately after a change. Redisplay requires that PSERVE be invoked again on the whole program, an expensive process. If PTOOL is to be a truly effective programming support tool, it must be converted to an interactive system. But an interactive programming system must also support other functions, such as editing, compilation, and execution. In short, the system must be a complete programming environment.

An interactive programming environment is appealing for other reasons. One of the main hindrances to accurate dependence analysis is variables whose values are known only at run-time. A compile-time analysis must make worst case assumptions about the values of such variables, resulting in dependences that may not be present at run-time. An interactive system can query the programmer regarding values of such variables, and can embed the information provided into the dependence graph and into the resulting code (in case a programmer's assertion turns out to be false). Additionally, a programming environment makes reasonable an enhanced form of interprocedural analysis, in which the effects of procedures on portions of arrays and vectors can be summarized. Such analysis can be especially beneficial on a parallel multiprocessor, since the ability to run procedure invocations in parallel is one of its primary advantages[23].

For these and other reasons, we are convinced that the proper context in which to employ dependence analysis is a programming environment such as $\mathbf{R}^n$[13]. Such a system takes advantage of the strengths of the system and the programmer: the programmer is able to concentrate on developing highly parallel algorithms, with prompting from the system when he strays from parallelism, while the system is able to focus on the mundane aspects of uncovering and scheduling parallelism, with help from the programmer when some information regarding the run-time values of variables is necessary. The resulting system should provide a highly effective mechanism for developing parallel programs.

Although PTOOL is still a fairly primitive system, it has demonstrated that loop carried dependence, properly presented, can be an effective debugging tool. We see it as a first step toward more powerful parallel programming tools based on interactive computation of loop carried dependence.

## Acknowledgements

Version 0.0 of PTOOL was a two inch thick paper listing of the data dependences in TRAC placed on Paul Fredrickson's desk in 1984. The system described in this paper evolved from version 0.0 due to the efforts of many people. Most influential were a number of Los Alamos personnel who are intimately involved with parallel machines, and who provided insight into the problems encountered in parallel programming. These people include Paul (who conceived of PTOOL in its first incarnation), Olaf Lubeck, Margaret Simmons, Jim Moore, Bob Hiromoto, Elizabeth Williams, and Anne Hays. Vasanth Balasundaram has invested large amounts of coding time in an Xedit version of PTOOL. To these people go our heartfelt thanks.

## References

[1] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence,"*Conference Record of the Tenth Annual POPL*, Austin, Texas, Jan. 1983, pp. 177-189.

[2] J.R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Dept. of Mathematical Sciences, Rice University, Houston, Texas, Apr. 1983.

[3] J.R. Allen and K. Kennedy, "PFC: a program to convert Fortran to parallel form," *Supercomputers: Design and Applications*, K. Hwang, editor, IEEE Computer Society Press (1985), pp. 186-205.

[4] J.R. Allen and K. Kennedy, "A parallel programming environment,"*IEEE Software* 2:4 (July 1985), pp. 22-29.

[5] J.R. Allen, D. Callahan, and K.Kennedy, "An implementation of interprocedural analysis in a vectorizing FORTRAN compiler," Dept. of Comp. Sci., Rice University, Dec. 1985.

[6] J.R. Allen, D. Callahan, and K. Kennedy, "Program transformations for parallel machines," Dept. of Comp. Sci., Rice University, Feb., 1986.

[7] W.F. Appelbe and C. McDowell, "Anomaly detection in parallel Fortran programs", Proc. Workshop on Parallel Processing Using the HEP, May 1985.

[8] R.B. Babb II, "Programming the HEP with large-grain data flow techniques," in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, J.S. Kowalik ed., MIT Press, 1985.

[9] R. Conradi, "Static flow analysis of large programs—some problems and results," Technical report 22/83, Division of Computer Science, University of Trondheim, Norwegian Institute of Technology, N-7034, Trondheim, Norway, Sept, 1983.

[10] R. Cytron, "Compile-time scheduling and optimization of asynchronous machines", University of Illinois at Urbana, Aug. 1984.

[11] F. Darema-Rogers, D.A. George, V.A. Norton, and G.F. Pfister, "VM/EPEX—A VM environment for parallel execution," IBM Research Report RC11225, Jan., 1985.

[12] H. Flatt and K. Kennedy, "The performance of parallel processors", Dept. of Comp. Sci., Rice University, June 1985.

[13] R. Hood and K. Kennedy, "A programming environment for FORTRAN," Dept. of Comp. Sci., Rice University, June, 1984.

[14] H.F. Jordan, "Structuring parallel algorithms in an MIMD shared memory environment, *Proc. 18th Hawaii Int. Conf. on System Sciences*, Jan., 1985.

[15] K. Kennedy, "A Survey of Data Flow Analysis Techniques," *Program Flow Analysis: Theory and Applications* (S.S. Muchnick and N.D. Jones, editors), Prentice-Hall, New Jersey, 1981, 1-54.

[16] K. Kennedy, "Automatic translation of Fortran programs to vector form," Rice Technical Report 476-029-4, Rice University, Oct. 1980.

[17] D.J. Kuck, *The Structure of Computers and Computations* Volume 1, John Wiley and Sons, New York, 1978.

[18] O.M. Lubeck, P.O. Fredrickson, R.E. Hiromoto, and J.W. Moore, "Los Alamos experience with the HEP," *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, MIT Press, 1985.

[19] O.M. Lubeck and M. Simmons, "An approach to partitioning scientific computations for shared memory architectures," Computing and Communication Division, Los Alamos National Laboratory, Los Alamos, N.M. 87545.

[20] J.R. McGraw and T.S. Axelrod, "Exploiting multiprocessors: issues and options", UCRL-91734, preprint, Lawrence Livermore National Laboratory, Livermore, CA, Oct. 1984.

[21] K.J. Ottenstein, "A brief survey of implicit parallelism detection," in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, J.S. Kowalik, ed., MIT Press, Cambridge, Mass., 1985.

[22] D.A. Padua, "Multiprocessors: discussion of some theoretical and practical problems," University of Illinois at Urbana, 1980.

[23] R.J. Triolet, "Contribution a la parellisation automatique de programmes FORTRAN comportant des appels de procedure," , L'Universite Pierre et Marie Curie (Paris VI), Dec. 1984.

[24] M.J. Wolfe, "Techniques for improving the inherent parallelism in programs," Report 78-929, Dept. of Computer Science, University of Illinois at Urbana, Urbana, Illinois, July 1978.

# The KAP/ST-100: A Fortran Translator for the ST-100 Attached Processor

Thomas Macke, Christopher Huson, James Davies
Bruce Leasure, Michael Wolfe

Kuck and Associates, Inc.
1808 Woodfield Drive
Savoy, IL 61874
217/356-2288

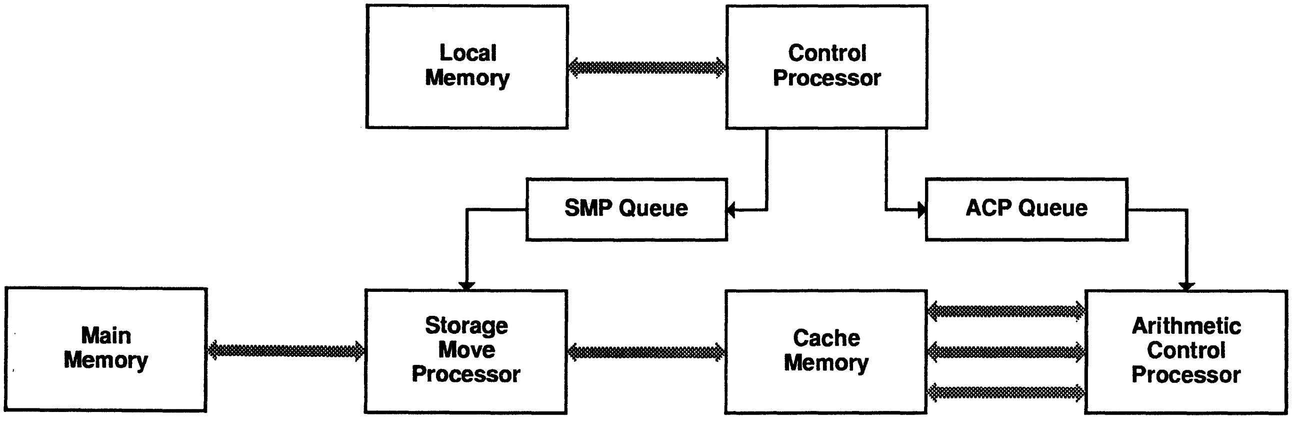
**Abstract.** The KAP/ST-100 is a Fortran source translator that simplifies the task of programming the Star Technologies' ST-100 attached processor. The ST-100 can execute certain vector operations very quickly. It is programmed using a Fortran-like language, called APCL; APCL routines are invoked from the host through system library calls. The KAP/ST-100 locates parts of Fortran programs that will execute efficiently on the ST-100. An APCL routine is created for each of these regions, and the original code is replaced by system library calls to invoke these routines. This paper explains the complications which were encountered in translating for the ST-100 and the methods which were used to overcome them.

## 1. Introduction

The Star Technologies' ST-100 attached processor [1] is designed for high speed execution of single precision (32-bit) signal processing kernels. It also has a wide range of application areas, due to the high speed of its arithmetic unit. The ST-100 is attached to a host computer, such as a VAX, IBM or Control Data mainframe, to which it looks like an I/O device. Since the ST-100 has its own main memory, data items are transferred from the host memory to the ST-100 main memory. The ST-100 executes asynchronously from the host; routines for the ST-100 are written in a Fortran-like language called APCL (Array Processor Control Language) [2]. A program running on the host calls system library routines to schedule execution and acquire ST-100 resources, to send data from the host to the ST-100 main memory, to load the APCL routine to the ST-100, and to initiate execution of that APCL routine. The host program may continue execution or may wait for completion of the APCL routine. The host program then calls other system library routines to bring data back from the ST-100 to the host for further processing, printing, and so on. As with all array processors, writing a program for the ST-100 is a nontrivial project. Because of this, array processors are commonly restricted to special applications. Now with the KAP/ST-100, the power of the ST-100 is available to Fortran programmers.

The KAP/ST-100 is a Fortran translator that eases the task of programming the ST-100. The KAP/ST-100 generates APCL routines for the regions of programs which execute efficiently on the ST-100. Four output files are generated by the KAP/ST-100: an APCL file, a host program file, an auxiliary file to assist in linking the APCL routines, and a listing. The advantages of using the KAP/ST-100 are that the Fortran program remains portable and the user need not learn APCL.

The next section of this paper describes the ST-100 attached processor, focusing on those aspects of the machine that require special handling in the KAP/ST-100. The third section gives an overview of the KAP/ST-100. The fourth section focuses on how the KAP/ST-100 generates APCL code from the Fortran program. The fifth section explains how the KAP/ST-100 decides what sections of code to leave on the host and what sections to move to the ST-100. The sixth section relates some of the problems we encountered with the user interface of the KAP/ST-100. The last section gives an example of the use of the KAP/ST-100 on a simple matrix multiply routine.



**Shaded arrows represent Data Paths**
**Solid arrows represent Control Paths**

Figure 1. Structure of the ST-100 Array Processor.

0190-3918/86/0000/0171 $01.00 © 1986 IEEE

## 2. The ST-100 Attached Processor

The ST-100 is composed of three asynchronous processors and three memories. The Control Processor (CP) executes the user's APCL routine. The CP controls the operation of the other two processors, the Storage Move Processor (SMP) and the Arithmetic Control Processor (ACP). The SMP and ACP are fast processors, with a 40 nanosecond clock period; the CP is a standard microprocessor with no floating point unit. The APCL code resides in Local Memory (LM), which is directly accessible by the CP. Data is sent from the host to the large Main Memory (MM). The ACP, which is the arithmetic engine, performs operations on data in a small, fast, reconnectable private memory, called the Cache Memory (CM). The SMP moves data between the MM and CM. The ACP and SMP are microprogrammed processors; the macro library for each processor can be customized for particular applications and even modified by users.

A block diagram of the ST-100 appears in Figure 1. The CP executes an APCL routine which issues SMP and ACP macros. These macros are loaded into the SMP and ACP queues; the queues allow the CP to continue execution without waiting for the SMP or ACP. Two types of synchronization are available. In the first type, the CP can wait until the queues are empty and activity is complete; this is useful when the CP needs to test a result in order to decide which branch to take in a program, or to end the program. Alternatively, the SMP and ACP synchronize with each other to reconnect the CM; the CM is divided into several banks, each of which is connected to either the SMP or ACP.

### 2.1 Speeds

In order to perform vector operations, data must be moved from the MM to the CM. The SMP can move one word in either direction per 40 ns clock. The ACP is pipelined, and can produce two floating point multiply and two floating point add results in each 40 ns clock.

For simple vector operations, the ST-100 is memory bandwidth limited; for each result produced by the ACP, the SMP must move two operands to the CM, and the result back to the MM. The speed payoff for this machine comes when there are many operations being performed on each datum, such as in a Fast Fourier Transform.

### 2.2 Programming Considerations

Since the CM is relatively small, *strip mining* [3] is required for long vector operations. The size of the strips is not fixed, as would be the case for vector register machines [4], but depends on the number of variables kept in the CM at one time; fewer variables allow longer strips.

To insure the best machine utilization, the SMP and ACP should be kept as busy as possible. This means that some banks of the CM should be connected to the SMP which is loading operands for the next operation or storing results from the previous operation, while other banks of the CM are connected to the ACP. The ACP has three bidirectional connections to the CM and can perform several operations simultaneously. Since the SMP has only one, the number of ACP operations per SMP operand must average at least three in order for the ACP and SMP busy times to be balanced. Temporary results which do not need to be saved in the MM can help increase this operation/operand ratio.

### 2.3 APCL

APCL is syntactically similar to Fortran, but there are special constructs and restrictions which have to do with the way the ST-100 works. For instance, there is no DOUBLE PRECISION data type, since the ST-100 works with only 32-bit floating point numbers. Also, all variables must be explicitly declared as residing in one of the three memories: LM, CM, or MM.

A single APCL routine is called a PROCESS. The APCL process is executed by the CP, just as a Fortran subroutine would be executed by a standard computer. To invoke ACP or SMP macros, the APCL programmer codes macro CALLs; these look like ordinary CALL statements. The APCL compiler compiles macro CALLs into the proper code to build a parameter block and issue the macro to the SMP or ACP queue.

Figure 2 shows a simple Fortran DO loop written in APCL with SMP and ACP macro calls. Figure 2(a) shows a vector multiply written as a Fortran subroutine. Figure 2(b) shows how this would be translated into APCL to use the fast ACP unit.

```
      SUBROUTINE MULT(X,Y,Z)
      REAL X(1000),Y(1000),Z(1000)
      DO 10 I = 1,1000
   10  Z(I) = X(I) * Y(I)
      RETURN
      END
```

(a)

```
      PROCESS MULT(X,Y,Z)
      MAIN MEMORY
      REAL X(1000),Y(1000),Z(1000)
      CACHE MEMORY
      REAL (C1,CZ(1000))
      REAL (C2,CX(1000))
      REAL (C3,CY(1000))
C
      CALL STSYNC (000000)
      CALL SMM2C  (X(1),1,4,0, CX(1),1, 1000)
      CALL SMM2C  (Y(1),1,4,0, CY(1),1, 1000)
      CALL STSYNC (111111)
      CALL AVMUL  (CX(1),1, CY(1),1, CZ(1),1, 1000)
      CALL STSYNC (000000)
      CALL SMC2M  (Z(1),1,4,0, CZ(1),1, 1000)
      CALL STWAP
      END
```

(b)

Figure 2. Vector multiply.

## 3. The KAP/ST-100

The KAP/ST-100 translator automatically translates parts of Fortran subroutines, functions or main programs into APCL code. This translation can proceed with or without help from the user. The KAP/ST-100 will decide what portions of the subroutine to translate, translate the code into APCL, and insert system library calls to move data to and from the ST-100. User directives can be used to tune the generated code in order to improve overall program performance.

The KAP/ST-100 uses the most sophisticated automatic Fortran vectorization technology currently available. The KAP/ST-100 vectorizer includes such advanced transformations as loop interchanging, loop distribution, vector optimization and recognition of many reduction operations. The KAP/ST-100 performs both exact data dependence tests for the most common types of loops and symbolic tests when appropriate. The rest of this section gives an overview of how the KAP/ST-100 works. The actual translation to APCL is explained in detail in the next section.

### 3.1 Scanner

The KAP/ST-100 accepts the Fortran-77 language [5]. Future versions of the KAP/ST-100 will accept certain Fortran extensions, such as VAX/VMS Fortran extensions [6] for VAX hosts. Acceptance of extensions is largely for the convenience of the user, and will not affect the quality of translation.

### 3.2 Candidate Finder

The *Candidate Finder* finds and marks DO loops which are candidates for vectorization. Notice that the KAP/ST-100 also converts IF loops to DO loops when appropriate. This pass is very machine specific, since different vector machines have very different vector instruction sets. For example, the ST-100 can not do DOUBLE PRECISION arithmetic, so the KAP/ST-100 candidate finder rejects any statement containing DOUBLE PRECISION constructs. Each DO loop is examined; if a DO loop has no candidate statements, then the whole loop is marked as unsuitable and vectorization will not be attempted.

Another pass is made that examines all statements, not just statements in DO loops. The KAP/ST-100 does more than convert vector DO loops to vector code; it converts scalar Fortran to scalar APCL. Thus, the KAP/ST-100 includes a pass that examines each statement to see if it can be converted to APCL. Any statement that cannot be converted to APCL is left on the host. This affects how the program gets translated into APCL later.

### 3.3 Preparation Phase

In order to improve vectorization, the KAP performs such common transformations as induction variable recognition [7] and promotion of scalars to arrays [8]. In addition, certain IF constructs are recognized, such as IFs which save the maximum of a vector, or which find the index of such a maximum.

### 3.4 Vectorization

The KAP/ST-100 finds the best method to vectorize each DO loop nest in the program for the ST-100. The vectorizer builds a high quality data dependence graph for each loop nest, which finds where data is generated and used. [9,10] The vectorizer then performs loop transformations such as loop interchanging and loop distribution to find the best loop to vectorize. The data dependence graph is used to check for legality of these operations. Preference is given to long vector lengths.

### 3.5 Process Finder

Up to this point, the KAP/ST-100 has not yet decided what part of the Fortran subroutine to translate to APCL. The *process finder* selects regions of code that will be converted to APCL PROCESSes. Statements that cannot be converted to APCL are left on the host, but not all of the rest of the program will necessarily be translated. Unvectorizable DO loop nests are left on the host computer, unless they appear between two vectorized regions. However, even some vectorized DO loops may be left on the host. For example, a DO loop that does just a vector multiply is not a good candidate for translation to APCL, since the translated program would most likely execute slower than the original program due to the cost of moving data to and from the ST-100. This subject is taken up in more detail in Section 5.

### 3.6 Cleanup

Certain minor cleanup passes over the program are performed at this time. These include a unique pass called *expanded array shrinking*. Since the KAP performs extensive loop interchanging, the loop which will finally be vectorized is not known at the time that scalars are promoted to arrays. To allow the most loop interchanging, the scalars are promoted into multidimensional arrays. After loop interchanging and vectorization, some of the loops will be left scalar, and some of the dimensions are not strictly necessary. The cleanup pass expanded array shrinking removes the unnecessary dimensions of the temporary arrays, reducing their size.

### 3.7 Translation to APCL

Finally, the parts of the subroutine which have been chosen for translation into APCL are examined and converted into APCL PROCESSes. The program segment is replaced with system library calls to properly invoke the APCL PROCESS. This is discussed in more detail in the next section.

## 4. Translation of Fortran into APCL

At this point in the KAP/ST-100, one or more segments of the Fortran subroutine may have been selected for conversion into APCL. Each such segment of the subroutine is converted into a different APCL PROCESS. This conversion takes place in two steps.

### 4.1 Host Interface

First, the segment is scanned to find all variables or arrays that are used or changed. Any variables or arrays whose values are used must be sent from the host to the ST-100, and any variables or arrays whose values are changed must be brought back to the host after the APCL PROCESS completes. Arrays which are assigned but not used in the segment are also sent to the ST-100, because the assignment may be conditional or may only assign part of the array. When the values are brought back from the ST-100, they will write over the values currently in the host.

At this point, the segment of Fortran code is removed from the subroutine and is replaced by library routine calls. A call to the library routine KSTSCH is inserted to reserve a block of ST-100 Main Memory and set time limits for the PROCESS. For each array that must be sent to the ST-100, a call to the library routine KSTARY is inserted to declare that array in MM. Any scalar parameters are packed into two arrays (REAL and INTEGER), and sent over all at once. Then each array is transferred to the ST-100 using a call to the library routine KSTWR. Then a call to KSTLGO is inserted to load and invoke the APCL PROCESS. After KSTLGO, any arrays (or variables) which were modified by the segment of code are brought back from the ST-100 using KSTRD calls. A final call to KSTPRG clears the ST-100 Main Memory for the next process; ST-100 resources are not released, so that they can be reused by the next KAP-generated process.

One problem encountered with many Fortran subroutines is that formal parameter arrays are often declared with an assumed size for the last dimension:

```
REAL A(100,*), or
REAL A(100,1).
```

In Fortran, the upper bound of the last dimension of formal parameter arrays is usually ignored, and may in fact be left unspecified. Thus, the true size of formal parameter arrays may not be known. For the KAP/ST-100 to work properly, all arrays must be explicitly declared to the proper size. A future version will examine loop bounds and array references to find the maximum size used for all arrays, rather than the declared size.

### 4.2 Process Creation

Next the KAP/ST-100 creates the APCL PROCESS out of the Fortran subroutine segment. Because of restrictions and limitations in the APCL compiler, certain scalar optimizations must be performed in the APCL source to get the best speed.

There are no implicitly declared variables in APCL, so all variables that are used in the PROCESS must be explicitly declared, and placed in one of the three memories (MM, LM or CM). The KAP/ST-100 will perform all floating point arithmetic on the ACP, rather than in software on the CP, since the ACP is so much faster; this includes any scalar floating point arithmetic in the "tissue code" between vectorized DO loops. All the code is translated into the appropriate ACPL; the control structure of the PROCESS (scalar IF tests, serial DO loops) will be similar to the structure of the Fortran subroutine. All floating point arithmetic causes SMP and ACP macros to be generated, to move values to the CM, compute some results, and occasionally bring the result back to the CP for testing.

Vectorized DO loops generate a sequence of macro CALLs to load the operands into CM, perform the computation, and store the results. Intermediate results are kept in the CM, so temporary MM arrays are not needed. If the DO loop bound is not known at compile time, or if it is very large, then the loop has to be strip mined, just as would be necessary for a vector register machine.

Since the CP is so much slower than the SMP and ACP, the CP code must be heavily optimized. The KAP/ST-100 helps in this matter by performing source-level optimization, similar to what an optimizing compiler might do. All arrays in the APCL PROCESS are linearized, that is, converted to single dimensional arrays by fully expanding the subscript expressions. This allows the KAP/ST-100 to remove most of the subscript evaluation through common subexpression discovery and code floating. Strength reduction is also important since multiplication on the CP is very slow, even for integers.

The cost of enqueueing a parameter block on the SMP or ACP queues is relatively large; much of this relates to the time taken by the software to build the parameter block. The KAP/ST-100 has no control over this, but it can help by reducing the total number of macro calls. Use of multioperation macros and elimination of unnecessary synchronization macros are two ways of doing this.

### 5. Process Selection

The ST-100 gives the best performance when the program performs many operations on each datum. While a simple vector multiply will execute very quickly in the ACP, the cost of moving the vector multiply from the host to the ST-100 is large:

1. Move the three vectors from host memory to MM
2. Invoke the APCL PROCESS
3. Move the output vector from MM to the host memory.

The APCL PROCESS itself must incur some overhead because it must move the vectors from MM to CM and the result back. Unless floating point multiplication is very slow on the host machine this process will run much slower than performing the operation on the host, regardless of the vector length.

The KAP/ST-100 computes the costs and benefits of using the ST-100 in terms of number of operations per operand. The number of operands is the number of data items that must be moved between the host and the ST-100. In the case of a vector multiply (with a vector length of N), 4N data items must be moved (2N for the input vectors, 2N for the output vector going over and back). In the case of a matrix multiply (with a matrix size of N×N), $4N^2$ data items must be moved (two input matrices and one output matrix of size N×N). The number of operations for a vector multiply is N (N multiplies); for a matrix multiply, it is $2N^3$. Thus, the operation/operand ratio for a vector multiply is N/4N, or 1/4; for a matrix multiply, it is $2N^3/4N^2$, or N/2. This is a very coarse measure of the amount of benefit relative to the overhead cost of moving a set of operations to the ST-100. It is somewhat oversimplified, in that it assumes that all operations are equal cost.

However, even this coarse measure may be used as a starting point in deciding whether or not a set of operations should be moved to the ST-100. Certainly, a large operation/operand ratio will produce more speedup than a small one. A segment of code with a ratio of less than one will never run faster on the ST-100. A segment of code with a ratio that grows with the size of the problem will get even more speedup with large problems than with small; these segments of code are especially suitable for moving to the ST-100.

Adjacent segments of code should be moved together, if at all possible, instead of making two or more adjacent APCL PROCESSes. The decision of whether to move a segment of code to the ST-100 must be made with consideration of the surrounding code. For instance, by itself the vector multiply above is not suitable to move to the ST-100; however, if it were surrounded by two operations that were very suitable for the ST-100, such as matrix multiplies, then the total cost of a PROCESS containing both matrix multiplies and the vector multiply may be less than two PROCESSes each containing one matrix multiply.

The KAP/ST-100 tries to estimate the speedup of moving each independent segment of code to the ST-100. It then looks at adjacent segments of code and combines them as long as there is still speedup. The goal is to generate a program which has the most potential speedup. This goal is made difficult since usually some important parameters, such as the loop bounds, are unknown at compile time. Thus, some loops which look particularly suitable may at run-time turn out to have very short vector lengths, resulting in poor speedup.

### 6. User Interface

The KAP/ST-100 must explain to the user somehow just what happened to his program. The KAP/ST-100 listing shows user's program with the sections that are ported to the ST-100 marked. Those statements that are executed in vector mode are also marked.

One of the problems facing KAP/ST-100 users is how to figure out why one section of code was ported and not another. A variety of conditions affect this decision, and user directives may change the result. One of two DO loops may not be vectorizable or may not fit any known macro set. In this case, it may be faster to leave that loop on the host, and still move the other loop. Another consideration is the amount of data to be moved. One DO loop may use many arrays; the KAP/ST-100 may decide that the operation/operand ratio was too low, and leave that loop on the host. The KAP/ST-100 leaves the last word with the user; KAP directives can force the conversion of a particular segment of code, as long as that segment can be fully converted to APCL.

Another KAP directive modifies the ratio at which code is considered suitable for conversion to APCL. Since the ST-100 is faster relative to a small host (such as a VAX 11/730) than it is to a larger host (like a VAX 8600), some segments of code may be faster on the ST-100 with one host and faster on the host in another configuration. This KAP directive can even be used to force the KAP/ST-100 to convert almost all code that can be converted; this situation may be useful in shops where the critical resource is not time but money, and host CPU time is expensive where ST-100 time is inexpensive.

### 7. Examples and Speeds

The KAP/ST-100 output from a matrix multiply subroutine is shown here, including execution times. The input subroutine is shown in Figure 3(a); the KAP-generated host subroutine is shown in Figure 3(b) and the APCL PROCESS is shown in Figure 3(c). The host program proceeds in seven steps:

1. Call KSTSCH to reserve $3N^2+1$ words of ST-100 MM;
2. Pack all scalars into a temporary array (just N here);
3. Call KSTARY to declare arrays in ST-100 MM;
4. Call KSTWR and KSTWRW to write data values to the ST-100;
5. Call KSTLGO to load and execute the PROCESS MATMU0;
6. Call KSTRDW to read results from ST-100;
7. Call KSTPRG to clear ST-100 MM.

For each array declared in ST-100 MM, a three word descriptor is kept in the host program. This is used to pass that array as a parameter to the APCL PROCESS. The parameters to the KSTLGO subroutine are the PROCESS name, the number of PROCESS parameters, and a "type field" and the descriptor for each of the PROCESS parameters.

The APCL PROCESS bears little resemblance to the original subroutine, but some things do remain similar. The DO J and DO K loops are still recognizable; notice that the DO I loop was interchanged and vectorized for the ST-100. This was done to get stride-one MM accesses; non-one strides on MM moves will run at full speed unless the stride is relatively prime to the number of MM banks (8). The KAP/ST-100 tries for stride-one MM accesses when it can, which are guaranteed to run at full speed. Note also that the parameter arrays are declared as single dimensional arrays with very large bounds (to bypass any bounds checking). All array accesses are done via subscript arithmetic; the IV variables in the code are used for strength reduction and code floating.

In this experiment, the host was an unloaded VAX 11/780 with a floating point accelerator running the VMS operating system. The times for running SUBROUTINE MATMUL on the VAX alone compared with the times for running the KAP/ST-100 processed subroutine are shown in Figure 4. The CPU times shown are the times charged to the batch job for this test; the real times shown are the elapsed times for the batch job and the page faults are also from the batch job log. Notice that the CPU time is the VAX CPU time; there is no way to find the actual time spent in execution on the ST-100, except to infer it from the elapsed time for the job. For matrix size of 100×100, the ST-100 version already runs at least as fast as the VAX version (comparing VAX CPU time to VAX/ST-100 real time). The ST-100 version gets better as the matrix gets larger. Notice also the page fault figure; the VAX has a paged virtual memory and the program begins to thrash when the matrix size gets large. This also seriously affects real time. The SUBROUTINE MATMUL could be optimized for the VAX by interchanging loops to make the array accesses all stride-one. When this is done, the VAX CPU times improve by a factor of two, and the matrix size at which page thrashing begins goes up to somewhere between 300 and 400. Even with that improvement, the ST-100 version will run several times faster than the VAX version, with that ratio increasing as the matrices get larger.

The ST-100 attached array processor offers programmers a low cost high performance computational engine. The KAP/ST-100 gives Fortran programmers a convenient method to access this power.

```
      SUBROUTINE MATMUL(A,B,C,N)
      INTEGER N
      REAL A(N,N),B(N,N),C(N,N)
      DO 100 I = 1,N
      DO 100 J = 1,N
      DO 100 K = 1,N
 100   C(I,J) = C(I,J) + A(I,K)*B(K,J)
      END
```

Figure 3(a). Matrix multiply.

```
      SUBROUTINE MATMUL(A,B,C,N)
      INTEGER N
      REAL A(N,N),B(N,N),C(N,N)
      INTEGER ISCLRS (1), ISCLR0 (3), C0 (3), B0 (3), A0 (3)
      CALL KSTSCH (3 * (N * N) + 1)
      ISCLRS(1) = N
      CALL KSTARY (ISCLR0,1,1)
      CALL KSTARY (C0,N * N,2)
      CALL KSTARY (B0,N * N,2)
      CALL KSTARY (A0,N * N,2)
      CALL KSTWR (C,N * N,C0)
      CALL KSTWR (B,N * N,B0)
      CALL KSTWR (A,N * N,A0)
      CALL KSTWRW (ISCLRS,1,ISCLR0)
      CALL KSTLGO ('(MATMU0)',4,2,ISCLR0,3,C0,3,B0,3,A0)
      CALL KSTRDW (C,N * N,C0)
      CALL KSTPRG
      END
```

Figure 3(b). Host program.

```
        PROCESS MATMU0 (ISCLRS, C, B, A)
        MAIN MEMORY
        INTEGER ISCLRS(16384)
        REAL A(16384), B(16384), C(16384)
        LOCAL MEMORY
        INTEGER IV3, IV2, IV1, IV0, CTMP0, K, J, N, LMISCL(1)
        EQUIVALENCE (LMISCL(1), N)
        CACHE MEMORY
        INTEGER (C1,IC1(16384)),(C2,IC2(16384)),(C3,IC3(16384))
        REAL (C1, C1(16384)), (C2, C2(16384)), (C3, C3(16384))
        EQUIVALENCE (C1, IC1), (C2, IC2), (C3, IC3)
        CALL STSYNC (000000)
        CALL STRDMA (ISCLRS,LMISCL,1)
        IV0 = 1 + N
         IV2 = 1
        DO 101 J=1,N
         IV1 = 1
         IV3 = IV2
        DO 100 K=1,N
        CTMP0 = IV0
         CALL SMM2C (A(IV1),1,4,0,C1(1),1,N)
         CALL SMM2C (B(IV3),0,4,0,C2(1),1,1)
         CALL SMM2C (C(IV2),1,4,0,C1(CTMP0),1,N)
         CALL STSYNC (111111)
         CALL AVMUL (C1(1),1,C2(1),1,C3(1),1,N)
         CALL AVADD (C1(CTMP0),1,C3(1),1,C2(CTMP0),1,N)
         CALL STSYNC (000000)
         CALL SMC2M (C(IV2),1,4,0,C2(CTMP0),1,N)
         IV1 = IV1 + N
         IV3 = IV3 + 1
100      CONTINUE
         IV2 = IV2 + N
101      CONTINUE
         CALL STWAP
         END
```

Figure 3(c). Created process.

| Matrix Multiply Results. | | | | | | |
|---|---|---|---|---|---|---|
| | VAX 11/780 | | | ST100 | | |
| N | cpu-time | real-time | faults | cpu-time | real-time | faults |
| 100×100 | 20.02 | 1:06.79 | 1043 | 4.54 | 17.71 | 1065 |
| 200×200 | 2:26.92 | 7:59.50 | 4411 | 5.65 | 40.08 | 1190 |
| 300×300 | 11:54.90 | 15:25.95 | 513753 | 7.00 | 1:18.77 | 1352 |
| 400×400 | 27:11.52 | 34:24.16 | 1116178 | 8.34 | 2:59.48 | 1451 |

Figure 4.

## References

[1] Star Technologies, Inc., *ST-100 Array Processor Hardware Specifications*, Rev. A, January, 1983.

[2] Star Technologies, Inc., *ST-100 Array Processor Control Language (APCL)*, Preliminary, Rev. F, Release 3, April, 1984.

[3] Loveman, D. B., "Program Improvement by Source-to-Source Transformation," *J. of the ACM*, Vol. 20, No. 1, pp. 121-145, Jan. 1977.

[4] Cray Research, Inc. *Cray X-MP Computer Series Mainframe Reference Manual*, publication HR-0032, 1982.

[5] American National Standards Institute, *American National Standard Programming Language Fortran*, X3.9-1978, 1978.

[6] Digital Equipment Corp, *VAX-11 Fortran Language Reference Manual*, Order. No. AA-D034C-TE, April, 1982.

[7] Aho, A. V. and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

[8] Wolfe, M. J., "Techniques for Improving the Inherent Parallelism in Programs," M.S. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 78-929, July 1978.

[9] Bannerjee, U., S. C. Chen, D. J. Kuck, R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops", *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 660-670, Sep. 1979.

[10] Wolfe, M. J., "Optimizing Supercompilers for Supercomputers", Ph.D. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 82-1009, Oct. 1982.

# DISTRIBUTED LANGUAGES DESIGN:

# CONSTRUCTS FOR CONTROLLING PREFERENCES

by

Tzilla Elrad :: Illinois Institute of Technology

Fred Maymir-Ducharme :: AT&T Bell Laboratories

## ABSTRACT

This paper first classifies controls used in concurrent programming languages within nondeterministic constructs. These controls are classified as private control, consensus control and hybrid control. We then illustrate the need for new and separate control, which we introduce and classify as preference control. Implementations of preference control using pre-existing primitives are illustrated and analyzed with respect to software engineering principles. The difference between preference control and priorities are discussed. Other issues and extensions are also discussed. The classification of these controls will help programmers to better understand and control nondeterministic constructs. The addition of a preference control primitive to a concurrent programming language will increase the language's expressive power if it could not previously implement it; otherwise, the inclusion of preference control as a primitive will simplify the language and satisfy software engineering principles.

**Key Words and Phrases:** Nondeterminism, Concurrency, control classifications, private control, consensus control, hybrid control, preference control, Ada, CSP, Concurrent C, priorities, language design and the implementation of the "pref" primitive.

## 1. *INTRODUCTION*

Many concepts and notations for distributed programming languages have been discussed in the past. [1] [5] [6] [12] [15] [16] This paper suggests and implements the addition of "preference control" to distributed programming languages with nondeterministic constructs. The paper is structured as follows:

Section 2 defines our notations and classifications of controls used to restrict nondeterminism in concurrent programming languages, which we call: private control, consensus control and hybrid control.

Section 3 describes: the need for a primitive to implement "preference control" to constrain nondeterministic constructs; some alternative implementations with the use of pre-existing control primitives; and the introduction and implementation of the "preference control" primitive for further control of nondeterminism in concurrent distributed programming languages.

Section 4 then discusses further extensions and issues raised with the implementation of preference control such as: fairness, priorities and conflicting concepts.

Examples are used from the following concurrent programming languages: CSP [8,9], Ada [11], Concurrent C [7], and Concurrent Prolog[13]. These examples are then analyzed with respect to software engineering standards [3] [10].

## 2. *CONTROL CLASSIFICATIONS AND NOTATIONS*

### 2.1 *NONDETERMINISM*

Pure Nondeterminism: an unconstrained choice from a finite number of alternatives. The select statement in Ada implements nondeterminism. Concurrent C also implements nondeterminism using the select statement. CSP uses the symbols "[" and "]" to surround the nondeterministic choices. Pure nondeterministic constructs are not sufficiently structured; hence, various programming languages have introduced the following controls for nondeterminism, which we have classified as:

### 2.2 *PRIVATE CONTROL*

Private control: nondeterminism restricted by boolean constraints, which we classify as private. Private control is considered open if the boolean expression is true; it is closed otherwise. Example 1a below illustrates private control in CSP:

Example 1a

```
[
 [] B1  ==>  S1
 [] B1  ==>  S2
 [] B2  ==>  S3
]
```

Example 1b -
illustrates private control in Concurrent C:

```
select {
        B1: immediate: S1;
        or
        B1: immediate: S2;
        or
        B2: immediate: S3;
       }
```

Example 1c below illustrates how private control can be indirectly implemented in Ada

0190-3918/86/0000/0176 $01.00 © 1986 IEEE

( although private control was not meant to be allowed in Ada, it can be implemented using the delay construct with 0.0 as the time parameter. ) :

Example 1c

```
select
   when B1 => delay 0.0; S1;
   or
   when B1 => delay 0.0; S2;
   or
   when B2 => delay 0.0; S3;
end select;
```

In these examples, statements S1 and S2 can only be chosen if the boolean expression B1 is true, and S3 can only be chosen if B2 is true. The choice is nondeterministic, yet restricted by the private control.

### 2.3 *CONSENSUS CONTROL*

Consensus control: nondeterminism restricted by environmental/communication constraints, which we classify as consensus. The choices are restricted to only those for which the communication is available from the rest of the system. Consensus control is considered established if the **rendezvous** can be established. Example 2a illustrates consensus control in CSP:

Example 2a

```
[
 [] P1?x   ==>   S1
 [] P1!y   ==>   S2
 [] P2?x   ==>   S3
]
```

Example 2b -
illustrates consensus control in Concurrent C:

```
select {
        accept entry1(x); S1;
        or
        accept entry2(y); S2;
        or
        accept entry3(z); S3;
        }
```

Example 2c -
illustrates consensus control in Ada:

```
select
    accept entry1(x:type_x) do S1;
    or
    accept entry2(y:type_y) do S2;
    or
    accept entry3(z:type_z) do S3;
end select;
```

In example 2a, S1 can only be chosen if process P1 is sending a value for x; S2 can only be chosen if process P1 is ready to receive a value y; and S3 can only be chosen if P2 is sending a value for x. In examples 2b and 2c, S1 will only be executed if another task is calling the entry1 and sending the value x. Likewise, S2 will only be executed if another task is calling for entry2 and sending a value of type y. The choice is made nondeterministically, yet it is restricted by the consensus control.

### 2.4 *HYBRID CONTROL*

Hybrid control: nondeterminism restricted by both boolean and environmental constraints ( private control and consensus control ), which we classify as hybrid control. Hybrid control is available if private control is open and consensus control is established.

Example 3a -
illustrates hybrid control in CSP:

```
[
 [] B1, P1?x  ==> S1
 [] B2, P1!y  ==> S2
 [] B2, P1?z  ==> S3
 [] B1, P2!z  ==> S4
]
```

In example 3a: S1 can only be chosen if B1 is true and P1 is sending a value for x; S2 can only be chosen if B2 is true and P1 is ready to receive a value y; S3 can only be chosen if B2 is true and P1 is sending a value for z; and S4 can only be chosen if B1 is true and P2 is ready to receive a value z. The choice is made nondeterministically, yet it is restricted by private control and consensus control.

Example 3b -
also illustrates hybrid control, in Ada :

```
select
 when B1 => accept entry1() ; S1; end entry1;
 or
 when B2 => accept entry2() ; S2; end entry2;
 or
 when B2 => accept entry1() ; S3; end entry1;
 or
 when B1 => accept entry2() ; S4; end entry2;
end select;
```

Example 3c -
illustrates hybrid control in Concurrent C:

```
select {
        B1 : accept entry1() ; S1;
        or
        B2 : accept entry2() ; S2;
        or
        B2 : accept entry1() ; S3;
        or
        B1 : accept entry2() ; S4;
        }
```

In examples 3b and 3c: S1 can be chosen if B1 is true and if entry1 is being called; S2 can be chosen if B2 and entry2 is being called; and so on... Again, the choice is made nondeterministically, yet controlled by private and consensus controls.

An alternative is ready if all of the controls have been satisfied; that is: if the private control is open and/or if the consensus control is established or if the hybrid control is available. The nondeterministic construct will check all of the controls in each alternative and then choose one of the ready alternatives. The programmer can include one, all or any combination of the controls allowed within the language′s nondeterministic structure to attain the desired expressive programming power.

## 3. *PREFERENCE CONTROL*

We will classify preference control as nondeterminism restricted by preferences; preference control gives the programmer the power to assign preferences to the choices within the nondeterministic construct. Each entry inside a nondeterministic construct may (but need not) have a preference. A lower value indicates a lower degree of urgency; the range of preferences is implementation defined. ( ie. a ready entry with a preference = 2 is chosen before a ready entry of preference = 1).

### 3.1 *OTHER EXISTING MECHANISMS FOR CONTROLLING NONDETERMINISM*:

Many languages contain primitives for the nondeterministic construct with private control, consensus control and/or hybrid control; but none contain the primitive for what we will classify as "preference control." The "by" construct was introduced by Andrews [2] and Concurrent C [7] later implemented the "by" construct and the "such that" construct. These constructs allow the entry to selectively choose an entry call from the entry queue instead of receiving the entry calls from the queue in FIFO order, which is the default. The "by" and the "such that" constructs differ from preference control in that they control choices from within an entry′s queue; "pref" controls the choice of alternatives within the select statement. The "cell" concept [14] allows explicit preference control amongst the entries within the select statement by attaching labels to entries and ordering the labels statically. Some languages can use other primitives to simulate preference control, but we will argue that these implementations are very complicated and unacceptable by software engineering standards. Preferences should allow some choices, within a nondeterministic construct, to be chosen before others. Concurrent C and Ada have implicitly defined preferences built into the language; in Concurrent C, an available accept alternative is chosen before an open immediate alternative. None of these languages contain a primitive to control preferences explicitly; this led us to the implementation of the "pref" primitive, which is necessary for preference control. Take for example, the resource manager that continuously releases and regains resources. Since the manager must have a resource in order to release it, a wise manager should give preference to the regaining of resources. The availability of the resource can be maintained by private control; the requests to release or return a resource can be checked by consensus control; but no primitive exists to express preference control, which is necessary in this case to give preference to the release of a resource. Therefore, we are forced to use other primitives to implement preference control, which we feel should be included as a primitive. These implementations require some programming effort and result in implementations that do not support software engineering principles. Some programming languages implement priorities; priorities will be discussed in a later section, since they are not part of what we have classified as controls for nondeterminism within unique tasks. Preference control can also be used to implement "fairness" and avoid process/task starvation; this will also be covered in section 4.1.

### 3.2 *IMPLEMENTATION OF PREFERENCES WITH EXISTING PRIMITIVES*

Suppose we have a process/task that is required to do one of four services ( service1, service2, service3 and service4), and that we wish to give the services preferences - that is, we wish to service4 if possible, else service3, else service2 and the least preference is to service1. Services 1 - 4 are all different. This cannot be implemented in CSP within the nondeterministic construct. This can be simulated in Ada in a couple of ways: (You might try to come up with some solutions yourself before reading the following known solutions)

1. Using low level system defined queues

Example 4

```
task CONTROLLER is
   entry SERVICE4(D:DATA);   -- very urgent
   entry SERVICE3(D:DATA);   -- urgent
   entry SERVICE2(D:DATA);   -- medium urgency
   entry SERVICE1(D:DATA);   -- low urgency
end CONTROLLER;


task body CONTROLLER is
begin
  select
    when B4 => accept SERVICE4(D:DATA) do
          ACTION4(D);
    end SERVICE4;
    or
    when B3 and (not(B4)
              or SERVICE4′COUNT = 0) =>
      accept SERVICE3(D:DATA) do
            ACTION3(D);
     end SERVICE3;
     or
     when B2 and (not(B4) or SERVICE4′COUNT = 0)
           and (not(B3) or SERVICE3′COUNT = 0)  =>
           accept SERVICE2(D:DATA) do
                 ACTION2(D);
            end SERVICE2;
     or
     when B1 and (not (B4) or SERVICE4′COUNT = 0)
         and (not(B3) or SERVICE3′COUNT = 0) and
         (not(B2) or SERVICE2′COUNT = 0) =>
            accept SERVICE1(D:DATA) do
                 ACTION1(D);
     end SERVICE1;
 end select;
end CONTROLLER;
```

1. This implementation is not very readable; it is not clear why the queues for service4, service3 and service2 are being checked if service1 is to be accepted.

2. All of those checks on the queues are expensive and make the implementation very complex.

3. Using the system defined low level queues forces the programmer to use low level commands, which ideally should be hidden from the user and are not desirable in a high level language like Ada.

4. This implementation requires a lot of modification if the user wishes to change the preferences or add entries with different preferences.

5. This implementation uses a "loop hole" in Ada, (ie. the COUNT attribute) to implement consensus control using private control. The use of the COUNT attribute as a condition is syntactically private control, and yet semantically consensus control. This inconsistency adds ambiguity to this problem.

6. This implementation is dangerous because it can cause the raising of an unnecessary exception. The semantics of the select statement specify that if all conditions ( the private control ) are false and there is no else alternative or terminate alternative included, then an exception should be raised. This rule was probably made assuming that the conditions covered by private control were static and could not change at a later time. But you can see by this example that if the boolean conditions are true, but the queues are empty, an unnecessary exception will be raised; although entry calls may be in progress. This faulty behavior can be caused by the inconsistent use of private control instead of consensus control ( as described in the latter paragraph).

2. Using nested select statements

Example 5

```
task CONTROLLER is
        entry SERVICE4(D:DATA);
        entry SERVICE3(D:DATA);
        entry SERVICE2(D:DATA);
        entry SERVICE1(D:DATA);
end;

task body CONTROLLER is
begin
Done:boolean = false;
while not(DONE) do
  loop
    select
      when not(Done) and B4 =>
      accept SERVICE4(D:DATA) do
           ACTION4(D); end SERVICE4; Done=true;

      else select
         when not(Done) and (not(B4)
              and B3) =>
              accept SERVICE3(D:DATA) do
                   ACTION3      (D); end SERVICE3; Done=true;

         else select
                    when not(Done) and (not(B4)
                    and not(B3) and B2) =>
                       accept SERVICE2(D:DATA) do
                            ACTION2(D); end SERVICE2; Done=true;

                    else select
                               when not(Done) and
                               (not(B4) and not(B3)
                               and not(B2) and B1) =>
                               accept SERVICE1(D:DATA) do
                                      ACTION1(D); end SERVICE1;
                                      Done=true;

                           else NULL;
                       end select;
                end select;
         end select;
    end select;
  end loop;
end CONTROLLER;
```

(NOTE: Could not use array/family of entries since all four entries are different)

1. This implementation is ambiguous; it is not clear why all of the nesting is necessary or how the nondeterminism will work in this case.

2. The nesting is very complex and will require further nesting if entries of different preferences were to be added!

3. This example requires an extra boolean variable (private control) and a loop (although the repetitive behavior of a loop is not desired) structure to make sure one and only one entry is accepted; this is extremely inefficient and confusing to a reader.

4. This example also implements "Busy Waiting", which is very expensive and inefficient.

5. This implementation is also very difficult to modify; changing the preferences of the entries would require renesting the entire select statement.

Both of these simulations are ambiguous and do not support software engineering standards. The need is then evident for a primitive to implement preference control within a nondeterministic construct.

### 3.3 *INTRODUCTION TO THE PREFERENCE CONTROL PRIMITIVE CONSTRUCT*:

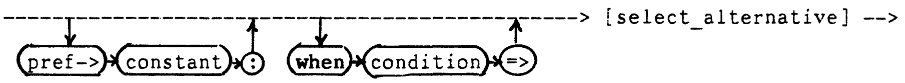
We propose the syntax: "**pref** -> <constant>" to assign a static preference control value within a nondeterministic construct. Dynamic preference control, in which we could pass "pref" the value of an expression, is much more powerful and expensive ( and out of the scope of this paper ). Static preference control is simple, easy to implement, and can be evaluated at compile-time. For the Ada nondeterministic construct, we suggest the following syntax for the select alternative:

Example 6

Syntax for the select alternative:

**pref** -> <constant>: **when** <condition> => select_alternative

SELECT ALTERNATIVE:



Example 7

```
selective_wait ::=
     select
        [pref -> constant : ]  [ when condition => ]
                select_alternative
       { or [pref -> constant : ]  [ when condition => ]
                select_alternative }
       [ else
          sequence_of_statements ]
      end select;


select_alternative ::=
    accept_statement [sequence_of_statements]
  | delay_statement  [sequence_of_statements]
  | terminate;
```

All of the nondeterminism constraints are listed before the entry call: first, the preference control (pref -> constant); followed by the private control (when <boolean expression>; and finally, the consensus control (=> accept <entry>). ( We considered placing the preference control declaration within the task entries declaration, but concluded it belonged within the select statement, with the other nondeterminism controls.) If a preference is not specified, it should default to the lowest value ( ie. if negative values for preferences are not allowed, then default to 0 ). The same value preference can be assigned to different entries within the same select statement to allow a greater amount of nondeterminism.

For example, using the nondeterministic construct "select" in Ada, combined with our proposed preference control construct, the problem described above can be solved by the following:

Example 8

```
task CONTROLLER is
   entry SERVICE4(D:DATA);        *** very urgent
   entry SERVICE3(D:DATA);        *** urgent
   entry SERVICE2(D:DATA);        *** medium urgency
   entry SERVICE1(D:DATA);        *** low urgency
end CONTROLLER;


task body CONTROLLER is
begin
  select
  pref -> 4: when B4 => accept SERVICE4(D:DATA) do
              ACTION4(D); end SERVICE4;
  or
  pref -> 3: when B3 => accept SERVICE3(D:DATA) do
              ACTION3(D); end SERVICE3;
  or
  pref -> 2: when B2 => accept SERVICE2(D:DATA) do
              ACTION2(D); end SERVICE2;
  or
  pref -> 1: when B1 => accept SERVICE1(D:DATA) do
              ACTION1(D); end SERVICE1;
  end select;
end CONTROLLER;
```

Semantics of preference control construct: Select one of the available alternatives with the highest pref value. More than one entry can have the same pref value; this will increase the nondeterminism of the choice.

1. This example is much more readable than the previous two and is much simpler and easier to understand.

2. Since nested select statements, loops and low level queues are not needed, this is also a more efficient implementation that supports software engineering standards.

3. Different entries could have the same preference and possibly differ in the private control or consensus control.

4. This construct adds expressive power to languages, like CSP, that cannot otherwise implement preferences.

5. A smart compiler could save some unnecessary evaluations by first evaluating the entries with the highest preferences.

6. This implementation is easy to modify; changes of priorities or the addition of entries would require minimal change of code.

7. This implementation does not contain busy waiting; nor does it include the danger of causing an exception handler to be raised unnecessarily because it makes a clear distinction between private control, consensus control and preference control.

The preference control primitive can directly be used to give an elegant solution to the resource manager problem by simply giving a higher preference to the releasing of resources. It should be noted that assigning preferences to the entries sequentially as they are ordered within the select statement is not a valid implementation of preference control, since the "pref" value of each entry would be unique; preference control requires that more than one entry can have the same "pref" value.

## 4. *EXTENSIONS & ISSUES OF PREFERENCE CONTROL*:

### 4.1 *FURTHER USE OF PREFERENCE CONTROL TO IMPLEMENT FAIRNESS*:

We propose the extension of the implementation of type tasks in Ada to allow specifying different preferences to the type tasks. A type task could be defined once (possibly with default preferences) and later, when these tasks are declared dynamically, the preferences could be listed as a parameter to the declaration. We could, for example, define :

Example 9

```
        task type CONTROLLERS( prefs -> a,b,c,d ) is
             entry entry1 ...
             entry entry2 ...
             entry entry3 ...
             entry entry4 ...
        end CONTROLLERS;

        task body CONTROLLERS( prefs -> a,b,c,d ) is
             :
             .
             select
                 pref -> a: ... accept entry1 ...
                 or
                 pref -> b: ... accept entry2 ...
                 or
                 pref -> c: ... accept entry3 ...
                 or
                 pref -> d: ... accept entry4 ...
             end select;
         end CONTROLLERS;
```

And then declare:

```
CONTROLLER1 : CONTROLLERS(1,2,3,4);
CONTROLLER2 : CONTROLLERS(4,3,2,1);
CONTROLLER3 : CONTROLLERS(2,3,3,1);
```

CONTROLLER1 is of type CONTROLLERS and was assigned a preference of 1 to entry1, a preference of 2 to entry2, a preference of 3 to entry3 and a preference of 4 to entry4. Oppositely, CONTROLLER2 was assigned a preference of 4 to entry1, a preference of 3 to entry2, a preference of 2 to entry3 and a preference of 1 to entry4. CONTROLLER3 illustrates that more than one entry can have the same pref value. Type tasks with preference control parameterizing would allow the programmer to assign different preferences to different instances of the same task without having to rewrite a copy of the task. The use of preferences can help avoid starvation of processes or tasks and could then implement fairness. Concurrent Prolog [13] implements fairness using the "stable machine", but the stable machine does not appear to implement nondeterminism as needed in our examples.

### 4.2 *PRIORITIES VERSUS PREFERENCE CONTROL*:

We distinguish between preference control and priority control. Priority refers to processes or actions to be chosen at the system level; preferences are defined at the user level. Priorities cannot be considered to be included in our classification of controls restricting nondeterminism within a task. Priority is a function from a set of tasks to a set of values; whereas, preference is a function from a set of entries within a task to a set of values. Priority controls the race among different tasks in the system level. Preferences control the race among entry calls within a task at the user level. Giving a process or task a priority would not solve the example above, since the same process can call the manager for the request or release a resource. Preferences give control to the server, whereas priorities give priority to the clients. Priority control is implemented in both Ada and Concurrent C.

In Ada, for example, the user can explicitly define the priority of a task using the "PRIORITY pragma", which is a suggestion/pragma to the compiler.

"Each task may be assigned a priority that overrides the default priority assigned to a task by the implementation. Tasks can be assigned a priority by using the PRIORITY pragma which is of the form: **pragma PRIORITY(P);** and is included in the specification of the task. P is a static expression of the implementation defined integer subtype PRIORITY. The higher the value of P, the higher the priority of the task."

Like Ada, Concurrent C implements process priority assignment. Concurrent C also implements a different version of priorities, using the "BY" construct, which allows the user to define/prioritize the tasks that could be served by an individual entry. The "BY" construct is considered a priority control because it controls the race among different tasks. Concurrent C controls this at the user level as an additional language within a nondeterministic construct, not at the system level like Ada does. The "BY" construct is not powerful enough to solve the resource manager problem, since we need to prefer one entry over other entries, not one task over another at the entry level.

Concurrent Prolog defines a stable machine as: " A Concurrent Prolog machine is ′stable′ if it always chooses the first unifiable clause for reduction, if several such clauses exist." Concurrent Prolog implements implicit preference control, but it does not do so within a nondeterministic construct. Therefore, we cannot classify Concurrent Prolog′s stable machine as a priority, nor as preference control, because it does not control a nondeterministic construct. Preferences are implicitly defined (and forced) by the ordering of the statements. Similar to preference control, the stable machine in Concurrent Prolog can implement fairness and decrease the chance of process/task starvation.

### 4.3 *NEW ISSUES RAISED BY PREFERENCE CONTROL*:

The additional expressive power of implementing preferences may bring about other difficulties such as: difficulty of implementation; and difficulty of use. Consider the following program in CSP with the addition of preferences:

Example 10

```
P:: P1 || P2

P1:: [
      [] pref->2, P2?x ==> S1;
      [] pref->1, P2!0 ==> S2;
     ]

P2:: [
      [] pref->2, P1?y ==> S3;
      [] pref->1, P1!0 ==> S4;
     ]
```

The above conflict arises when P1 & P2 are executed in parallel and are in essence "deadlocked"; this new type of error can be introduced by preferences. The programming methodologies of Communication Closed Layers [5] of constructing well-structured distributed programs may help avoid such errors, but cannot insure their discovery or elimination before run time. Ada and Concurrent C avoid this conflict by the asymetric use of the accept alternatives and the call alternatives within separate select alternatives.

These issues, and others that may arise from the combination of preference control and different language constructs, are out of the scope of this paper. We simply wish to introduce the concept of "preference control" restricting nondeterminism for concurrent programming languages. The actual implementation of preference control will have to vary for each individual programming language, depending on the language′s programming philosophies and methodologies.

## 5. *CONCLUSION*:

Nondeterminism is a central concept and issue in modern concurrent programming languages that exploits explicit parallelism. Controlling nondeterminism should therefore be an important issue of distributed programming language design. In this paper, we first classified the current available primitives that can be used for controlling nondeterminism : private control, consensus control and hybrid control. We then introduced and suggested the addition of a new language independent primitive to control nondeterminism, which we call preference control. Our examples have illustrated the expressive programming power of controlling nondeterminism with the above mentioned controls. We strongly suggest the use of preference control in real time applications, where it is urgently needed.

## REFERENCES

1. G.R.Andrews, F.B.Schneider, "Concepts and Notations for Concurrent Programming" 1983 ACM 0010-4892/83/0300-0003.

2. G.R.Andrews, "Synchronizing Resources," ACM Trans. Programming Languages Syst. Vol. 3, Oct. 1981.

3. G.Booch, "Software Engineering with Ada" The Benjamin/ Cummings Publishing Co.,Inc 1983.

4. T. Elrad, F. Maymir-Ducharme "Introducing the Preference Control Primitive: Experience with Controlling Nondeterminism in Ada", proceedings of the 1986 Washington Ada Symposium in Laurel, Maryland, March 24 - 26, 1986.

5. T. Elrad, N. Francez, "Decomposition of Distributed Programs into Communication Closed Layers" Science of Computer Programming 2, 1982.

6. N. Gehani, "Ada: Concurrent Programming", Prentice Hall, 1984.

7. N. Gehani, W. Roome "Concurrent C*" AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1985.

8. C.A.R. Hoare, "Communicating Sequential Processes" CACM 21,8, August 1978.

9. C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall International, 1985.

10. E. Horowitz, "Fundamentals of Programming Languages," Computer Science Press, 1984.

11. J.D. Icbiah, et al, Reference Manual for the Ada Programming Language January 1983.

12. B.Liskov, "Primitives for Distributed Programming" Computation Structures Group Memo 175 MIT Laboratory for Computer Science (May 1979).

13. E. Shapiro, "Programming in Concurrent Prolog - Lecture Notes" The Weizmann Institute of Science, July 1985.

14. A. Silberschatz, "Cell: A Distributed Computing Modularization Concept", IEEE Trans. Softw. Eng., Vol. SE-10, No. 2, March 1984.

15. S.A.Smolka, P.Wegner, "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives" IEEE Transactions Software Engineering 1983.

16. P.D.Stoutts, "A Comparative Survey of Concurrent Programming Languages" ACM SIGPLAN Notices.

# REFINED FORTRAN:
# ANOTHER SEQUENTIAL LANGUAGE FOR PARALLEL PROGRAMMING

*Henry Dietz and David Klappholz*

Center for Distributed Processing
Stevens Institute of Technology
Castle Point
Hoboken, NJ 07030

**Abstract** — This paper presents the application of the refined-language methodology to **ANSI FORTRAN 77**. The resulting language, **RF77**, permits:

- Users to write code which differs from standard **ANSI FORTRAN 77** code only in that *data access* rights are more precisely specified
- Compilers, using well-known flow-analysis techniques, to generate consistently good, highly-parallel, race-free, code for virtually any machine architecture.

In modifying **ANSI FORTRAN 77**, our goal is not merely to find parallelism where none was envisioned by the programmer, but to provide a more general way of expressing algorithms for parallel computers of any type (MIMD, VLIW, SIMD, etc.).

## Introduction

In our paper, "Refined C: A Sequential Language For Parallel Programming," which appeared in the proceedings of ICPP 1985 [Die 85], we presented both a definition of the language **RC**[1] and a discussion of what the refined-language methodology is. We also claimed that the methodology could be applied to other languages — and everyone asked "Can you do it with **FORTRAN**?"

In this paper, we present both a review of the problems encountered when flow analysis is applied to unmodified **ANSI FORTRAN 77** (toward concurrency-detection), and the specification of **RF77**: *refined* **ANSI FORTRAN 77**.

As might be expected, the substantially different natures of **C** and **FORTRAN** are reflected in significantly different-looking refinements being made to **FORTRAN** as compared to **C**. However, the methodology remains intact.

## The Methodology

The *refined*-language approach begins with a conventional HLL — nearly any HLL: in this paper, **ANSI FORTRAN 77**. We have chosen this particular dialect of **FORTRAN** because it is a superset of most dialects in popular use, yet its specification is readily available. Because the **ANSI FORTRAN 8x** specification includes vector operations, it might seem, at first, that we are attempting to "re-invent the wheel" — but vector operations do not make generation of good MIMD code significantly easier [Die 86]. In addition, the refinements made to **ANSI FORTRAN 77** are compatible with the vector notation of **ANSI FORTRAN 8x**; hence, **RF77** can be easily transformed into **RF8x**.

Since **ANSI FORTRAN 77** contains no explicit parallelism or synchronization constructs, it is naturally impossible to write a race condition in the language. Likewise, a flow-analyzing compiler re-structuring code into a parallel form by using only correctness-preserving transformations will be incapable of generating a race condition. Therefore, if such a compiler is given pure **ANSI FORTRAN 77** code, the programmer is guaranteed that the parallelized program will produce the same result as the sequential program — the program will be debuggable. Perhaps even more important, using a compiler with a "back-end" appropriate for each parallel or sequential machine, the program will be completely portable.

Unfortunately, the amount of useful parallelism found by a flow-analyzing compiler examining a typical (unrefined) **ANSI FORTRAN 77** program is not necessarily all (or even a large fraction of all) that is present in the program

[1] The syntax of **RC** has been slightly altered to better conform to the draft specification of **ANSI C**. A revised definition of **RC** is available from the authors upon request.

0190-3918/86/0000/0184 $01.00 © 1986 IEEE

[Kuc 72]. This discrepancy is caused by certain language constructs obscuring (from the compiler's flow analysis) the fact that some operations could be parallelized. In refining **ANSI FORTRAN 77**, constructs which obscure the needed information are removed from the language and are replaced by modified versions which do not inhibit the analysis. These replacements can be made to look much like the original constructs and can provide all, or nearly all, of their expressive power. All the other language constructs remain as they were.

The resulting language, **RF77**, looks and "feels" like **ANSI FORTRAN 77**, but, unlike the latter, can be compiled into *reasonably efficient* race-free code for any kind of machine, parallel or sequential (assuming a compiler has been constructed for the machine in question).

## What's Wrong With FORTRAN?

For the purpose of automatically generating parallel code, it is convenient that typical (unrefined) **FORTRAN** programs are far simpler and more static than programs written in most other languages. Programs are simpler because **FORTRAN** is a relatively spartan language — there are not as many different ways to say the same thing as in most other languages. Programs are more static in that more information than usual about the run-time behavior of a program can be determined at compile-time. For example, the amount of data space required by a typical **FORTRAN** program is known at compile time; in most languages, the ability to perform recursive calls and to dynamically request chunks of memory makes determining the run-time size equivalent to solving the halting problem.

In these respects, **FORTRAN**, in any of its major dialects, is an ideal language for a compiler to analyze. This fact is evidenced by the multitude of parallelizing compilers for *dusty deck* **FORTRAN** [All 82] [Fis] [Kuc 84] [Nic 85] [KAI 85] and the lack of parallelizing compilers for almost any other language.

Although the simple, static, nature of **FORTRAN** programs aids the compiler in its analysis, there are **FORTRAN** constructs which severly impede analysis for automatic parallelism-detection. Any construct which blurs the compiler's picture of which data items might be accessed by any particular reference will result in the compiler making a "safe" assumption. For example, in:

```
C      For this and the following examples, assume
C      that labels 10 and 20 are never referenced;
C      they appear only to relate each example to
C      the discussion in the text.
 10    A =B * C
 20    D =E * F
```

it is obvious that the statements labeled 10 and 20 can be executed either sequentially or in parallel, but if and only if the answers to the following questions are "No":

(1) Is B or C an alias for D? Is E or F an alias for A?

(2) Are A and D aliases for each other?

If any part of (1) was answered "Yes," executing statements 10 and 20 in parallel could produce an incorrect result — a write/read race condition would exist. If (2) was answered "Yes," executing statements 10 and 20 in parallel would produce unreliable results — a write/write race condition would exist.[2] If static analysis could not answer either (1) or (2), or if either answer was "Sometimes Yes," the only "safe" assumption is that parallel code for statements 10 and 20 cannot be generated.

The previous example is somewhat contrived, because it is (usually) trivial for a flow-analyzing compiler to determine the answers to each of the above questions — any of the "dusty-deck" parallelizing compilers mentioned above could answer them. However, we will use minor variations on this example to demonstrate the analysis problems caused by each of the **ANSI FORTRAN 77** constructs which must be *refined.*

## References to Global Data (COMMONs)

Suppose that A, B, and C are all defined as independent variables which reside in a **COMMON** and that the statement labeled 20 is changed so that we now have:

```
10    A =B * C
20    CALL SUBR
```

[2] It is interesting to note that, in general, if an answer is known to be "Yes" then a sequential code optimization is possible. For example, if A and D are aliases for each other, statement 10 is a dead computation and can be eliminated.

Also assume that SUBR is a **SUBROUTINE** which is defined in another file. The questions the compiler must answer are essentially the same:

(1) Does SUBR (or any subprogram invoked by SUBR) contain any stores into B or C or any loads of A?

(2) Does SUBR (or any subprogram invoked by SUBR) contain any stores into A?

In order for the compiler to generate code which can safely execute statements 10 and 20 in parallel, the answers to both questions must be known to be "No."

The time complexity of some flow-analysis techniques used to attempt to answer these questions forces flow-analysis to be localized to small regions of a program (for example, a **SUBROUTINE** or **FUNCTION** at a time) — analysis of larger regions would take an unacceptably long time. Since the above example may require this analysis to be performed on the entire program,[3] the compiler would probably be unable to answer these questions. This, in turn, would force the compiler to make the "safe" assumption that every **SUBROUTINE** or **FUNCTION** call might affect every variable that appears in any **COMMON**. Sequential code results.

A less obvious, but very similar, situation exists relative to the use of I/O channels by subprograms. The I/O channel numbers act like global variables. Consider:

```
10   WRITE (6,*) A
20   CALL SUBR
```

It is impossible to tell if SUBR uses I/O channel 6 without looking, at the very least, at the code for SUBR.

## References using Pointers (Call-by-Address)

Let us now assume that A, B, and C are all defined as independent variables local to the **SUBROUTINE** in which the following code appears:

```
10   A =B * C
20   CALL SUBR(A, B)
```

As before, SUBR is assumed to be defined in another file; but, since we have already considered the problem, we will assume that there are no **COMMONs** in SUBR. Although **FORTRAN** does not explicitly support *pointers*, it does use call-by-address in passing arguments to **FUNCTIONs** and **SUBROUTINEs**. The compiler must now find answers to:

(1) Does SUBR (or any subprogram invoked by SUBR) store into B or load from A?

(2) Does SUBR (or any subprogram invoked by SUBR) store into A?

which, of course, would normally be too expensive for the compiler to answer.

## References to Indexed Data Structures

Let us return to our original example, again, slightly modified:

```
     IMPLICIT INTEGER A-Z
     DIMENSION G(100)
10   G(A) = G(B) * G(C)
20   G(D) = G(E) * G(F)
```

Again, we must answer nearly the same questions as for the original, trivial, example:

(1) Is G(B) or G(C) the same element as G(D)? Is G(E) or G(F) the same element as G(A)?

(2) Is G(A) the same element as G(D)?

However, these questions are much harder to answer. In fact, they cannot be answered at compile time if any of A, B, C, D, E, and F is given a value at run-time which is not related to all the other variables' values in an obvious way. For example, suppose a value is **READ** for A in the statement which is executed just before 10. It is not merely difficult to find answers to these questions by analysis; it is theoretically impossible in such a case. In other cases, answering these questions requires theorem proving.

If the value of any of A, B, C, D, E, and F is affected by a parameter entering the **SUBROUTINE** which contains the above example, then the corresponding argument to *every* **CALL** of that **SUBROUTINE** must be examined. This may be theoretically possible, but it certainly is not practical.

Since it can be very difficult to determine which element(s) of a data structure an indexed reference can access, it is often necessary to

[3] Recently, substantial advances have been made toward limiting the scope of analysis by constructing "programming environments" which incrementally collect the needed information. A good example of this is [Coo 85]. However, this mechanism alone is not capable of solving the problems described in section on **References to Indexed Data Structures.**

assume that such a reference potentially affects any (every) element of the array.

Several attempts have been made to resolve these analysis problems by having the programmer insert assertions. Some of these assertions have been of the form "Trust Me . . . it's safe to do this in parallel," which is very thinly-disguised explicit parallelism — with all the disadvantages thereof. The best approach to adding assertions is described in [Fis] and [Fis 84]. These assertions take the form of equations which can actually be checked for validity at run-time, but solving the equations is a complex task for the compiler and the assertions themselves are quite alien to the average **FORTRAN** programmer.

The **RF77** equivalent to assertions (the concept of *partitioning*) is both efficient and safe, but, most importantly, it makes sense in terms of expressing an algorithm.

### The Refinements

In each of the situations described above, the compiler's inability to resolve exactly which data items might be accessed by a particular reference must result in the "safe" assumption — that all possibly touched items are not "available" across such a reference — which typically forces the generation of sequential code. If we can enable the programmer to specify what really happens in these cases, more computations can be known to the compiler to be available, fewer precedence constraints need be artificially imposed by the compiler, and less of the generated code will be forced to be sequential.

Each refinement can be viewed as simply providing the programmer with a language construct which, while being intuitive and natural to the programmer, allows him to provide exactly the extra information the compiler needs.

### Access Permissions for Globals

As pointed-out in the discussion above, **FORTRAN** supports two basic kinds of global data:

- **COMMON**s, which are used to group sets of variables together by name and to give **FUNCTION**s and **SUBROUTINE**s access to the variables by these group (**COMMON**) names.
- I/O channel (logical unit, logical file) numbers, each of which is used to give the set of data within a file a name (number) and to give **FUNCTION**s and **SUBROUTINE**s access to the variables by that name.

the two are very similar in concept.

The kind of language structure needed to solve the problems associated with global data references closely resembles a **COMMON**. Unfortunately, although **COMMON** statements provide the naming features needed, they are scattered throughout the source program and potentially across a large number of files. To make compilation speed acceptable, the information must be available without having to scan the entire source program.

The required information about use of globals in an **RF77** program is placed in a separate *interface specification* file,[4] which is **#INCLUDE**d by all files that constitute the source program — much as **C** programs **#include** "header files." The *interface specification* file simply contains the definitions of global variables and the access permission each **FUNCTION** and **SUBROUTINE** has for each global.

Borrowing the terminology of Ada [Ada 80], there are two primitive kinds of access permission relevant in performing concurrency detection and generation of efficient parallel code:

**IN** For a variable, permission for the variable's *rvalue* to flow into the **FUNCTION** or **SUBROUTINE**; for a file, permission only to **READ** from the file.

**OUT** For a variable, permission for the *rvalue* of a variable to be different as the variable flows out of the **FUNCTION** or **SUBROUTINE** from what it was at entry; for a file, permission to **WRITE** to the file.

Also, as in Ada, these access permissions may be combined:

**IN OUT** For a variable, permission for the variable's *rvalue* to both flow into the **FUNCTION** or **SUBROUTINE** and to be different as the variable flows out; for a file, permission both to **READ** from and to **WRITE** to the file.

COMMON Permissions. Each individual entry within an **RF77** *interface specification* takes one of the following forms:

[4] A software tool currently under development, called **PREFINE**, will automatically convert a **FORTRAN** program into its **RF77** equivalent — including the creation of an appropriate *interface specification* file.

```
IN /global_name/ subprogram_list
OUT /global_name/ subprogram_list
IN OUT /global_name/ subprogram_list
```

where IN and OUT are as indicated above and IN OUT is equivalent to *both* IN and OUT permission. Note that *global_name* may be either the name of a COMMON or it can be blank, representing the unnamed COMMON.

The following is a skeletal example of the use of IN, OUT, and IN OUT:

```
C      this would be the interface spec., "TEST.H"
       IN /A/ SUBR1,FUNC2
C      SUBR1 and FUNC2 both can examine any var in A
       OUT /A/ SUBR2
C      SUBR2 can change any var in COMMON A
       IN OUT /A/ FUNC1
C      FUNC1 can examine and change any item in A

C      the following appears in TEST1.RF
#INCLUDE TEST.H
       FUNCTION FUNC1(. . .)
       COMMON /A/ X,Y,Z
       . . .
       END

C      the following appears in TEST2.RF
#INCLUDE TEST.H
       FUNCTION FUNC2(. . .)
       COMMON /A/ X,Y,Z
       . . .
       END

C      the following appears in TEST3.RF
#INCLUDE TEST.H
       SUBROUTINE SUBR1(. . .)
       COMMON /A/ X,Y,Z
       . . .
       END

C      the following appears in TEST4.RF
#INCLUDE TEST.H
       SUBROUTINE SUBR2(. . .)
       COMMON /A/ X,Y,Z
       . . .
       END
```

It is important to note that the RF77 compiler will flag any attempt to reference a global for which permission was not explicitly or implicitly granted: if any subprogram attempts to exceed the access rights granted by the *interface specification*, a fatal compile-time error will result. By the *interface specification* given in the previous example, the following definition of SUBR1 is an error because only IN rights were granted for members of the COMMON A:

```
SUBROUTINE SUBR1(B)
COMMON /A/ X,Y,Z
X = 5.0
END
```

By the same principle, any attempt to CALL a SUBROUTINE or FUNCTION which has access privileges beyond those of the caller also constitutes a fatal compile-time error:

```
SUBROUTINE SUBR1(B)
COMMON /A/ X,Y,Z
CALL SUBR2
END
```

I/O Channel Permissions. As we pointed-out at the start of this section, I/O channels are also a form of global, named by the INTEGER channel number (which is considered to be a *pre-defined* COMMON name). For example, the ability to READ from I/O channel 7 would be granted to subroutine SUBR3 by:

```
IN   /7/   SUBR3
```

The parallelization of I/O operations is traditionally one of the most difficult and unreliable language features, as evidenced by the lack of I/O operations in many new parallel-processing languages. In creating RF77, this problem is even more difficult because much of the flavor of FORTRAN is its style of I/O — drastically changing the I/O would have made RF77 drastically different from FORTRAN. We have chosen to maintain the FORTRAN I/O style, at a slight cost in reliability of parallelization.

To function properly, I/O operations would have to be based on naming *files* — but FORTRAN, and hence RF77, I/O is based on naming *channels.* RF77 *compilers assume that operations performed using different I/O channels are always independent of each other.* Strictly speaking, this is not always true — a single file may be associated with several I/O channels: if one is WRITEing, the compiler might accidentally create a race condition by assuming that operations on two different channels can proceed simultaneously. In fact, this can also cause unpredictable results on some single-processor machines, due to I/O buffering problems.

### Argument Passing & Parameter Definition

Each FORTRAN FUNCTION or SUBROUTINE is able to accept any number of call-by-address arguments. Since call-by-address is used, each argument passed to a subprogram could be carrying **IN**, **OUT**, or **IN OUT**, permissions to the *rvalue*: for the same reasons that a parallelism-detecting compiler must know the access permissions that subprograms have to **COMMON**s and I/O channels, the compiler must know which of these permissions each argument carries. As with global information, the same specification must be available to the compiler during compilation of both the **CALL**er and the **CALL**ed subprogram. The **CALL**ed routine cannot require access privileges not granted by the **CALL**er.

The access privileges granted by the caller for each argument should generally match those rights required by the called subprogram of its parameters.[5] By placing this information in the *interface specification*, it need be given only once for each **FUNCTION** or **SUBROUTINE**. **RF77** uses the following syntax to state which access rights are carried by each argument of a subprogram:

```
function_spec ::= ARGUMENT FORTRAN_type arg_spec
subroutine_spec ::= ARGUMENT arg_spec
arg_spec ::= sub_name ( a_p_list )
a_p_list ::= a_p_list , perm type_size
        | perm type_size
perm ::= IN | OUT | IN OUT | null_string
type_size ::= FORTRAN_type ( dim_list )
        | FORTRAN_type
        | null_string
```

where *FORTRAN_type* is any data type available in **ANSI FORTRAN 77**.

Normally, **ARGUMENT** specifications will state the access permission carried by each argument; however, if the access permission carried by an argument is not specified then the permission is assumed to be **IN**. The optional *type_size* specifications, if given, allow the **RF77** translator to perform type checking on arguments — a feature not related to parallel execution, but desirable for other reasons.

As an example of access permission specification, a subroutine to add its first two arguments and store the result in the third might be written as:

```
C       this would be the interface spec., "TEST2.H"
        ARGUMENT ADDSUB(IN, IN, OUT)

C       the following appears in TEST2.RF
#INCLUDE TEST2.H
        SUBROUTINE ADDSUB(A, B, C)
        C = A + B
        RETURN
        END
```

And ADDSUB would be called as, for example:

```
CALL ADDSUB(5, D, E)
```

A more concrete (but still artificial) example of the use of these constructs is the following matrix multiplication code:

```
C       this would be the interface spec., "MATMUL.H"
C       MATMUL and DOTPRO examine COMMON AB
        IN /AB/ MATMUL, DOTPRO
C       DOTPRO can examine,examine,&change its args
        ARGUMENT DOTPRO(IN,IN,OUT)
C       MATMUL can change its arg
        ARGUMENT MATMUL(OUT)

C       the following appears in some other file
#INCLUDE MATMUL.H
        SUBROUTINE MATMUL(C)
        REAL C(100,100)
        COMMON /AB/ A(100,100),B(100,100)
        DO 10 I=1,100
            DO 20 J=1,100
                CALL DOTPRO(I, J, C(I,J))
 20         CONTINUE
 10     CONTINUE
        STOP
        END

C       the following appears in yet another file
#INCLUDE MATMUL.H
        SUBROUTINE DOTPRO(I, J, C)
        COMMON /AB/ A(100,100),B(100,100)
        SUM = 0.0
        DO 10 K=1,100
            SUM = SUM + A(I,K) * B(K,J)
 10     CONTINUE
        C = SUM
        RETURN
        END
```

The constructs presented in this and the previous section enable the **RF77** compiler to determine which subprograms can be executed in parallel with one another *without* requiring expensive global analysis. The constructs presented in the next section permit the programmer to express information beyond that which can be expressed in a "dusty-deck" language, hence greatly increasing parallelism.

[5] In fact, these permissions often do not match. An individual call might pass more restricted rights than the subprogram normally requires, but that are known to be sufficient for that call. Capitalizing on this was deemed too risky.

## Indexing and Partitioning

Partitioning is the technique used by refined languages to create new names for arbitrary portions of a data structure. Once these names exist, it is a trivial matter to independently control the access permissions of each piece (called a *partition-element*).

C-Style Partitions. Partitioning can be viewed as a way of more-precisely specifying what pointers have permission to point at, which may imply a conceptual rearrangement of the object data structure such that elements which are not accessible appear to be removed from the structure. This the primary function of partitions in languages like C [Die 84] [Die 85].

Partitioning a matrix by an arbitrary formula specifying which partition-element each datum belongs to can produce partition-elements which *are not the same size or shape as the original array — perhaps not even rectangular.* For example, each "row" might be of a different length. This style of partitioning is what we call partitioning with *dynamic indexing*; that is, the shape and indexing structure may change dynamically according to the partitioning specification.

FORTRAN Partitions. FORTRAN doesn't support pointers. FORTRAN programmers don't want to think in terms of non-rectangular structures because conventional FORTRAN doesn't provide any way of building such structures.

In RF77, the shape and indexing structure of the *original data structure* must be preserved independently of the partitioning specification — we call this partitioning with *static indexing.*

Static indexing means that, for example, when a matrix is partitioned into partition-elements above and below the diagonal, each partition-element has the same indexing formulas and shape as the original matrix, but some of the data of the original matrix are inaccessible by each partition-element name: a reference to a datum by indexing the "above" partition-element, where the datum is conceptually contained in the "below" partition-element, is an error. In effect, static indexing means that partitioning is a zero-cost operation and each partition-element has a *membership test formula* associated with it.[6]

Partitioning is accomplished by the (apparently executable) PARTITION statement:

```
partstat ::= PARTITION ( structure , partlist part )
partlist ::=part ( condition ) , partlist
         | null_string
```

where *structure* is the name of the structure being partitioned (including dummy variables naming each subscript) and *part* is a name for a partition-element.

The *conditions* within a PARTITION statement are evaluated left-to-right *on only those data which have not yet been placed in a partition-element*, therefore, all partition-elements are guaranteed to be mutually exclusive: each datum can belong to only one at a time.

The following code illustrates the definition and use of a PARTITION which creates partition-elements naming the portions of the array A which are, respectively, above, on, and below, the diagonal:

```
      REAL A(100,100)
      . . .
      PARTITION(A(I,J), AUPPER(J .GT. I),
     1     ADIAG(I .EQ. J), ALOWER)
C     the following reference is valid
      AUPPER(5,7) =3.14159265
C     the following reference is not valid
C     and would cause a fatal compile-time error
      ALOWER(5,7) =3.14159265
```

The user may explicitly test whether a particular datum is a member of a partition-element by using the unary prefix operator .MEMBER., which simply applies the membership test formula for the following subscripted reference and returns a logical value of .TRUE. if that reference is valid. Thus:

```
C     the condition below is obviously .TRUE.
      IF (.MEMBER. AUPPER(1,99)) GOTO 30
C     this might or might not be .TRUE.
      IF (.MEMBER. AUPPER(4*K-1,L)) GOTO 30
C     the condition below is obviously .FALSE.
      IF (.MEMBER. AUPPER(I,I)) GOTO 30
```

## Compilation Techniques

Fundamentally, compilation of RF77 into efficient parallel code *is identical to* compilation of FORTRAN into efficient parallel code. Any

[6] The concept of membership also applies to other data structures. For example, an array membership test formula simply checks that subscripts are within bounds.

of the techniques of [All 82] [Fis] [Kuc 84] [Nic 85] [KAI 85] could be used and the result would be at least as good as — probably much better than — that obtained from "pure" FORTRAN.

Of course, just as we have our own parallel computer design [Par 86] [Die 85/2] in mind for the execution of these programs, we also have our own way of constructing *refined*-language compilers. The most unusual features of our approach are:

- Use of conventional (sequential) optimizing compiler flow-analysis concepts and techniques, as per [Coc 70] and [Aho 79], to build an *access-flow* graph.
- Use of non-deterministic *process-packaging*. The kind of parallel code generated is determined by a pruned search for target-machine-dependent structures within the *access-flow* graph. The basic technique is similar to that used by [Kru 82] for sequential code generation.

## Conclusions

In this paper, we have given a detailed presentation of the application of the *refined*-language methodology to ANSI FORTRAN 77, and we have given a reasonably precise definition of the resulting language, RF77, including many small examples.

Throughout our modifications, we have not changed the FORTRAN flavor of the language nor have we imposed any particular view of parallel processing. We have, however, built the language in such way that it can be easily compiled into efficient code for nearly any kind of parallel computer. Further, since these refinements aid *any* compiler in building a more accurate flow-graph, RF77 is completely compatible with, and efficiently usable by, compilers for single-processor machines. In fact, by applying conventional optimization techniques to the better quality flow graph, RF77 may actually be a more efficient language for SISD machines than ANSI FORTRAN 77.

## References

[Ada 80] Military Standard, *Ada Programming Language*, MIL-STD-1815 (the green book), Dec. 1980.

[All 82] J. R. Allen and K. Kennedy, "PFC: a Program to Convert Fortran to Parallel Form," Report MASC TR 82-6, Dept. of Math. Sciences, Rice University, Houston, TX, Mar. 1982.

[Aho 79] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Reading Massachusetts; Addison-Wesley, pages 262-265, Apr. 1979.

[Coc 70] John Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, New York; New York University Courrant Institute, pp. 320-334, 1970.

[Coo 85] Keith D. Cooper, Ken Kennedy, and Linda Torczon, "The Impact of Interprocedural Analysis and Optimization on the Design of a Software Development Environment," ACM 0-89791-165-2/85/006/0107, 1985.

[Die 84] Henry Dietz and David Klappholz, "Refining A Conventional Language For Race-Free Specification Of Parallel Algorithms," IEEE Proceedings of ICPP 1984.

[Die 85] Henry Dietz and David Klappholz, "Refined C: A Sequential Language For Parallel Programming," IEEE Proceedings of ICPP 1985.

[Die 85/2] Henry Dietz and David Klappholz, "RISC CPU Design for MIMDs," presented at The Second SIAM Conference on Parallel Processing for Scientific Computing, Nov. 1985.

[Die 86] Henry Dietz, "The Case for Sequential Languages for Parallel Machines — The Myth of Machine Independence," document in preparation.

[Fis] Joseph A. Fisher, "Parallel Processing: A Smart Compiler and a Dumb Machine," Yale University.

[Fis 84] Joseph A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," IEEE Computer, pp. 45-53, July 1984.

[KAI 85] KAI, *Mini-KAP/AF*, Kuck and Associates Inc., Professional Commerce Center, 1808 Woodfield Dr., Savoy, IL 61874. Aug. 1985.

[Kru 82] D. W. Krumme and D. H. Ackley, "A Practical Method for Code Generation Based on Exhaustive Search," Proc. of the SIGPLAN '82 Symposium on Compiler Construction, pages 185-196, June 1982.

[Kuc 72] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," IEEE Trans. on Computers, Vol. C-21, No. 12, pages 1293-1310, Dec. 1972.

[Kuc 84] David J. Kuck et al, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," IEEE Proc. of ICPP 1984, pages 129-138, Aug. 1984.

[Nic 85] Alexandru Nicolau, "Uniform Parallelism Exploitation In Ordinary Programs," IEEE Pro. of ICPP 1985, pages 614-618, Aug. 1985.

[Par 86] H-C Park and David Klappholz, "Single-Stage MIMD Design With Smart Nodes," document in preparation.

# A VLSI COMPARISON OF SWITCH-RECURSIVE BANYAN AND CROSSBAR INTERCONNECTION NETWORKS *

*T.H. Szymanski*

Department of Electrical Engineering
University of Toronto
Toronto, Ontario, Canada

Abstract – A class of multistage interconnection networks (MINs) that minimizes the number of wire crossovers is presented. The basic idea is to use a MIN (made with smaller MINs) as the basic switch in the realization of a larger MIN. The resulting *switch-recursive* MIN has far fewer wire crossovers when compared to others MINs, such as the SW-Banyan, Baseline, Delta and Omega networks. More importantly, switch-recursive MINs have very distinct clusters in the topology, with heavy communications within clusters, and little communications between clusters. In a discrete component realization, such clustering makes the partitioning of these networks onto printed circuit boards, in a way to minimize edge connector and board costs, relatively straight forward. The network's physical size is reduced, and hence delays through the network are also reduced. In a VLSI realization, where the entire MIN is realized in a single integrated circuit, such clustering reduces the area requirements of these networks by up to 50%, when compared with the SW-banyan layout.

## 1. Introduction

With the emergence of VLSI technology and wafer-scale integration, the prospect of one day integrating an entire system on one wafer is growing. An important component in many systems, such as multiple processor architectures, is the interconnection network. A large number of interconnection networks have been proposed, the most predominant ones being the crossbar network, numerous multistage interconnection networks (MINs), and the single stage shuffle exchange network (SEN). Optimal layouts for SENs have been presented in [7], and the VLSI aspects of SW-banyans [6] and crossbars have been analysed in [4]. This paper extends the latter by introducing a class of banyan networks, called switch-recursive banyan networks (or SR-banyans), with a topology that tends to minimize the "crossovers" of wires. Such a topology is very useful in the VLSI layout and in the discrete component layout (i.e., with printed circuit boards) of MINs.

An analysis by Franklin on the VLSI requirements of crossbar and SW-banyan networks indicated that "for reasonable values of [network size] the two have roughly comparable space-time performance" [4]. A strong result of Franklin's work is that the area of both networks grows as $O(N^2)$, where $N$ is the number of input or output ports. While there is little that can be done to improve the layout or performance of a crossbar, a large number of parameters may be selected to improve the layout or performance of a banyan network. Switch-recursive banyan networks have such a topology. In addition, other aspects of the system such as the use of multiple metal layers for routing are examined in this paper. We present much simpler and more general models for VLSI area comparisons than presented in [4], and use these to calculate the area requirements of conventional SW-banyan and switch-recursive banyans.

One area of research identified by Franklin was the general question of which MIN, out of many topologically equivalent ones, (i.e., the SW-banyan [6], baseline [2], delta [10], omega [8], indirect binary $n$-cube [11]) to use in a discrete component environment. "The printed circuit boards and the wire interconnecting the boards are much larger contributors to system delays [compared with chip delays], primarily because the distances the signals must travel are orders of magnitude longer" [13]. Our analysis indicates that switch-recursive banyan networks are preferable to other MINs in a discrete component environment by virtue of their minimized wire crossovers, which will result in increased packaging densities and thus fewer boards and connectors to realize the system. Hence, the delays through interconnect in the network will also be reduced.

The use of recursion to realize large switches is not a new concept. However, the topological considerations of switch-recursive networks have gone largely unnoticed. A detailed examination of these issues is presented in this paper.

This paper is organized as follows. Definitions are presented in Section 2. Section 3 presents a simple VLSI area model for crossbars and SW-banyans. This model will be used to calculate the area requirements of the switch-recursive MINs introduced in a later section. Section 4 summarizes Franklin's VLSI delay models for SW-banyan and crossbar networks. Section 5 presents a comparison of SW-banyans and crossbars in a VLSI environment. Our assumptions and analysis are different from Franklin's, and hence our results are significantly different. Section 6 introduces the switch-recursion used to realize MINs with low wire crossover counts. Section 7 presents a VLSI comparison of switch-recursive banyans and crossbars in a VLSI environment. Section 8 contains some concluding remarks.

## 2. Definitions

Many of our definitions have been previously presented by Franklin [4]. We present a brief overview.

An $N \times N$ crossbar connects $N$ processors to $N$ memories, as shown in fig. 1a. The basic switch used in these crossbars has 2 states, which are shown in fig. 1b.

An $N \times N$ *SW-banyan* network, built with $2\times2$ switches, connects $N$ processors to $N$ memories, as shown in fig. 2a [6]. The basic switch used in these networks has 6 states, which are shown in fig. 2b.

*Network Cost:* In a discrete component environment, the cost $C$ of a network is usually the number of switches required to realize it. The cost of the crossbar $C_{cb}$ is $N^2$, and the cost of the banyan $C_{ba}$ is $(N/2)\cdot\log_2 N$.

*Network Delay:* The *average network delay* $D$, experienced by any individual connection request, can be modelled as $1/(1-pb)$ times the average path length through the network, were $pb$ is the *blocking probability* of a network under the assumed operating mode [10].

* This research was partially supported by NSERC (Canada) Grant A-5192.

0190-3918/86/0000/0192 $01.00 © 1986 IEEE

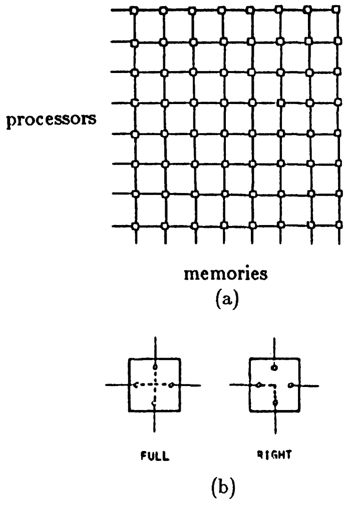


Fig. 1: a) an 8×8 crossbar network. b) the 2 states of a basic switch.

In a crossbar network, a request must travel through $N$ switches on average, hence the delay of a crossbar $D_{cb}$ is given by

$$D_{cb} \approx N \cdot \frac{1}{1-pb_{cb}} \tag{1}$$

where $pb_{cb}$ is the blocking probability of the crossbar network.

In a banyan network, a request must travel through $\log_2 N$ switches, hence the delay of a banyan $D_{ba}$ is given by

$$D_{ba} \approx \log_2 N \cdot \frac{1}{1-pb_{ba}} \tag{2}$$

where $pb_{ba}$ is the blocking probability of the banyan network.

The blocking probabilities of these networks are more difficult to arrive at. For simplicity assume that the network is synchronously circuit switched and all $N$ input ports are occupied in each cycle, (i.e., the network is fully saturated). Franklin used the $pb$ when the set of $N$ requests submitted during each cycle was a randomly selected permutation in the equations above. In this case the $pb$ of a crossbar is 0 and the $pb$ for the banyan was obtained by simulations. However, this permutation model biases the results in favour of the crossbar, since the banyan has large $pb$ 's under this assumption, even for very small $N$.

A more realistic, general performance comparison is to use the blocking probability when the requests are random and independent during each cycle. This model would correspond to general *MIMD* operating environments. In this case, the difference between the $pb$ 's for both networks is reduced considerably.

We assume that the requests submitted during each cycle are random and independent. Under this assumption, the $pb$ of an $N \times N$ crossbar is simply [10]

$$pb_{cb} = 1 - \frac{u_2}{u_1} \tag{3}$$

$$u_2 = 1 - \left(1 - \frac{u_1}{N}\right)^N$$

where $u_1$ is the probability that each processor issues a request in every cycle.

Under these assumptions, the blocking probability of an $N \times N$ banyan, built with 2×2 switches and with $n = \log_2 N$ stages, is simply [10]

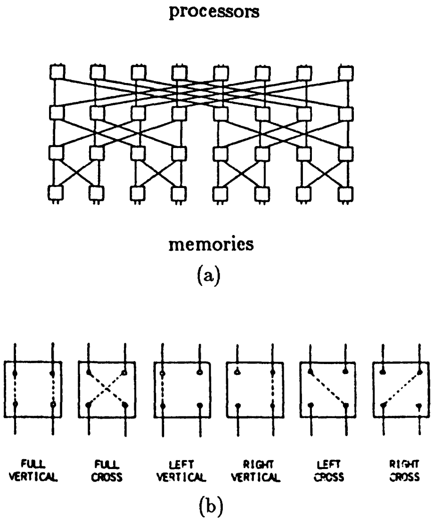


Fig. 2: a) a 16×16 SW-banyan network. b) the 6 states of a basic switch.

$$pb_{ba} = 1 - \frac{u_n}{u_1} \tag{4}$$

$$u_n = 1 - \left(1 - \frac{u_{n-1}}{2}\right)^2$$

and $u_1$ is as above.

*Cost-Delay Products:* A general figure of merit that has been used to compare networks is the ***cost · delay*** product [4]. Using these equations, the cost-delay figure of merit $C \cdot D$ for each of these networks, in a discrete component environment, can be computed:

$$C_{cb} \cdot D_{cb} = N^3/(1-pb_{cb}) \tag{5}$$

$$C_{ba} \cdot D_{ba} = N(\log_2 N)^2/2(1-pb_{ba}) \tag{6}$$

The remainder of this paper examines how this figure of merit changes when the entire network is realized in a single integrated circuit. As will be shown later, the area requirements of the SW-banyan are largely due to routing, and a topology that minimizes the wire crossover count will be introduced.

## 3. VLSI Area Models

We now present a simple and general model for the VLSI area requirements of SW-banyan networks. This model includes the effects of the use of multiple layers of metal for the routing, and is easily extensible to other topologically different MINs. In a later section we introduce the class of switch-recursive banyan networks, which require about 50% less area than the SW-banyan for large network sizes. The area models developed here will be adapted to calculate the area requirements of these recursive MINs.

### 3.1. VLSI Area Model For Crossbar Networks

An accurate model for the crossbar is difficult without undertaking a detailed design in a particular technology. To simplify the analysis, we assume the existence of a basic 2-input 2-output switch, with $w$ bits in each input and output, as shown in fig. 3. We assume that metal lines are 3 λ wide and must be spaced 3 λ apart, where λ is a basic dimension [9]. Assume that the basic switch is square and has dimensions given by Franklin's equation

$$L = 6\sqrt{K(\gamma + w^2)}$$

where $K \approx 1.5$, and $\gamma \approx 20$ [4]. We assume that the switch is

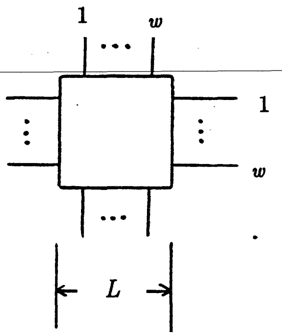


**Fig. 3:** the basic switch with $w$ bits in each input and output

square for simplicity, however, we note that non-square switches would reduce the area of the banyan network considerably.

Assuming the crossbar topology as shown in fig. 1a, the area of a $N \times N$ crossbar is approximately $(N \cdot L_{cb})^2$, where $L_{cb} = L + 3$ (the 3 is due to spacing between switches).

### 3.2. VLSI Area Model for SW-Banyan Networks

We assume that the SW-banyan network is made with basic $2 \times 2$ switches, with $w$ bits in each input and output, as shown in fig. 3, and that these switches have the same area as the basic switches used in the crossbar [4]. This assumption is valid if the control logic associated with each switch is not significant [4], and that the basic switch dimensions are limited by the minimum possible pitch for a transistor in the fabrication technology.

We currently ignore the increased area required by the line drivers used in the switches in the banyan network; this will be examined later. The SW-banyan network topology we are assuming is shown in fig. 2a.

A $2 \times 2$ switch requires height $L_{cb}$, since we assume the basic switches have comparable complexity, as in [4]. The wires into and out of the sides of the basic switch must be bent to go vertically, which will increase the horizontal space required by each switch. We assume, as Franklin did, that each switch can be realized in width $W_{ba} \approx L_{cb} + 6w$, as shown in fig. 4a. Note that $W_{ba}$ may be decreased if more layers of metal are used for routing, and the area of the banyan will be reduced proportionately.

Assume the technology used has at least 2 layers of metal used for routing. By implementing all horizontal inputs in one layer and all horizontal outputs in another, the layout in figures 4a and 4b are realizable. A small area above or below each switch may have to be used to convert from one layer to another (for channel routing).

Consider the $N \times N$ SW-banyan network shown in fig. 2a. This network clearly has a recursive structure; it consists of one stage of $2 \times 2$ switches, connected to two smaller networks of size $N/2$ each. Between the first and second stages (from the top), exactly $N/2$ links, with $w$ bits each, will cross an imaginary line drawn vertically through the middle of the network. Assuming that the vertical components of links are routed on one layer and the horizontal components of links are routed on another, the vertical height required for routing these links is then $(N/2)6w$. Note that this is true regardless of the basic switch size used in the banyan network.

If we have more than 2 layers of metal, then assume any extra layers are used to route the horizontal links between stages. If we let $s$ denote the number of metal layers used for horizontal routing, then the vertical area required is reduced by a factor of $s$.

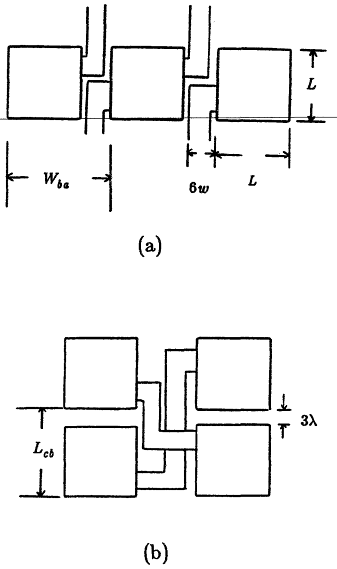


**Fig. 4:** a) minimum horizontal spacing for switches. b) minimum vertical spacing for switches.

(The effects of extra layers on the switches must be reflected in the basic switch size $L$, but it appears that $L_{cb}$ will not change much for a crossbar; In the case of the banyan, $W_{ba}$ may be reduced, and the network's area requirements are reduced proportionally).

The height of the network, $H_{sw}(N)$, is determined by the the height of the first stage of $2 \times 2$ switches, the height required for routing between the first stage and the two smaller networks of size $N/2$, and the height of the smaller network of size $N/2$:

$$H_{sw}(N) = \frac{6wN}{2s} + H_{sw}(2) + H_{sw}(N/2)$$

$$H_{sw}(2) \approx L_{cb}$$

Assuming that a $4 \times 4$ banyan has the layout shown in fig. 4b, then

$$H_{sw}(4) \approx 2L_{cb}$$

### 3.3. Driver Considerations

The preceding development has ignored the area required for drivers in each stage of the banyan network. The propagation delay through a long line is minimized by having a sequence of successively larger drivers driving the line [9]. Franklin has shown that the area of a driver will require between 10 and 30% of the area of the line being driven (assuming NMOS fabrication) [4], and assumed the value 25%, as we will do (i.e., $v = 0.25$). We now must derive an expression for the line length.

Assume the line length must be greater than some minimum value $(2/0.75\alpha)$ in order for a driver to be necessary. The reason for this choice will become apparent in the section on delays.

Consider switches in the first stage of the $N \times N$ SW-banyan. Each switch will have two types of drivers; $w$ drivers that drive a line with a vertical component only, and $w$ drivers that drive a line with both vertical and horizontal components [4]. The vertical component of a line after the first stage is $\approx 6wN/2s$. The sum of the vertical and horizontal components of a line with both components is $\approx (6wN/2s) + (N/2) \cdot (W_{ba}/2)$, of which $\approx W_{ba}/2$ is used to reach the drivers (if any), and the remaider is the length of the line being driven.

We assume these drivers are implemented in a way to increase the vertical dimensions of the switch, rather than the horizontal [4]. Each switch may have $w$ drivers of each type, which must be realized within the width of the switch $W_{ba}$ (note that we differ from Franklin here, and this difference is not insignificant). Assume each line contributes height $dr_{sw}(p)$ to the height of the switch, where $p$ is the line length:

$$dr_{sw}(p) = \frac{v(3 \cdot p)}{W_{ba}} \quad \text{if } p \geq \frac{2}{.75 \cdot \alpha}$$

$$dr_{sw}(p) = 0 \qquad \textit{otherwise}$$

The added height due to drivers after the first stage, $DR_{sw}(N)$, is then

$$DR_{sw}(N) = w \cdot dr_{sw}\left(\frac{6wN}{2s}\right) + w \cdot dr_{sw}\left(\frac{6wN}{2s} - \frac{W_{ba}}{2} + \frac{N}{4}W_{ba}\right)$$

$$DR_{sw}(2) = 0$$

Assume that a 4×4 banyan has the layout shown in fig. 4b and that no drivers are used:

$$DR_{sw}(4) = 0$$

The height of an $N \times N$ SW-banyan network is then

$$H_{sw}(N) = \frac{6wN}{2s} + DR_{sw}(N) + H_{sw}(2) + H_{sw}(N/2)$$

The width of the banyan network is determined by the $(N/2)$ 2×2 switches, and is given by $\approx (N/2)W_{ba}$. Hence, the total area required for a $N \times N$ SW-banyan, made with 2×2 switches, with $w$ bits per input/output, and with $s+1$ metal layers for routing, is given by

$$\approx H_{sw}(N) \cdot \frac{N}{2} W_{ba} \tag{7}$$

## 4. VLSI Delay Models

We use the NMOS delay models developed by Franklin [4], but with a few modifications. Recall that Franklin assumed that the $pb$ used in equations (5) and (6) was the observed blocking probability when the request pattern submitted during each cycle was a randomly selected permutation. We assume that the individual requests are random and independent. Hence, in our delay model the crossbar also has a non-zero blocking probability.

In addition, Franklin assumed that all lines being driven in the banyan required drivers. This is not true in the last few stages where the line lengths are relatively small.

### 4.1. General VLSI Delay Models

A very brief summary of Franklin's NMOS delay models is presented [4]. Assume each switch is implemented with NOR gates in NMOS technology. If each gate has a fanout of $f$, and each switch has $m$ levels of logic, then the switch delay is typically [4]

$$\approx 2.5mf\ \tau$$

where $\tau$ is the transit time through the gate's active region.

If the gate is driving a line of length $p$ whose total capacitance is comparable to the gate capacitance, then the delay associated with driving such lines is [4]

$$\approx \tau(1 + 0.75\alpha p)$$

If the line to be driven has a total capacitance that is much larger than a minimum sized gate's capacitance,then the delay is minimized by having a sequence of successively larger drivers driving the line [4,9]. Assuming each gate is 4 times the size of its predecessor [4], then the delay associated with driving a line of length $p$ is

$$\approx 2\tau \log_2(0.75\alpha p)$$

If the line is very long, then a resistance factor should be taken into account and a transmission line model should be used. We assume that a simple capacitance based model is sufficient.

Franklin assumed a value of $\alpha$=0.1. However, an examination of *SPICE* circuit parameters for NMOS and CMOS indicates that $\alpha$=0.03 is typical today. The value $\alpha$=0.03 will be used throughout this paper. An implication of this assumption is that lines can be much longer before a driver is required.

Let $d(p)$ be the delay due to a line of length $p$:

$$d(p) \approx 2\,\tau \log_2(0.75\alpha p) \quad \text{if } p > \frac{2}{0.75\alpha}$$

$$d(p) \approx \tau(1+0.75\alpha p) \quad \text{if } p \leq \frac{2}{0.75\alpha}$$

$$d(p) = 0 \quad \text{if } p = 0$$

### 4.2. Crossbar Delay Model

The delay of an $N \times N$ crossbar is given by [4]

$$D_{cb} = \frac{\tau}{1-pb_{cb}}\Big(2.5Nmf + (N-1)(1+0.75\alpha 3)\Big) \tag{8}$$

We will use $m=2$, $f=2$, and $\alpha$=0.03 throughout the paper, and $pb_{cb}$ can be calculated from equation (3). The parameter $\tau$ will cancel out later.

### 4.3. SW-Banyan Delay Model

Let $D(N)$ be the average path delay through an $N \times N$ SW-banyan:

$$D(N) \approx \frac{1}{2}\left[ d\left(\frac{6wN}{2s}\right) + (1+0.75\alpha W_{ba}/2)\tau + d\left(\frac{6wN}{2s} - \frac{W_{ba}}{2} + \frac{N}{4}W_{ba}\right) \right] \tag{9}$$

$$D(N/2) + D(2)$$

where $D(2) = 2.5mf\ \tau$. Within the large brackets, the first term is the delay of a line with a vertical component only, the second term is the delay to reach the line drivers for lines leaving the side of the switch, and the third term is the delay over a line with a vertical and horizontal component.

If we assume that 4×4 banyans have the layout shown in fig. 4b and have no drivers, then

$$D(4) \approx (1+0.75\alpha W_{ba})\tau + 5mf\ \tau$$

Hence, the delay of an $N \times N$ SW-banyan is given by

$$D_{ba} = \frac{1}{1-pb_{ba}} D(N) \tag{10}$$

where $pb_{ba}$ can be calculated from eq. (4).

## 5. VLSI Comparisons

### 5.1. SW-Banyan and Crossbar Comparisons

In fig. 5, the area, delay and space-time comparisons of SW-banyan and crossbar networks are presented. The space-time ratios in the VLSI environment are similar to the cost-delay ratios in the discrete component environment.

Our area results for the SW-banyan are lower than Franklin's by about 30% [4], which can be explained as follows. While Franklin's analysis was basically correct, his equation (16) increased the area of the banyan by one extra stage of drivers. (This alone accounted for an error of 25%). Also, Franklin's analysis assumed that the area required by drivers was realized in such a way that the width of the switch did not increase (only its height was allowed to increase). Hence, the vertical contribution to a switch of width $L$ due to a driver requiring area $A$ was given by $A/L$. However, while the true switch width is $L$, switches are

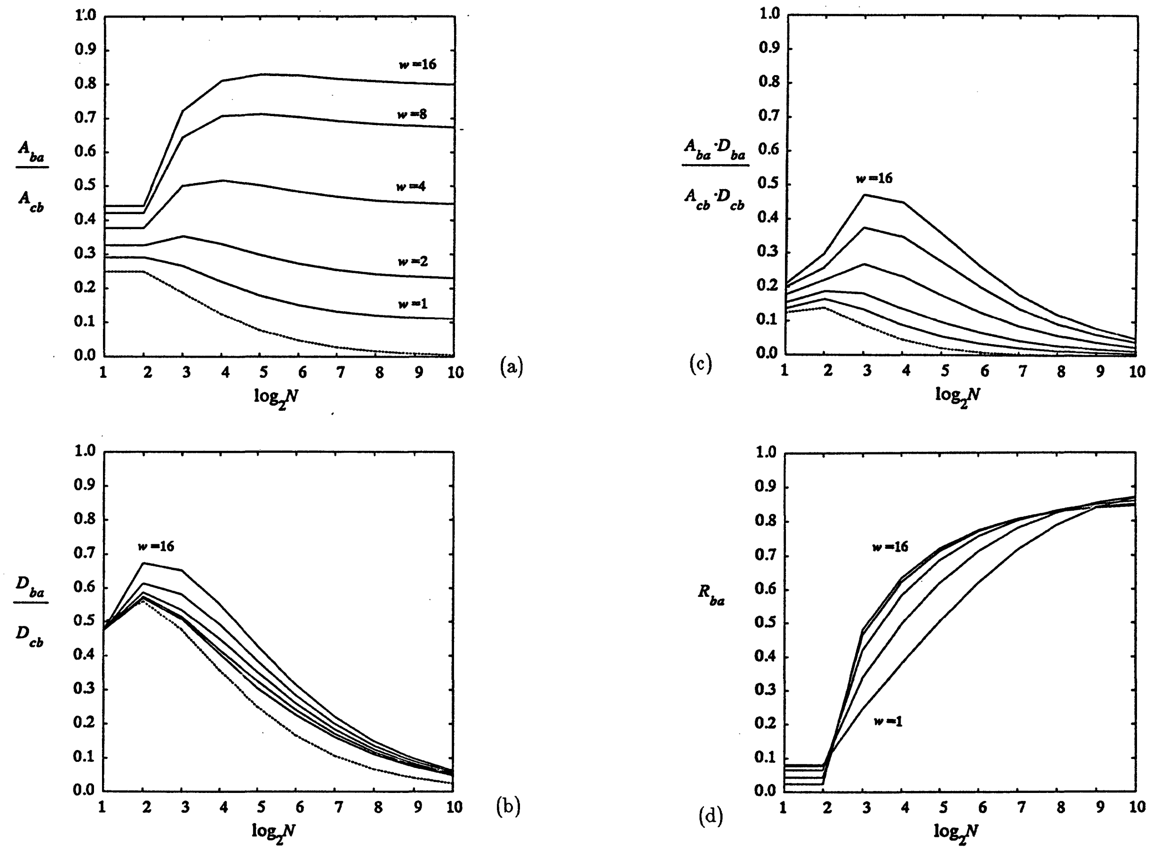


**Fig. 5:** a) ratio of SW-banyan to crossbar areas. b) ratio of SW-banyan to crossbar time delays. c) ratio of SW-banyan to crossbar space-time products. (dashed curves are from traditional analysis); d) fraction of SW-banyan area used for routing.

spaced $6w$ apart, so each switch requires $L + 6w \approx W_{ba}$. Hence, our analysis assumed that the drivers can be realized over the width $W_{ba}$, and the added contribution to a switch's height is then $A / W_{ba}$.

Our delay curves are significantly different from Franklin's since we assumed that the blocking probability of these networks under the assumption of random request distributions was used in equations (8) and (10), and we have used the value of $\alpha$=0.03. The effectiveness of each network is indicated in fig. 5c, the space-time ratios. Our space-time ratio drops off more rapidly than Franklin's, indicating that banyans become more effective as $N$ increases.

Fig. 5d illustrates the area required by routing in the SW-banyan networks. As the network size $N$ increases, this area rapidly approaches an asymptotic limit. This limit can be explained as follows. As $N$ increases, the area required by switches in a banyan approaches 0, and the majority of the area is due to routing and drivers. The area of actual metal routing (not including the spacing between metal lines) is about 50% of the banyan's area, and the drivers require 25% of this area (by assumption). Hence, the drivers require about 12.5% of the area of a banyan, for large $N$.

## 6. Recursive MINs

The preceding graphs indicate that in a VLSI realization, the majority of a large SW-banyan's area is devoted to interconnect between stages. This area can be reduced by selecting a topology that minimizes the wire crossover count. In a discrete component realization, minimizing the wire crossovers will result in increased packaging densities, and hence minimized cost and delays. In this section, we present such topologies.

The SW-banyan network shown in fig. 2a (in section 2) clearly has a recursive structure; a network of size $N$ consists of one row of 2X2 switches that are connected to two smaller networks of size (N/2) each. We introduce another type of recursion, which we call ***switch-recursion***. The basic idea is to use a MIN (made with smaller MINs) as the basic switch in the realization of a larger MIN. We illustrate the concept by examples on various MINs, which we call SR-MINs.

### 6.1. Switch-Recursive Delta Networks

Consider a 64X64 ***delta*** network [10] made with 2X2 switches, as shown fig. 6a. This network can be reorganized into a network where each recursive switch is a 4X4 delta network, as shown in fig. 6b, and into a network where each recursive switch is a 8X8 delta network, as shown in fig. 6c. An advantage in the latter realizations is the decreased wiring complexity. In a printed-circuit board environment, each recursive switch should be realized on the same board. This would translate into an increased component density per board (since less board area is used for routing to edge connectors), fewer boards to realize the network, and decreased interboard communications and delays (through edge connectors and ribbon cables). In a VLSI environment, the clustering and high connectivity within each recursive switch will result in reduced wiring area. (Many automatic placement and routing programs look for such clustering in the "net list" of a circuit, for improved layout [12]).

### 6.2. Switch-Recursive Banyan Networks

The previous figures illustrated the recursive process when

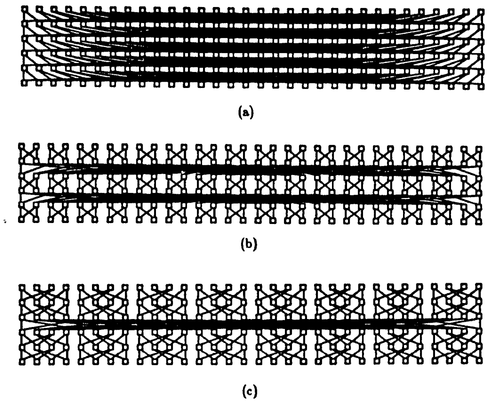


**Fig. 6:** a) a 64×64 delta network. b) a 64×64 delta network with 4×4 recursive switches. c) a 64×64 delta network with 8×8 recursive switches.

applied to delta networks. In a VLSI environment, the SW-banyan network has significantly lower area requirements than the delta. The recursive process can also be applied to SW-banyan networks, as shown in fig. 7. (Note that "delta" networks are actually a subclass of "banyan" networks, as defined in [6]; we use the term "SW-banyan" to denote the topology illustrated in fig. 2a and fig. 7a).

### 6.3. Generalized Switch-Recursive Banyans

A large number of possible topologies exist when switch level recursion is used. A conventional $N \times N$ network can be realized by an $m$ stage network, where each stage is made of $k_i \times k_i$ crossbars [3]. The integers $<k_1,k_2,...,k_m>$ correspond to the factors of $N$, i.e., $k_1 \cdot k_2 \cdot \cdots \cdot k_m = N$. Given a basic switch size, such as a 2×2 crossbar, we can synthesize each $k_i \times k_i$ crossbar with a MIN, provided the crossbar size is a power of 2. After applying switch level recursion corresponding to some factorization of $N$, we are left with a $\log_2 N$ stage network with $N/2$ switches per stage, as before. However, the topology is different and generally has more localized communications.

Clearly, this recursion can be extended to many levels, so that a $k \times k$ switch could be realized by recursive switches, which in turn use recursive switches. We have not yet proven results on an optimal factorization of $N$. However, a heuristic that seems to give optimal results is as follows (let $n = \log_2 N$). To realize an abstract $N \times N$ network, we use a 2 stage recursive SW-banyan, with recursive switches of size $2^{\lfloor n/2 \rfloor}$ in one stage, and recursive switches of size $2^{\lceil n/2 \rceil}$ in the other stage. One recursive banyan network derived from this heuristic is shown in fig. 7b. This particular network requires only one level of recursion.

The recursion process only improves the area requirements when the network size is >8. We have written a ***branch and bound*** routine to try all possible factorizations for a particular $N$, and the heuristic always yields the optimal result.

## 7. VLSI Comparison of SR-Banyans and Crossbars

An analysis of the area requirements of SR-banyans is now presented. Let $H_{sr}(N)$ represent the height required for an $N \times N$ SR-banyan. The SR-banyan consists of 2 stages, with recursive switches of size $2^{\lfloor n/2 \rfloor}$ in one stage, and recursive switches of size $2^{\lceil n/2 \rceil}$ in the other stage, where $n = \log_2 N$.

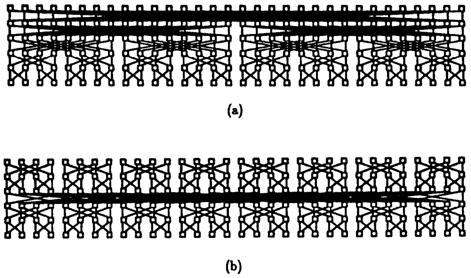


**Fig. 7:** a) 64×64 SW-banyan network, made with 2×2 switches. b) 64×64 SW-banyan network, made with recursive 8×8 switches, in turn made with 2×2 switches.

Our analysis indicates that the area and delays are minimized when the smaller switches are in the first stage, so let each recursive switch in the top stage be of size $2^{\lfloor n/2 \rfloor}$. The recursive switches in the top stage have width $T_{ba} = 2^{\lfloor n/2 \rfloor - 1} * W_{ba}$, and the recursive switches in the bottom stage will have width $B_{ba} = 2^{\lceil n/2 \rceil - 1} * W_{ba}$, where $W_{ba}$ is the width of a 2×2 switch as described in section 3.

Since $N/2$ links, with $w$ bits each, will cross an imaginary line drawn through the middle of this network, the vertical height required for routing between these two stages is $6wN/2s$, where $s$ is the number of metal layers used for horizontal routing.

The height required by drivers for a recursive switch in the top stage must be calculated. Consider a recursive switch nearest the middle of the row in the top stage. At this point, the channel used for routing the wires is most dense and any height required for drivers will contribute to the height of the network (at the ends, the drivers can extend into the channel since very few wires exist there). One line leaving this switch will have ≈ a vertical component of $6wN/2s$ only, two lines will each have this vertical component and a horizontal component of $\approx B_{ba}$, another pair of lines (if any) will have this vertical component and a horizontal component of $\approx 2B_{ba}$, etc. Assuming that the total driver area can be realized over the width of the switch $T_{ba}$, then each line of length $p$ contributes height $dr_{sr}(p)$ to the height of the switch (note that $T_{ba}$ changes according to $N$):

$$dr_{sr}(p) = \frac{v(3p)}{T_{ba}} \quad \text{if } p > \frac{2}{0.75\alpha}$$

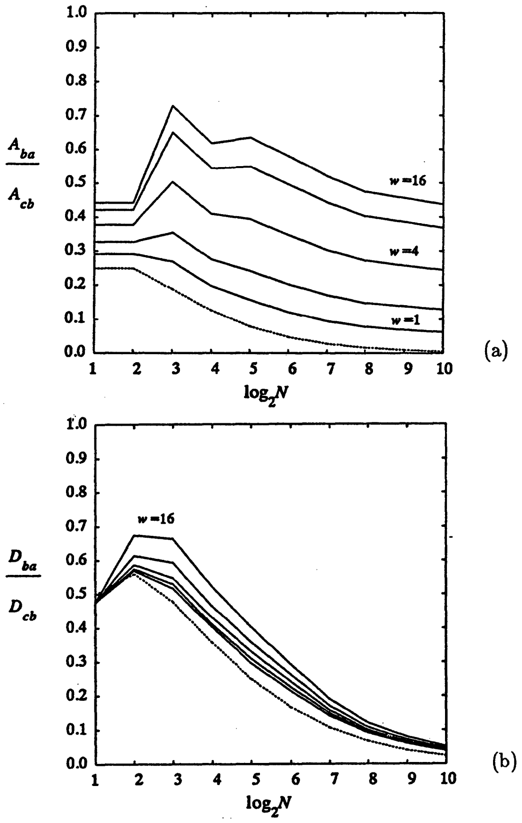
$$dr_{sr}(p) = 0 \quad \textit{otherwise}$$

The total height added to a switch due to drivers is then

$$DR_{sr}(N) \approx \sum_{i=1}^{2^{\lfloor n/2 \rfloor}} w \cdot dr_{sr}\left(\frac{6wN}{2s} + abs\,(2^{\lfloor n/2 \rfloor - 1} - i) \cdot B_{ba}\right)$$

where $n = log_2 N$. Hence, the total height required for the SR-banyan is then

$$H_{sr}(N) = \frac{6wN}{2s} + DR_{sr}(N) +$$
$$H_{sr}(2^{\lceil n/2 \rceil}) + H_{sr}(2^{\lfloor n/2 \rfloor})$$



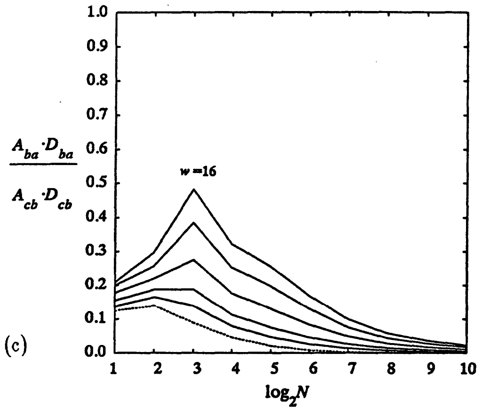


**Fig. 8:** a) ratio of SR-banyan to crossbar areas. b) ratio of SR-banyan to crossbar time delays. c) ratio of SR-banyan to crossbar space-time products. (dashed curves are from traditional analysis);

where $H_{sr}(2)$ and $H_{sr}(4)$ are as before.

The total area of an $N \times N$ SR-banyan is then

$$\approx H_{sr}(N)\cdot\frac{N}{2}W_{ba} \qquad (11)$$

The delay equations from section 4 can be adapted in a similar manner to yield the delay of SR-banyans. (We have used the average delay over a recursive switch at the end of each row and a recursive switch nearest the middle of each row.)

Fig. 8 illustrates the area, delay, and space-time comparisons between SR-banyans (given by the heuristic) and crossbars in a VLSI environment. The area requirements of the SR-banyans are about 50% less than that of SW-banyans for large $N$. For $N=1024$, with $b=16$ and $s=1$, the SR-banyan requires 45% less area than the SW-banyan, and has 19% less delay. For $N=16{,}384$, the savings are 49% and 25% respectively. (Note that Delta [10] and Omega [8] networks require $O(N^2 \cdot \log_2 N)$ area, and are not even considered here.)

Fig. 9 illustrates the area and space-time comparisons when 2 and 4 layers of metal are used for horizontal routing. Clearly, as more layers of metal are used, the banyan network becomes more attractive.

## 8. Conclusions

A class of multistage interconnection networks, called SR-banyans, that minimizes the number of wire crossovers has been presented. The basic idea is to use a MIN (made with smaller MINs) as the basic switch in the realization of a larger MIN. We have presented a heuristic that appears to yield the optimal network, in terms of minimized wire crossovers, for a given network size $N$ and a given basic switch size $k$. The resulting ***switch-recursive*** banyan has far fewer wire crossovers when compared to others MINs, such as the SW-Banyan, Baseline, Delta and Omega networks.

More importantly, switch-recursive MINs have very distinct clusters in the topology, with heavy communications within clusters, and little communications between clusters. It has been estimated that printed circuit boards and wire interconnect are much bigger contributors to system delays than gates, in a discrete component realization [13]. In [2] it is estimated that the round trip network delay is dominated by cable delay, which is irreducible because it is imposed by the physical size of the assemblage. The clustering in switch-recursive MINs makes the partitioning of these networks onto printed circuit boards, in a way to minimize edge connector and board costs, relatively straight forward. The delay through a switch-recursive banyan realized in a discrete component environment should be reduced significantly, since fewer edge connectors and ribbon cables must be traversed, and since the network size should be reduced significantly.

In a VLSI realization, where the entire MIN is realized in a single integrated circuit (as a $\log_2 N$ stage network with $N/2$ switches per stage), such clustering reduces the area requirements of these networks by up to 50% (for large $N$) when compared with the SW-banyan layout. The average delay through the network is reduced by up to 25% (for large $N$) when compared with the SW-banyan layout. (The area required for an 8×8 banyan can be improved further; see [14].)

However, even better layouts for banyans exist if we assume that the links into the network are not necessary. This could be true if a) processors are realized on the same substrate and have negligible area, as in [7], or b) each switch actually contains a processor. Asymptotically up to 50% of the area of the SR-banyan or up to 75% of the area of the SW-banyan can be saved. The savings is achieved by laying out the top row of a SW-banyan as two horizontal rows, one above the other, and each half as long as the original. The two subnetworks (of size $N/2 \times N/2$) are placed one above and one below this "row" (using the SR-banyan layout), for a total area about equal to that for the two subnetworks. (For $N=1024$, $b=16$, and $s=1$, this layout requires 48% less area than that for the SR-banyan, and 70% less than that for the SW-banyan.)

While this paper has focused on SW-banyans and SR-banyans, the results are more general. It has been proved that all "strict-buddy type MINs" (i.e., SR-banyan, SW-banyan, baseline, delta, indirect binary n-cube, and omega networks) are isomorphic [1]. Hence, if we are able to change the labels assigned to the pro-

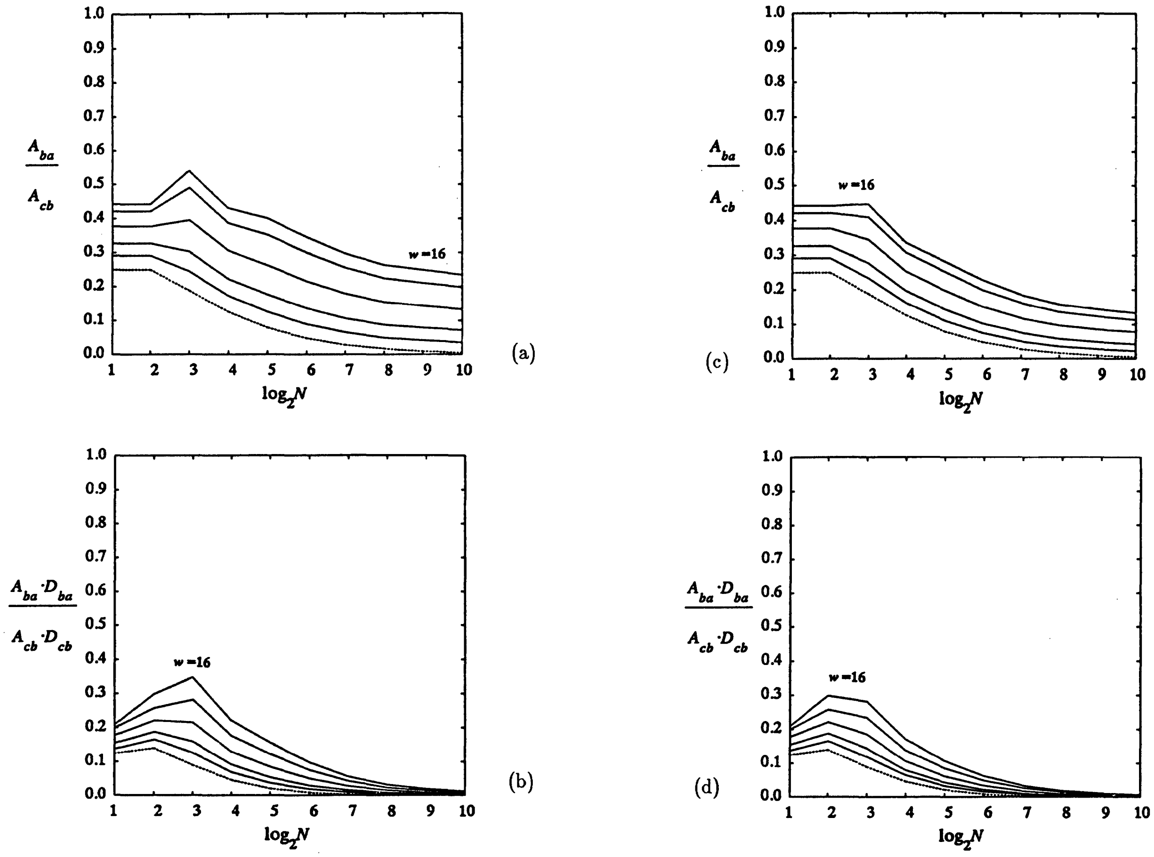


**Fig. 9:** a) ratio of SR-banyan to crossbar areas when 2 layers of metal are used for horizontal routing. b) ratio of SR-banyan to crossbar space-time products when 2 layers of metal are used for horizontal routing. c) ratio of SR-banyan to crossbar areas when 4 layers of metal are used for horizontal routing. d) ratio of SR-banyan to crossbar space-time products when 4 layers of metal are used for horizontal routing.

cessors connected to an SR-banyan, than any strict-buddy type MIN can be realized. This labelling can be accomplished by having a writable register associated with each processor that contains its label. This result has an important implication. The omega network is very useful in array processor architectures [8], but has a very high cost due to the number of link crossovers. However, we can transform the omega network into an SR-banyan, and obtain the exact same behaviour at a much lower cost. Similarly, any other strict-buddy type MIN can be realized simply by changing the processor labels.

## 9. References

[1] D.P. Agrawal, "Graph Theoretical Analysis and Design of Multistage Interconnection Networks", IEEE Trans. Comput., Vol C-32, July 1983, pp. 637-648

[2] G.H. Barnes and S.F. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems", IEEE Computer, Dec. 1981, pp. 82-90

[3] L.N. Bhuyan and D.P. Agrawal, "Design and Performance of Generalized Interconnection Networks", IEEE Trans. Comput., Vol. C-32, Dec. 1983, pp. 1081-1090

[4] M.A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communications Networks", IEEE Trans. Comput., Vol C-30, April 1981, pp. 283-290

[5] M.A. Franklin, D.F. Wann, W.J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Networks", IEEE Trans. Comput., Vol C-31, Nov. 1982, pp. 1109-1116

[6] G.R. Goke and G.J. Lipovski, "Banyan Networks for partitioning multiprocessor systems", Proc. 1st Annual Symp. Computer Architecture, 1973, pp. 21-28

[7] D. Kleitman, F.T. Leighton, M. Lepley, G.L. Miller, "New Layouts for the Shuffle-Exchange Graph", ACM Symposium on the Theory of Computing, 1981, pp. 278-292

[8] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans. Comput., Vol C-24, Dec. 1975, pp. 1145-1155

[9] C. Mead and L.Conway, "Introduction to VLSI Systems", Reading MA: Addison-Wesley, 1979

[10] J.H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors", IEEE Trans. Comput., Vol C-30, Oct. 1981, pp. 771-780

[11] M.C. Pease, "The Indirect Binary n-Cube Microprocessor Array", IEEE Trans. Comput., Vol C-26, May 1977, pp. 458-473

[12] J.S. Rose, "Fast, High Quality VLSI Placement on an MIMD Multiprocessor", PhD Thesis, in preparation, Dept. of Elec. Eng., University of Toronto

[13] L.K. Steiner, "Supercomputer Design Challenges", First Int. Conf. on Supercomputing Systems, Dec. 1985, pp. 3-7

[14] T.H. Szymanski, PhD Thesis, in preparation, Dept. of Elec. Eng., University of Toronto

# EVALUATION OF THREE INTERCONNECTION NETWORKS FOR CMOS VLSI IMPLEMENTATION[1]

**Pinaki Mazumder**

**Computer Systems Group**
**Coordinated Science Laboratory**
**University of Illinois**
**Urbana, IL 61801.**

## ABSTRACT

This paper envisages to evaluate three types of static interconnection networks for VLSI implementation. At first, a realistic VLSI computational model has been proposed taking into account several aspects of VLSI technology, like chip embedding, interconnection delay, chip yield, device dissipation, device failure, etc. The criteria of evaluation have been selected from three orthogonal aspects - physical (chip area and dissipation), computational speed (message delay and message density) and cost (chip yield, operational reliability and layout cost). The result of the evaluation reveals that the Cellular Networks similar to two dimensional meshes are most suitable for on chip parallel processing in VLSI. The salient feature of this paper is to augment the selection criteria for the interconnection networks from the classical $AT^2$ metric and to provide results pertaining to realistic VLSI implementation.

## 1. INTRODUCTION

Over the past two decades many interconnection networks have been proposed in the literature for SIMD architectures. Extensive accounts of these networks and their performance evaluation have been reported in [1, 2, 3, 4]. Now with the advent of submicron silicon technology, more than ten million devices can be integrated in the VLSI circuit [5] and it has opened a new vista in parallel processing. On-chip multiprocessing by several processors for executing special algorithms is envisioned to be the major application goal of the future generation computers using massive parallelism.

In order to justify the need for re-evaluation of interconnection networks, it should be noted that in the design of earlier networks adapted for non-VLSI environment

a. spatial distribution of the processors is not a constraint on the design,

b. signal propagation time is exclusively determined by the velocity of electromagnetic wave in the resistive medium and is negligibly small compared to the speed of operation. Thus the length of the interconnecting wire is not a constraint on design,

c. cost of the system is directly proportional to the average number of links per node,

d. fault-tolerance capability of such networks is merely a topological property (i.e., whether alternate message transmission routes exist or not)

On the contrary, under a VLSI environment

a. the spatial distribution of the processors play an important role on the total chip area.

b. the interconnecting wire behaves as transmission line having both resistive and capacitive components and signal propagation time is largely dependent on these values which are directly proportional to the length of the interconnection,

c. link per node does not have any direct relevance to design cost. The regularity of the networks topology decides the layout cost and size of chip decides the fabricational cost,

d. fault-tolerance has an additional role to play. To improve the device yield and thereby to reduce the overall cost, it is necessary to introduce redundant processors. The existence of long interconnects increases the chip failure probability, both at the time of fabrication and in normal operation.

This paper which was written as a term project in an architecture course envisages to evaluate the performance of three types of static interconnection networks and to determine how these new constraints modify the performance of these interconnection networks in VLSI implementation. The choice of the networks has been motivated from the facts that these networks have been widely pursued in the literature for designing many algorithms [6, 7] and all of them have optimal VLSI layouts. Also, these provide the insights to three topologically different classes of networks. The networks discussed are: two dimensional meshes, binary trees and Cube Connected Cycles (CCC). These three networks conceptually belong to separate classes in the sense that if the interprocessor link is constrained to have constant length, their distribution in three dimensional space reveals somewhat planar, conical and spherical surfaces, respectively. Thus the results of this evaluation can be easily extended to assess the performance of other networks, because most networks have one of these three topologies in three dimensional space. Networks with wraparound connections like the ILLIAC IV describe a torroidal surface and are not suitable for VLSI implementation because the endaround interconnection introduces large delay and also expands the area by a factor of two. Otherwise, the overall performance of these networks is similar to two dimensional meshes [3].

At first, a computational model for the VLSI technology is proposed and the results of the model is employed to evaluate the performance of the networks. The model is similar to Thompson's [8] model and additionally accounts for device faults and chip yield. The criteria of evaluation are enumerated from three orthogonal viewpoints, viz. area, speed and cost. The results of the analysis are compared and it is concluded that the regular structured Cellular Networks having short interconnects similar to the two dimensional meshes are the most suitable candidates for VLSI implementation with two layered interconnects. It may be pointed out that the conclusion

[1] This research was supported by the *Semiconductor Research Corporation* (SRC) under contract 84-06-049.

made here is on the basis of asymptotic performance and may not hold good for small sized networks. Some of the results discussed here have been reported by other researchers. But the modest objective of this paper is to identify the relevant aspects of VLSI technology and to demonstrate that under the new constraints (like chip yield, layout regularity, failure probability of long interconnects, reliability improvement per redundant processor, etc.), Cellular Networks indicate overall better performance than H-tree, CCC and other fast networks. This is in direct contrast to the results of [3, 9] where it was concluded that fast networks like CCC, PSN, dual bus hypercubes, etc., have better overall performance. The paper is organized as follows: Section 2 discusses the VLSI model, Section 3 discusses the criteria of evaluation, Section 4 provides the performance analysis, Section 5 compares the performance of these three networks and Section 6 concludes the paper.

## 2. VLSI MODEL OF COMPUTATION

The model is based on MOS and C(omplementary)MOS technologies and assumes a two layered model for interconnects. The two layered model uses one layer of polysilicon and one layer of metal for interconnection networks. A two layered model is easy to fabricate and provides a high yield. A composite two layered metal interconnects like Al-Pt-Ti-Au improve the device performance, but is difficult to fabricate and is not assumed in this model. The model makes four types of assumptions as discussed below:

### 2.1. Embedding Assumptions:-

**Assumption 2.1.1:** *All processors are identical in shape and size.*

Since the processors have identical computational power, they need equal computational area and hence can be identical in shape and size.

**Assumption 2.1.2:** *Each processor occupies O(1) area and can be represented as a square of unity area.*

The natural layout of any computational circuit has a rectangular topology. Any rectangular topology can be converted to a square topology by increasing the overall area by at most a factor of three [10, 11].

**Assumption 2.1.3:** *Wires always run either in vertical or in horizontal direction in two different layers.*

This scheme is called the Manhattan interconnection technique [12]. Since the processors are aligned parallel to the Cartesian co-ordinates, interconnection wires being perpendicular to the sides of the processors run along the horizontal and vertical directions.

**Assumption 2.1.4:** *At most two wires may cross over each other at any point in the plane.*

In MOS technology metal wires can cross over either polysilicon or diffusion without making contact. Whenever polysilicon runs over diffusion automatic contact is established resulting in a transistor at the overlapping surface.

**Assumption 2.1.5:** *Wires can bypass the faulty nodes.*

Switches can be placed over the processor to isolate them [13] and built in self test circuit can be incorporated to test whether the processor is faulty.

**Assumption 2.1.6:** *Wire of length $l$ behaves as a transmission line having $O(l)$ resistance and $O(l)$ capacitance.*

Since the interconnection wire has distributed resistance and capacitance all along the length of the wire, $O(l)$ assumption is justified.

### 2.2. Timing Assumptions:-

**Assumption 2.2.1:** *Processors have O(1) computational delay.*

Since the processors are identical in shape and occupy O(1) area, the delay is constant.

**Proposition 1** *Wire of length, $l$ introduces $O(l^2)$ delay and this is the upper bound of propagation delay.*
Proof: see [14].

For metal wire, the delay can be reduced to $O(\log l)$ by introducing a cascade of drivers as stated in the following proposition.

**Proposition 2:** *The metal wire of length, $l$ has a minimum of $O(\log l)$ delay and it needs $O(\log l)$ stages of drivers having a combined area of $O(l)$.*
Proof: see [15].

Mead & Conway [14] have shown that quadratic propagation delay in polysilicon (or diffusion ) wire of length $l$ can be minimized to $O(l)$ by introducing $l/k$ drivers at regular interval such that the driver delay is equal to signal propagation time over polysilicon (or diffusion) of length $k$. Thus

**Proposition 3:** *The poly/difusion wire of length, $l$ has a minimum of $O(l)$ delay and it needs $l/k$ stages of drivers having a combined area of $O(l)$.*
Proof: see [14].

Ramachandran [16] has shown that if many parallel long wires exist and space available externally to the processors is not sufficient to lay down all the drivers, then the delay can be minimized at the cost of increasing the size of the processors. In the worst case, this results in quadrupling the area of the processor.

### 2.3. Energy Dissipation Assumption:-

**Assumption 2.3.1** *Each processor of size $O(1)$ consumes $O(1)$ power in unit time.*

Since the power consumption in CMOS occurs due to charging and discharging of nodal capacitors, the switching power consumption at a node is equal to $0.5CV^2f$, where $f$ is the frequency of operation and C is the nodal capacitance. For a static CMOS circuit, the total switching power dissipation depends on the input to the processor and the internal circuit configuration of the processor. On the average, the dissipation will be a constant fraction of the total processor area and is bounded by $O(1)$. For dynamic CMOS circuit, since each gate will be charged by the refreshing clock, an $O(1)$ power consumption automatically follows.

**Proposition 4:** *Wire of length $l$ consumes at most $O(l)$ power.*
*Proof:* The power consumed by the resistor $R(l)$ in charging the capacitor $C(l)$ is equal to $0.5C(l)V^2f$. The capacitor stores charge only and does not dissipate any energy. Hence the upper bound is $O(l)$. □

### 2.4. Failure Assumptions:-

The failure in VLSI can be classified into two categories - chip related defects which lower the IC yield and the field operational which is a function of time.

**Assumption 2.4.1:** *Defects in the fabrication of an IC (viz., pinhole defects in oxide, defects in photoresist, implant defects, etc.) are randomly distributed and statistically independent.*

Generally, gross imperfections causing large areas of the chip to be bad (i.e., area defects) are detected at slice test and and line defects (like scratches) do not occur in a well-controlled process [17]. The most commonly encountered chip related flaws are random isolated spot defects. Clusters of spot defects also occur, but they can be treated as a single defect since usually the size of the processor is large enough to encompass the whole cluster. Thus the probability of failure of processors is independent and the same for all the processors.

**Assumption 2.4.2:** *The yield of an IC decreases inversely with the size of the chip.*

Since the defects occur randomly as Poisson's process, the yield (i.e., the probability of having no defect in a chip) is given by

$$Y_0 = e^{-AD_0} \qquad (2.4)$$

Practical results show that this under-estimates the yield slightly for larger size chips [18] and Poisson's distribution of random defects represents the yield pessimistically [19]. A better fitting with practical results has been obtained by employing Bose-Einstein statistics and it is shown [20] that the yield of defect-free chips are given by

$$Y_0 = \frac{1}{1+AD_0} \qquad (2.5)$$

where, $AD_0$ is the average number of defects per chip.

The random distribution of defects can be assumed to increase linearly with size for small sized defects like pinholes and to decrease as the cube of defect size for large sized defects like those occurring in diffusion patterns [21]. Using this random defect distribution, it can be shown that the effect of these defects on interconnection is very drastic and the failure rate is related to the aspect ratio of the interconnection wire. Formally,

**Proposition 5:** *Long wire of length, $l$ and width, $\lambda_w$ can fail with a probability proportional to $O(l/\lambda_w)$.*

Proof: Let L be a long wire of length, $l$ and width, $\lambda_w$. Let a circular defect of diameter, $\eta$ occur randomly on the conducting wire resulting in a hole as shown in Figure 1. If the width of the conducting material available for current conduction is sufficient to carry current in normal operation without causing any catastrophic failure, then the pinhole caused by the defect will not have any effect on the wire. Let $\delta$ be the minimum width of the wire required for normal operation at a particular current density, then defects of diameter, $\eta<(\lambda_w-\delta)$ will not cause any failure, while the defects of diameter, $\eta \geqslant (\lambda_w-\delta)$ will cause failures if they occur such that they do not leave $\delta$ width of conducting wire. The locus of the center of defects that lead to failure is called *critical area*, $A(\eta)$ and is given by

$$A(\eta) = \begin{cases} 0 & \text{for } 0 \leqslant \eta < (\lambda_w - \delta), \\ (\eta - \lambda_w) l, & \text{for } (\lambda_w - \delta) \leqslant \eta < \infty. \end{cases}$$

If $f(\eta)$ denotes the distribution of defect density, then the average value of the critical area with respect to defect size distribution is given by

$$\bar{A} = \int_0^\infty A(\eta) f(\eta) d\eta.$$

Very small defects can be assumed to increase linearly with defect size up to a certain value (say $\eta_0$) and large defects can be assumed to vary inversely as the cube of defect size [21]. Thus

$$f(\eta) = \eta/\eta_0^2 \quad \text{for} \quad 0 \leqslant \eta < \eta_0,$$
$$f(\eta) = \eta_0^2/\eta^3 \quad \text{for} \quad \eta_0 \leqslant \eta \leqslant \infty.$$

Hence,

$$\bar{A} = \int_{\lambda_w}^{\infty} (\eta - \lambda_w) l (\eta_0^2/\eta^3)\, d\eta = O(l/\lambda_w), \quad \text{assuming } \delta << \lambda_w.$$

The probability of failure of the wire is directly proportional to its critical area and is $O(l/\lambda_w)$. □

## 3. CRITERIA FOR EVALUATION

The interconnection networks have been modeled to study three aspects - a) the physical aspects, b) the computational aspects and c) the cost aspects. The overall performance of the networks has been estimated from these viewpoints which form three orthogonal classes.

### 3.1. Physical aspects:

The physical aspects relate to the chip area and power consumption by the chip.

### 3.1.1. Chip Area

The chip area refers to the total area required to lay the processors and the communication links. The area occupied by the processors is the computation area. The ratio of the computation area to the total area is the performance metric and is defined as *area efficiency*.

### 3.1.2. Power Consumption

Like the chip area, the total power consumed by the chip can be divided into computational power which is equal to the power dissipated by the processors and the power required to drive the communication links. The ratio of the computational power to the total power consumed by the chip is defined as the *power efficiency*. The average power consumption inside the chip is an important parameter because in VLSI the chip computational limitation is largely due to finite device dissipation. The standard commercial epoxy and ceramic packages allow about 2-5 watts of steady state power dissipation.

### 3.2. Computational Aspects:

The computational aspects refer to the speed of computation and the message flow within the networks. The speed is estimated by computing the delay in interprocessor communications and the message density in the interprocessor links reveals the message flow.

### 3.2.1. Delay

The delay refers to the time needed in interprocessor communication to execute a computational task. This is both the property of the topology of the network and the length of interconnects. The average energy dissipation is measured as a product of the total chip area and the average delay.

### 3.2.2. Message Traffic Density

An important aspect of the computational power of the networks is the distribution of data flow within the networks. An efficient network should avoid the message traffic congestions at the links and should distribute the data (message) flow uniformly across all the available links. The message traffic density of at the links with respect to the networks size is a good measure of the computational power of the network. The average rate at which each node originates messages is assumed to be fixed at one message per time unit regardless of network size $N$. Also it has been assumed that the average rate at which any node, $i$ within the network transmits messages to another node $j \neq i$ in the network, is constant. The average message traffic density is defined as the product of number of processors ($N$) and the ratio of average number of nodes visited by a message to the total number of links in the network and is denoted by $\bar{M}$.

### 3.3. Cost Aspects:

The cost aspects consider the fabrication cost and the replacement cost due to poor reliability of the networks. The manufacturing cost of the IC is related to the total chip area (Assumption 2.4.2) and the regularity of the layout [22]. The reliability of the networks largely depends on the presence of long interconnects (Proposition 4) in the embedding and the topology of the networks. Depending on the existence of alternate message routes within the networks, the networks can fail completely or partially.

### 3.3.1. Yield

The yield of the IC is largely dependent on the total chip area. The occurrence of random spot defects will reduce the yield by a factor inversely proportional to the area, $A$ of the chip. This $O(1/A)$ factor is called the ***yield factor*** and is a measure of manufacturing cost of the IC. The defective processors can be replaced by redundant processors, but the chips with defects on the interconnects cannot be salvaged. The size of the interconnect is highly relevant for the chip yield (Proposition 4) and the presence of long wires will be enumerated for evaluation of the manufacturing cost.

### 3.3.2. Regularity

The regularity of the network largely decides the layout cost. Since all processors are identical, an $O(1)$ layout cost can be assumed for laying out a processor. Each link between the processors also can be made hierarchically, the actual cost to layout the links may be much less than the actual number of links. The ***regularity factor*** is a measure of layout cost and is defined as the ratio of total number of interconnections to the number of interconnections actually laid.

### 3.3.3. Fault-tolerance

The fault-tolerance capability largely decides the reliability of a working chip. Due to a host of causes, like electromigration, Kirkendall's effects, hot electron effects, etc., a processor or a link may fail during the normal use of the chip. Depending on the topology of the networks, the effect of failure of a single processor or a link will adversely affect the operation of the network. Normally the level of masking and processing associated with the interconnect is far more simpler than the processors and the reliability of the interconnect is higher than the reliability of the processor. So the reliability due to the processor failure and the interconnect failure are separated and different measures are used here. Since by Proposition 4, the probability of interconnect failure is directly proportional to its length, total length of the interconnects in the network is used as the measure of fault-tolerance due to interconnect failure and is denoted by $R_l$. The failure of a single processor will result in performance degradation because it may isolate one or more processors. The degradation in computing will be determined by the maximum number of connected processors (say $N'$) in the network due to the occurrence of a single processor failure. The value of $N'$ depends on the topology and the location of failed processor. The ratio of maximum value of $N'$ to the original size of the network is defined as the degradation factor, $\delta$ and is used as a measure of fault-tolerance. The reliability of the network can be improved by introducing redundant processors. The ratio of the reliability of redundant network to that of the non-redundant (original) network is defined as reliability improvement factor ($RIF$). Since the addition of redundant processors increases the chip area, the ratio of $RIF$ to the number of redundant processors (denoted by $\rho$) is also used here as a measure of fault tolerance. Overall fault-tolerance capability of three networks will be graded as **High, Medium** and **Low** by making a relative comparison of these three measures.

## 4. EVALUATION OF NETWORKS

### 4.1. Two Dimensional Meshes

The two dimensional mesh is shown in Figure 2. By Assumption 2.1.2, each processor is represented as a unit square and the interconnect length can be ignored. Thus both the area efficiency and the power efficiency of two dimensional meshes are approximately equal to 1.

In order to compute the delay in a $\sqrt{N} \times \sqrt{N}$ mesh, it should be noted that the message path length between two arbitrarily located processors at $(i, j)$ and $(k, l)$ within the square grids is given by the *city block distance* $d = |i-k| + |j-l|$. Thus the average message path length from a source processor is a function of its location $(i, j)$, assuming the lower leftmost processor is at $(0, 0)$. If the source processor is at any of the locations $(0, 0)$, $(n, 0)$, $(0, n)$ or $(n, n)$, ( where $n = \sqrt{N}$ ), then the average message path length is $\bar{d}_1 = n^{-2}[2*1+3*2+\cdots+n*(n-1)+(n-1)*n+\cdots+(2n-2)] = O(n) = O(\sqrt{N})$. If the source processor is at $(n/2, n/2)$, then the average message path length is $\bar{d}_2 = 4n^{-2}[2*1+3*2+\cdots+\frac{n}{2}*(\frac{n}{2}-1)+(\frac{n}{2}-1)*\frac{n}{2}+\cdots+(n-1)] = O(n) = O(\sqrt{N})$. For the source processor in any other position it can be shown that the average message path, $\bar{d}$, is $O(\sqrt{N})$ and satisfies the inequality $\bar{d}_2 < \bar{d} < \bar{d}_1$.

Assuming an $O(1)$ delay time (Assumption 2.2.1) associated with each processor, the average delay between the processors is $\bar{D} = O(\sqrt{N})$.

The total number of links in the square mesh is equal to $2\sqrt{N}(\sqrt{N}-1) = O(N)$ and the average message path is $O(\sqrt{N})$. Assuming all the $N$ nodes issue messages simultaneously, the average message traffic density is then $\bar{M} = NO(\sqrt{N})/O(N) = O(\sqrt{N})$.

Since the average delay is $O(\sqrt{N})$ and the chip size is $O(N)$, the average chip dissipation is $O(N^{3/2})$. Also the area efficiency is 1 and by Assumption 2.4.2, the yield is $O(1/N)$.

The layout can be constructed hierarchically and a block of $4^k$ processors can be laid in $k$-th step paying $2^{k+1}$ cost. Thus a network of size $N$ needs $\sum_{k=1}^{\log_4 N} 2^{k+1} = 2^{\log_4 N+2}-4 = O(\sqrt{N})$ cost. The regularity factor is thus $O(N)/O(\sqrt{N}) = O(\sqrt{N})$.

Since the network consists of nearest neighbor type connections, interconnect reliability is $R_l \approx 1$. The failure of a single processor does not impair the performance of the networks drastically. Due to the presence of many parallel paths in the square grids, the failure of a single processor results into isolation of the failed processor only and does not impair the performance of the networks drastically. The degradation factor is thus $\delta = 1/N$. If $R_p = e^{-\lambda_p t}$ is the functional reliability of each processor in the meshes, then the overall reliability of the network is $R_M(t) = R_p{}^N$. This reliability can be sufficiently ameliorated by adding a redundant row and the overall network can be made to be $(\sqrt{N}-1)$ fault-tolerant. The reliability of the redundant mesh network is $R_{rM}(t) = (R_p{}^n + n(1-R_p)R_p{}^{n-1})^n$ , where $n^2=N$. The reliability improvement factor, $RIF_M$ due to the redundant processors is given by $RIF_M = R_{rM}/R_M$ such that $\rho = (1/(2n-1))(1+n(R_p{}^{-1}-1))^n$.

The delay can be improved for mesh networks if the processors belonging to each column and each row are connected hierarchically as binary trees. Such networks are known in the literature as orthogonal tree networks [23], mesh of trees [24, 25] and orthogonal forests [26]. The average delay for such networks reduces to $O(\log N)$ but the chip area increases to $O(N\log^2 N)$. The overall performance thus does not improve. On the contrary, the presence of long interconnects of length $O(\sqrt{N})$ actually increases the average delay to $O(\log^2 N)$ (by Proposition 2). Moreover, these networks suffer from many practical limitations as poor yield (due to large chip size), poor regularity (due to presence of mesh and trees combined), $O(N\log N)$ cross-overs, long interconnects, etc.

### 4.2. Binary Tree Networks:

The complete binary tree networks of $N$ processors is shown in Figure 3 which needs at most $O(N \log N)$ layout area corresponding to $O(N)$ leaves and $O(\mathit{log}N)$ height of the tree. A better layout which needs optimal $O(N)$ area can be constructed using the concept of an H-diagram, originally proposed by Marihugh and Anderson [27] as a graphical approach to logic design. Horowitz and Zorat [28] have constructed the algorithms for the generation of such a layout and the modified network is henceforth referred as an **H-tree**. The H-tree layout of the binary tree is shown in Figure 4. The total area for a complete binary tree of $N$ processors can be computed from the following recursive relationship

$$A(N) = [2\sqrt{A(\lceil N/4 \rceil)} + 1]^2 \qquad \text{with } A(1) = 1.$$

Assuming $N = 2.4^k - 1$, it can be shown that $A(N) = 4^{k+1} - 2^{k+2} + 1 = 2N - 2.82\sqrt{N+1} + 3$.

Thus, the area efficiency is better than 0.5. The longest wire in the layout is of size $\sqrt{N}/2$ and the total length of wires in the layout is given by the recurrence relation:

$$L(N) = 4L(\lceil N/4 \rceil) + \sqrt{N} \qquad \text{with } L(7) = 1$$

which gives a solution $L(N) = O(\sqrt{N} \log N)$.

The power efficiency depends on $L(N)$ and the width of the wires and is more than 0.5.

The worst case delay occurs when a message is propagated between the leaves through the root of the tree. Assuming $O(l)$ delay for both metal wire (without driver) and polysilicon wire (with interspersed driver), the worst case delay can be given by $D_{max}(N) = 2^*[2\sum_{j=1}^{k}(2^{k-j}-1)+2k+1] = 2\sqrt{2(N+1)} - \log(N+1) - 2 = O(\sqrt{N})$. But this delay is smaller than in mesh network, since only $2\log N$ intermediate processors are visited as opposed to $2\sqrt{N}-1$ in mesh network. The average message delay between root and other processors can be given by $\bar{D} = \sum_{j=0}^{2k-1}\sum_{i=1}^{\lfloor j/2 \rfloor} 2^{j-2k}(2(2^{k-i}-1)+(j \bmod 2)(2^{k-\lfloor j/2 \rfloor -1}-1)+\mathit{log}\ j) = O(\sqrt{N})$.

To calculate the message traffic density on a link, consider $N-1$ time units during which $N(N-1)$ messages are generated and each node on the average will have sent a message to each of the others. Let $h = \log(N+1)$ be the height of the tree network, denoted as $T_h$, such that $|T_h| = N$. A subtree of $T_h$ at level $k$ from leaves is shown as $T_k$ such that $|T_k| = 2^k - 1$. A link between level $k$ and level $k+1 \leqslant h$ will be used to transmit messages between i) all the nodes of the left and the right subtrees each of size $|T_k|$ and ii) one subtree of size $T_k$ connected by the link and $(N - 2T_k)$ nodes of the tree, $T_h$ (Figure 5). Thus, the message density per unit time at a link between level $k$ and level $k+1$ is $M(k) = \frac{2}{N-1}[|T_k|^2 + (N - 2|T_k|).|T_k|)] \approx (2^h - 2^k)\left(\frac{2^k - 1}{2^{h-1} - 1}\right)$. Since M(k) is a monotonically increasing function of $k$, the maximum congestion occurs at the link between the root and its sons (i.e., $k = h-1$) which can be obtained by solving $\frac{\partial M(k)}{\partial k} = 0$. The total number of messages that pass through these links is $M(h-1) = 2^{h-1} \approx N/2 = O(N)$.

Since the area efficiency is less than 1, the yield is not likely as high as the meshes, but it is asymptotically equal to $O(1/N)$. The average energy dissipation of the chip is $O(N^{3/2})$.

The layout can be constructed hierarchically and each level of embedding needs $7 \times O(1)$ cost and the overall connection cost for a network of $N$ processors is equal to $O(\log_4 N)$. The total number of links in a binary tree is equal to $N-1$. Thus the regularity factor is $O(N/\log N)$.

The fault-tolerance capability of the network due to the failure of a single processor depends on the location of the processor. If the external communication is done solely through the root, its failure will have total disastrous effect invalidating the usability of the IC. Since there is no parallel path for message flow, failure of any links will truncate the operability of the chip. If any processor other than the root fails, will also reduce the performance of the IC by an amount depending on its location in the tree. The computational degradation that occurs due to a faulty processor or an interconnect at level $i$ from the leaf nodes can be defined as the number of processors which are eliminated due to the fault at level $i$ and is equal to $2^i-1$. If the external communication is made through leaf nodes, then $\delta = (N-1)/(2N)$. The network can be restored to function normally by replacing the defective processor by redundant processor. Redundant processors can be placed in the extra space available within the chip and re-routing can be done by electrically programmable routing technique. Since only $N - 2.8\sqrt{N+1} + 3$ space is available for laying out the redundant processors redundancy can be added for nodes till level 2 from the leaf nodes, i.e., level $h-2$ from the root. Thus the leaf processors and their fathers are not replicated and all other nodes in the tree are replicated at locations shown by * in Figure 4. If $R_p$ is the reliability of each processor then the overall reliability of the tree network without any redundancy is $R_T = R_p{}^N$. Clearly, $R_l = O(\sqrt{N} \log N)$. If the redundancy is added as described above, then the reliability of the redundant network is $R_{rT} = R_p{}^{3N/4} \times (2R_p - R_p{}^2)^{N/4}$. The reliability improvement factor, $RIF_T$, is given by $RIF_T = R_{rT}/R_T = (2-R_p)^{N/4}$ and $\rho = (4/N)(2-R_p)^{N/4}$. Thus the reliability of the tree network is poorer compared to the meshes.

### 4.3. Cube Connected Cycles:

An $m$ dimensional Cube Connected Cycle (CCC) is a network which can be derived from a boolean hypercube of $2^m$ vertices by replacing each vertex with a cycle of $m$ vertices. This was originally proposed by Preparata and Vuíllemin [7] to ensure that the degree of each vertex is bounded to 3 and not to $m$ as in an $m$-cube network. The topology of a 3-dimensional CCC with $3.2^3 = 24$ processors is shown in Figure 6 and its optimal VLSI layout has been given in Figure 7. It should be noted from Figure 7 that the layout of an $m$ dimensional CCC can be made by laying out $m$ $C_i$ routes in an $m$-cube [29] on a grid graph and replacing the vertices of $m$-cube by $2^m$ cycles of size $m$. Clearly, the maximum height of the cycle is $\sum_{i=0}^{m-1} 2^i = 2^m$ and there are totally $2^m$ cycles. Thus the total area required by an $m$ dimensional CCC is $2^{2m+1}$ assuming that the width of the edge is equal to the square root of processor area. Since $N = m2^m$, then $2^m = N / (\mathit{log}N - \log m)$, i.e., $m \approx \log(N/\mathit{log}N)$. Thus the total chip area is approximately $O(N^2/\log N^2)$. Thus both the area efficiency and the power efficiency are $O((\log^2 N)/N)$.

Let the interprocessor link in the cycle be called ring link (vertical lines in Figure 7) and the interprocessor link between two adjacent cycles be called vertex link (horizontal lines in Figure 7). Using Wittie's algorithm for message routing, it can be shown that on the average a message traverses $m/2$ vertex links and $(5m/4)-2+2^{1-m}$ ring links if $m$ is even (and an additional $1/(4m)$ ring links if $m$ is odd). It may be noted that Wittie's average path length analysis by his routing algorithm was incorrect and the correct result is stated above. Wires are either horizontal or vertical and cannot be both metal (Assumption 2.1.4). Thus in order to reduce the propagation delay, it is needed that the cycle links should be made of metal and the vertex link should be made of polysilicon (or diffusion). It may be noted that in an $m$ dimensional CCC, there are $m\,2^{m-1}$ vertex links having a total interconnection length of $2^{m-1}\sum_{i=1}^{m} 2^i$ so that average vertex length is $(2^{m+1}-2)/m$. Thus the average vertex delay is $4(2^m-1)$ i.e., $O(N/\log N)$. The asymptotic average delay over metal ring link is $O(\log N)$. The worst case delay is due to message transmission between two processors at the opposite edges of the chip and is proportional to the perimeter of the chip, i.e. $O(N/\log N)$. Since each node has degree 3, the total number of links is equal to $3N$. If $N$ messages are generated on the average in unit time, then the average message density is equal to $\overline{\mathrm{M}} = (N/3N) \times O(m) = O(\mathit{log}N)$.

Since the chip area is very large, the yield is $O(\log^2 N/N^2)$, which is very poor compared to the mesh and the tree networks. The average energy dissipation of the chip is $O(N^3/\log^3 N)$.

The layout can be partially constructed hierarchically. An $N(=m\,2^m)$-node CCC composes of two $(m-1)2^{m-1}$-node CCC and $2^{m-1}$ connections as is evident from Figure 7. Thus the layout cost to construct an $N$-node CCC is $2^m+m-1 = O(N/\log N)$. Since the total number of connections in the layout is equal to $1.5N$, the regularity factor is $O(\log N)$ and is poor compared to the mesh and the tree networks.

The fault tolerance capability of CCC is good because there is always an alternate path to re-route the message like in the mesh. Thus the failure of a single processor will not have any drastic effects on the performance of the network and $\delta = 1/N$. But due to presence of many long interconnects and large chip area, the reliability is not as good as the meshes. Total length of interconnect within the chip is $O(N^2/\log^2 N)$ due to the presence of $N/\log N$ cycles of average height $N/\log N$. Thus $R_l = O(N^2/\log^2 N)$. The reliability of the network due to processor failure can be given by $R_{CCC} = R_p{}^N$. If one redundant processor is added to each cycle, then the reliability improves to $R_{rCCC} = (R_p{}^m + m(1-R_p)R_p{}^{m-1})^{N/\log N}$. The reliability improvement factor, $RIF_{CCC}$, is given by $RIF_{CCC} = R_{rCCC}/R_{CCC}$ and $\rho = (1/m)(1+m(1/R_p-1))^{N/\log N}$.

## 5. COMPARISON OF THREE CLASSES OF NETWORKS

From the analyses done in the previous section, it is evident that each network has certain strong aspects and certain weak aspects. It is difficult to relate all these aspects by a compact formula which can be utilized as a performance metric. A weak effort in this respect was originally done by Mead and Rem [14] relating the area ($A$) and the speed ($T^{-1}$) and proposing the *rental time* of the chip $AT$ as a metric. Thompson [8] has extended the concept for any arbitrary network by showing that $AT^2$ indicates a better performance metric. He has related the area and the speed of computation in a network through its *minimal bipartition width*, $\omega$ and have shown that a computational problem can be solved by exchanging information over $\omega$ such that the speed of computation is directly proportional to the cardinality of $\omega$ while the area of planar implementation of the network is directly proportional to the square of the size of $\omega$. But this metric accounts for the lower bound of the chip area. Savage [30] has contended that $A^2T$ reflects a better evaluation for certain computational problems like binary sorting. From all these contradictory claims, it is evident that it is virtually not possible to correlate all the criteria discussed in section 3. An alternative strategy has been adopted here. The networks have been given credit points for each criterion depending on their relative merits and the total points have been used as a performance index for the network. The results of the evaluation with respect to different criteria have been shown in Table 1. The credit points have been assigned on the basis of relative merits of the networks. From the values of total points, it can be seen that the two dimensional mesh networks indicate overall better performance than the H-tree and the CCC. This is in direct contrast to the results of Wittie [3], Siegel [31, 9], etc., who have concluded that fast networks like CCC, PSN, spanning bus hypercubes, etc., have better overall performance. It may be argued whether it is appropriate to give same weight to all the criteria. But it can be easily seen that the conclusion remains true even if different weights are ascribed to three orthogonal aspects (i.e. criteria having same aspect only has same weight).

## 6. CONCLUSION

The basic conclusion from this paper is that the Cellular Networks which have a similar structure to the mesh can be cost effectively implemented for VLSI implementation and are highly suitable for VLSI parallel processing. The penalty in delay can be offset by the gains of several criteria discussed in this paper. The faster topologies like CCC, PSN, tree, etc., do not provide an overall good performance because of long interconnects which introduce high delay and large chip area which reduces the chip efficiency and the chip yield. A multilayered model with more than one layer of metal will improve the performance of these topologies and a judicial laying out is necessary to reduce the chip area under the multilayered model.

## REFERENCES

[1] K. J. Thurber, "Interconnection networks - A survey and assessment," *AFIPS Conference Proceedings*, vol. 43, pp. 909-919, 1974.

[2] G. A. Andersen and E. D. Jensen, "Computer interconnection structures: taxonomy, characteristics and examples," *ACM Computer Survey*, vol. 7, pp. 197-213, December 1975.

[3] L. D. Wittie, "Communications structures for large networks of microcomputers," *IEEE Transaction on Computers*, vol. c-30, pp. 264-273, April 1981.

[4] Tse-yun Feng, "A survey of Interconnection Networks," *IEEE Computer*, vol. 14, pp. 12-27, December 1981.

[5] D. F. Barbe, *Very Large Scale Integration: Fundamentals and Applications*. Springer-Verlag, 1980.

[6] M. J. Atallah, *Algorithms for VLSI networks of processors*. PhD Thesis, The Johns Hopkins University, 1983.

[7] F. P. Preparata and J. Vuillemin , "The Cube Connected Cycles: A versatile network for parallel computation," *Proceedings of 20th Annual IEEE Symposium on Foundations of Computer Science* , pp. 140-147, 1979.

[8] C. D. Thompson, *A complexity theory for VLSI* . PhD Thesis, Carnegie-Mellon University, 1980.

[9] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Transaction on Computers*, vol. c-28, pp. 907-917, December 1979.

[10] R. Aleliunas and A. L. Rosenberg, "On embedding rectangular grids in square grids," *IEEE Transaction on Computers*, vol. C-31, pp. 907-913, September 1982.

[11] C. E. Leiserson, *Area-Efficient grph layouts (for VLSI)*. PhD Thesis, Carnegie-Mellon University, 1981.

[12] C. Y. Lee, "An algorithm for path connection and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 346-365, September 1961.

[13] A. L. Rosenberg, "The Diogenes approach to testable fault-tolerant arrays of processors," *IEEE Transaction on Computers*, vol. C-32, pp. 902-910, October 1983.

[14] C. A. Mead and L. A. Conway, *Introduction to VLSI systems*. Addison, 1980.

[15] C. Mead and M. Rem, "Minimum propagation delays in VLSI," *IEEE Journal of Solid State Circuits*, vol. SC-17, pp. 773-775, August 1982.

[16] V. Ramachandran, "Driving many long parallel wires ," *Proceedings of 23th Annual IEEE Symposium on Foundations of Computer Science*, pp. 369-378, 1982.

[17] B. T. Murphy, "Cost-size optima of monolithic integrated circuits," *Proceedings of IEEE*, pp. 1537-1545, December 1964.

[18] G. E. Moore, "What level of integration is best for you?," *Electronics*, pp. 126-130, February 1970.

[19] R. B. Seeds, "Yield, economic and logistic models for complex digital arrays," *IEEE Int. Convention Rec.*, pp. 60-61, March 1967.

[20] T. E. Mangir, *Fault-tolerant design for VLSI design: Effect of interconnect requirements on yield improvement of VLSI design*. PhD Thesis, University of California, Los Angeles, 1981.

[21] C. H. Stapper, "Modeling of integrated circuit defect sensitivities," *IBM Journal of research and development*, vol. 27, pp. 549-557, June 1983.

[22] C. H. Sequin, "Managing VLSI complexity: an outlook," *Proceedings of the IEEE*, vol. 71, pp. 149-166, 1983.

[23] D. Nath, S. N. Maheswari, and P. C. P. Bhat, "Efficient VLSI networks for parallel processing based on orthogonal trees," *IEEE Transaction on Computers*, vol. c-32, pp. 569-581, June 1983.

[24] J. D. Ullman, *Computational Aspects of VLSI*. Computer Science Press, 1984.

[25] F. T. Leighton , "A layout strategy for VLSI which is provably good," *Proceedings of the 14th Symposium on Theory of Computation*, pp. 85-98, 1982.

[26] P. R. Cappello and K. Steiglitz, "Area-efficient VLSI structures for multiplying at clock rate," Technical Report #289, Princeton University , 1981.

[27] G. E. Marihugh and R. E. Anderson, "The H diagram: a graphical approach to logic design," *IEEE Transaction on Computers*, pp. 1192-1196, 1971.

[28] E. Horowitz and A. Zorat, "The binary tree as an interconnection network: applications to multiprocessor systems and VLSI," *IEEE Transaction on Computers*, vol. c-30, pp. 247-253, April 1981.

[29] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill Book Company, 1984.

[30] J. E. Savage, "Planar circuit complexity and the performance of VLSI algorithms," *Proceedings of the CMU Conference on VLSI Systems and Computations*, pp. 61-67, 1981.

[31] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Transaction on Computers*, vol. c-26, pp. 153-161, February 1977.

**Table 1: Evaluation of Three Static Networks**

| Evaluation of Meshes, H-Tree and CCC for VLSI application with respect to Physical, Computational and Cost Aspects | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| NETWORK STRUCTURE | EVALUATION CRITERIA AND PERFORMANCE | | | | | | | OVERALL RESULTS |
| | Average Message Delay | Average Message Density | Area Efficiency | Power Consumption | Chip Yield Factor | Layout Regularity Factor | Fault Toler -ance | |
| MESHES | $O(\sqrt{N})$<br>2 | $O(\sqrt{N})$<br>2 | 1<br>3 | $O(N^{3/2})$<br>3 | $O\left[\frac{1}{N}\right]$=Max<br>3 | $O(\sqrt{N})$<br>2 | High<br>3 | 18 |
| H-TREE | $O(\sqrt{N})$<br>3 | $O(N)$<br>1 | <1<br>2 | $O(N^{3/2})$<br>2 | $O\left[\frac{1}{N}\right]$<Max<br>2 | $O(\frac{N}{\log N})$<br>3 | Low<br>1 | 14 |
| CCC | $O\left[\frac{N}{\log N}\right]$<br>1 | $O(\log N)$<br>3 | $O\left[\frac{\log^2 N}{N}\right]$<br>1 | $O\left[\frac{N^3}{\log^3 N}\right]$<br>1 | $O\left[\frac{\log^2 N}{N}\right]$<br>1 | $O(\log N)$<br>1 | Medium<br>2 | 10 |

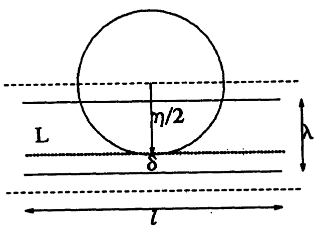


Figure 1: Occurrence of Circular Defects of diameter $\eta$

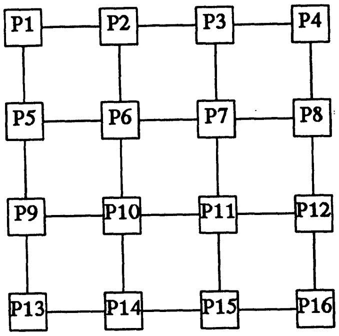


Figure 2 : A 4 x 4 Mesh Network

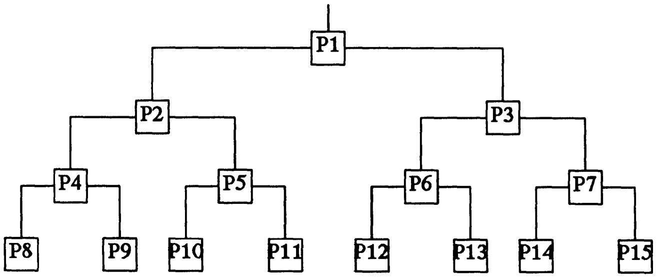


Figure 3: Layout of a Binary Tree

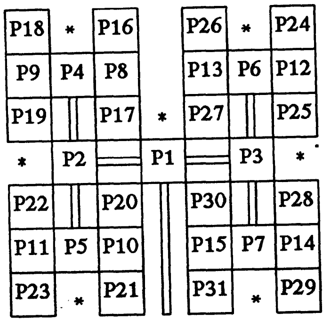


Figure 4: H-Tree Layout

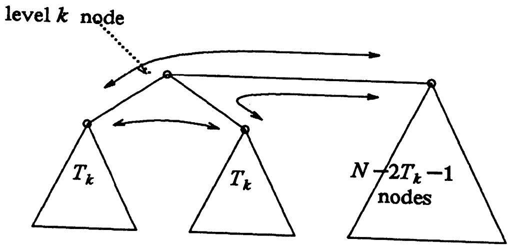


Figure 5: Message Flow through a level k node in a Tree Network

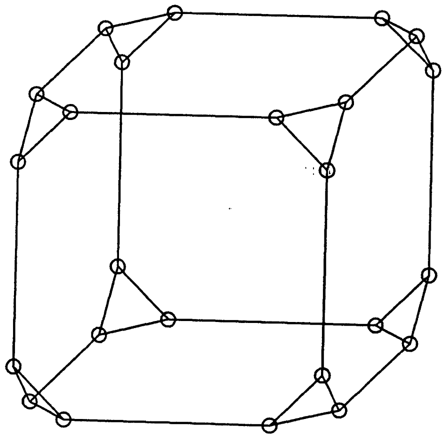


Figure 6: Cube Connected Cycle topology for 3. $2^3$ processors

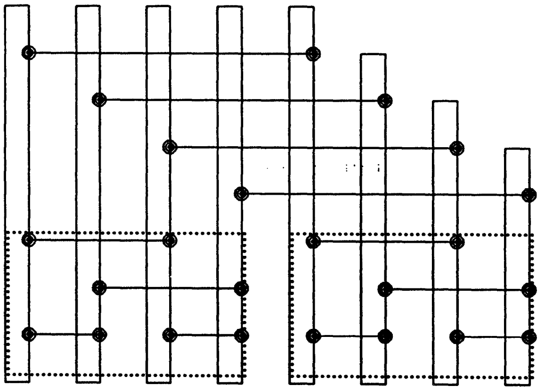


Figure 7: Layout of CCC with 3. $2^3$ processors

# On Designing Interconnection Networks For Multiprocessors

Mark A. Franklin and Sanjay Dhar

Center For Computer Systems Design
Washington University
St. Louis, Missouri 63130

## ABSTRACT

This paper considers various physical constraints which influence the design of VLSI based interconnection networks used in multiprocessor systems. Design expressions are presented for implementing an $N \log N$ packet passing interconnection network composed of circuit switched crossbar chip modules. Expressions reflecting chip level and board level pin and area constraints are derived and used to determine the network delay expected at a given clock frequency. Logic and memory delay, signal path delay, clock skew and clock tree delay parameters are defined and used to determine the maximum frequency which can be obtained with a given design. An example 2048x2048 network design is considered.

## 1. Introduction and Overview

The design of effective multiprocessor systems involves numerous interacting elements ranging from parallel algorithms to programming languages to computer architecture. This paper focuses on the computer architecture question and, in particular, on the design of VLSI based electrical interconnection networks for use in multiprocessor systems. Due to their potentially critical effect on overall multiprocessor performance, interconnection networks have been widely studied. Various studies have focused on their functional properties (permutations, control algorithms) [2,9,14,18], their complexity and performance [5,16,17,23], and their actual design [7,20].

One way of characterizing interconnection networks relates to the style of multiprocessor system in which they are used. At one end of the spectrum are systems where the interconnection network is the central communications component between the processors (as in a message passing system) or between processors and the main memory (as in a shared memory system). The NYU Ultra computer [10], the related IBM RP3 [19] computer and the BBN Butterfly [3] are examples of this style of design. It is convenient to refer to these systems as NETWORK CENTERED multiprocessors since the network is a central resource which has a major effect on overall system performance. This system style encourages viewing memory access and communications interchange in a uniform manner with costs associated with access and communications being roughly independent of physical location. (We ignore questions of cache and processor local memory here. If most accesses are local, then the need for a large and complex interconnection network is questionable). Algorithm development in this environment tends to encourage the use of large data structures which span the memory space and develop processor access patterns which are relatively uniform in nature.

At the other end of the spectrum are systems where the processors are embedded within the network itself with direct communications taking place primarily on a physically local basis. The Intel Cosmic Cube [22], the Blue Chip computer and various tree machines [24,4,11] are examples of this style. These systems are PROCESSOR CENTERED in that processor performance is a main determining factor in overall performance. This system style encourages viewing the world as being made up of local processing niches with data exchange between processors becoming more costly as one reaches further from the local niche. Algorithms which can take advantage of memory access and communications locality associated with a given interconnection structure can operate effectively in this environment.

In the middle of these two ends of the multiprocessor spectrum are machines where some (typically small) portion of the processing task is allocated to the interconnection network. The idea here is that since data must pass through the interconnection network (and be delayed) anyway on its way to and from memory, why not do some processing on the way.

This paper considers the design of large (several thousand inputs and outputs) VLSI based interconnection networks for the case of NETWORK CENTERED multiprocessor systems. The emphasis is on the physical design aspects of the network, and the implications of design constraints on network implementation and performance. A simple modified packet switched $N \log N$ multistage interconnection network topology is assumed, with each node being implemented as a crossbar switch having a limited amount of packet buffering at its input. Interconnection network designs using optical techniques are not considered though they may well serve as the basis for networks of the next generation.

The next section presents overall network topology and operation. Section 3 considers pin and area limitations at both the chip and board levels. Section 4 presents simple models of overall network delay as a function of clock frequency. Section 5 derives data rate equations for the network in terms of various design parameters such as logic, memory, and data path delays. Clock skew and distribution factors are included in the model. The model presented can be used to determine the maximum clock rate achievable and hence the expected delay through the network. Section 6 explores an example design of a 2048 input/output switch. The conclusion indicates that, in the current design environment, a large NETWORK CENTERED shared memory multiprocessor is likely to encounter a large performance penalty when accessing memory on the other side of the switch.

## 2. The Overall Network

The overall network topology considered is of the Boolean hypercube variety as shown in Figure 1. All switch modules are the same and are designed to be crossbar (CB) switches sized so that each one fits entirely on a single chip. Earlier studies have indicated that from an area-time performance viewpoint there is little difference between using an $N \log N$ versus a CB network within a chip [7]. Across the network as a whole, however, use of a Boolean hypercube structure is significantly less costly in terms of the total number of chips required [8].

While each switch module (being a CB) is nonblocking, the network as a whole is a blocking network. The number of stages in the network is $\log_N N'$ where $N'$ is the size of the overall network and $N$ is the size of the CB on a chip (see Table 1 for various definitions). Since the blocking probabilities of the network decrease as the number of stages decrease, it is advantagous to place as large a CB as possible in each switch module. This is shown in Figure 2 which contains a plot of blocking probability versus number of stages for a network of size 4096 (based on the formula derived in [17]). Note that reducing the number of stages from 5 to 3 decreases the blocking probability by about 10%.

* This research has been sponsored in part by funding from NSF Grant DCR-8417709 and ONR Contract N00014-8D-C-0761.

0190-3918/86/0000/0208 $01.00 © 1986 IEEE

Note also that the length of off-chip signal lines generally increases as the number of stages increases. Thus, for example, the maximum line lengths between the first and second stages in the network of Figure 1 are shorter than those between the second and third stages. This is important because of the delays associated with driving these lines. Thus, in general, reducing the number of stages reduces the off-chip delays.

Overall network throughput can be increased by using a modified packet switched rather than a pure circuit switched design. A modified packet switched design places a limited number of packet buffers at the input to each of the switch modules. However, within the module circuit switching occurs. Thus, a packet holds an entire path within each switch module as it passes through that module. On leaving the module the path is released for use by some other packet. Increases in throughput which result from a packet switching approach have been discussed elsewhere [5]. These studies also indicate that most of the potential gain from buffering is achieved with a limited number of buffers (about 4) on each switch module input. For simplicity, in our discussion here we assume a single packet buffer. However, this is not critical to the analysis which follows. We also assume fixed size packets of length 100 bits which is about what is needed to include sufficient bits for data, memory module address, intra-memory module address, and return processor address.

Figure 3 shows an example path through a three stage network. In addition to the input packet buffer, a single bit buffer has been placed at each CB output to allow for a limited pipelining capability. Notice that a dotted line is present in the diagram around the input buffers. This indicates the presence of a cut-through mechanism which permits packets to stream through the switch module without going through a buffer filling process if the down stream switch module it requires has an empty input buffer. Under light loading conditions this will allow packets to pass from one switch module to another without being slowed down by buffer fill times [12].

| VARIABLE NAME | TYPICAL VALUE | DEFINITION |
|---|---|---|
| $N'$ | 2048 | Size overall network |
| $N$ | 16x16 | Size crossbar module (NxN) |
| $N_p$ | 250 | # pins on module chip |
| $N_k$ | 40 | # pins on module chip for ctrl, clk, power |
| $W$ | 1,2,4,8 | # of lines in a data path |
| $P$ | 100 | Packet size in bits |
| $F$ | 20,40,60,80 | Clock frequency (MHz) |
| $T$ | 2 | Packet time through ntwk ($\mu$sec) |
| $M_{su}$ | 2 | # clock cycles to perform chip setup functions (DMUX/MUX) |
| $V_{dd}$ | 5 | Power supply voltage (volts) |
| $\Delta V_{\max}$ | 0.50 | Max. variation in power supply voltage (volts) |
| $Z_0$ | 50 | Line driver impedance (ohms) |
| $L$ | 5 | Pin inductance (*nano* henries) |

Table 1: Variable Name Definitions

The broad design philosophy taken is to keep the switch as simple as possible since simplicity generally leads to speed. In this spirit, combining networks and network information processing other than routing is omitted. Hot spots [20] and other problems are assumed either to be dealt with by the operating system, or are accepted and result in performance penalities which hopefully are partially overcome by having a fast network. The network is thus of the RISC (Reduce Interconnection System and Complexity) style of design.

## 2.1. Network Control

The general design methodology employed utilizes clocked as opposed to asynchronous control. Other studies have shown that given today's design environment, system sizes and data rates, clocked designs adequately meet most performance requirements [6,25]. At very high clock rates, multiple clocked approaches will need to be considered, however they are not treated here [1]. We assume that a two phase clock is used and that two pins on each chip are allocated for this purpose.

A complete interconnection network requires control provision for:

- path establishment and data transfer
- detection of a blocked path
- indication of end of transmission
- path clearing

Packets are self routing, moving from CB chip to CB chip according to address bits in the header portion of the packet. Within each CB the entire path is held from chip input to output. Feeding back from each input buffer to the associated output of the preceding chip is a buffer full line. This indicates whether the buffer is full and thus the path is blocked at that point. Packets are backed up and held in prior buffers until the buffer full line indicates transfer can proceed. For each $N$x$N$ CB chip there are thus $2N$ buffer full control lines, $N$ of these indicating whether its buffers are full, and $N$ indicating whether the buffers downstream are full. All packets proceed in lock step from stage to stage in the network.

Given fixed length packets, an on-chip counter can determine when packets have completed transmission. Path clearing will occasionally be necessary when certain error conditions occur. We assume that such clearing operations constitute a type of network reset and that such a drastic action will be initiated by some master processor for the network as a whole. One pin per chip is allocated for this purpose. Thus, ignoring power and ground for the moment, $2N+3$ control lines are needed per chip.

## 2.2. Crossbar (CB) Design

There are many ways of designing CB switches. In this paper we consider two approaches (see Figures 4a and 4b). The first is referred to as the MESH CONNECTED CROSSBAR (MCC) design [16]. In this design, $N^2$ two by two crosspoint switches are placed on the chip, with each switch having a packet routing capability and one bit of buffering to allow for limited pipelining. Packet routing is thus completely local, the layout is planar and the distance between adjacent switches constant. The design is modular and as technology improves larger on-chip CBs can be easily implemented by replicating the basic switch crosspoints. The area of the entire CB grows as $O(N^2)$ while the time delay grows as $O(N)$ .

The second design approach is referred to as the DMUX/MUX CROSSBAR (DMC) design [13]. In this case, after $\log_2 N$ bits have arrived at an input port, an input port controller (IPC) (i.e. a demultiplexer) determines on which output line to route the arriving message. The IPC also signals an output port controller (OPC) (i.e. multiplexer) which selects between multiple input packets that request the same output port. An $N$x$N$ DMUX/MUX crossbar would require $O(N^2)$ two input gates and have a time delay of $O(logN)$ gate delays. In addition to gate delays, however, there are the line delays associated with the potentially long on-chip lines between the input and output ports and controllers. This is due to the fact that path topology from input to output represents a complete bipartite graph whose layout area grows as $O(N^4)$. Certain results show that the overall delay with this type of CB grows as $O(N^2)$ [16].

## 3. Pin and Area Constraints

A serious constraint in the physical design of large networks is imposed by the limited number of signal pins available both at the chip and board levels. In this section these pin limitations are explored assuming a Pin Grid Array (PGA) packaging technology that is aggressive, but currently realizable, and board edge connectors of standard high performance design.

### 3.1. Chip Pin Constraints

Pin usage can be broadly grouped into three categories: data pins ($N_{p.d}$), control pins ($N_{p.c}$), and ground and power pins ($N_{p.g}$). The total number of pins on a chip, $N_p$, is thus given by:

$$N_p = N_{p.d} + N_{p.c} + N_{p.g} \tag{3.1}$$

Since the size of the subnetwork on a chip is $N$x$N$ and $W$ is the data path width, the number of data pins is:

$$N_{p.d} = 2WN \tag{3.2}$$

The control information required for setting up paths in the network is obtained as part of the data and hence requires no extra signal pins. However, for each input or output port, a control signal indicating the state of the buffer (full or not) is required. This necessitates $2N$ pins for the buffer full signals. In addition, we allocate two pins for a two phase non-overlapping clock and a pin for system reset. Hence:

$$N_{p.c} = 2N + 3 \tag{3.3}$$

Normally, when small circuits are considered, especially those that have a small number of signal pins, the use of a single pin for power and a single pin for ground is sufficient. However, for large chips, especially those that have a large number of signals all of which can switch at the same time, it may be necessary to allocate more pins for power and ground in order to maintain ground and power voltage variations to within acceptable limits.

Each signal pin has an associated inductance. When a signal switches between its high and low voltage states, the change in current through this inductance causes a voltage to appear across the pin. As the number of signal pins increases, in the worst case, all of them can switch in the same clock cycle, thus causing a large voltage change of either the ground or the power net. Given the impedance of the pin driver circuit and assuming that this circuit itself is driven by an exponential driver, the rate of change of supply current with respect to time for a single pin can be obtained (by analytic or simulation methods). For a 50 ohm driver driving a 30 inch metal pc board line with typical characteristics, the rate of change of current with respect to time, $di/dt$, is about $7*10^6$ amps/second. The following expression can be used to determine the number of power and ground pins, $N_{p.g}$, needed for a given number of output signal pins, $N(W+1)$, maximum permissible voltage variation, $\Delta V_{\max}$, and pin inductance, $L$.

$$N_{p.g} = \frac{2N(W+1)L}{\Delta V_{\max}} \left( \frac{di}{dt} \right)_{avg} \tag{3.4}$$

The expressions 3.2, 3.3 and 3.4 can now be substituted into 3.1 and the overall number of pins, $N_p$ determined. Using the parameter values indicated in Table 1, Table 2 indicates $N_p$ as a function of subnetwork size and path width. Assuming that the maximum number of pins available on a chip is 240, the portion of the table to the left and top of the heavy lines indicate designs that satisfy the pin constraints.

### 3.2. Chip Area Constraints

We now consider constraints on chip area and obtain the size of the largest network that can be implemented in a single chip. As mentioned earlier, maximizing the size of the CB subnetwork (residing entirely on a single chip) reduces blocking and hence is desirable. The two designs introduced earlier (i.e. the mesh connected CB, MCC, and the demultiplexer/multiplexer CB, DMC) are explored. The development follows that of Padmanabhan [16].

For the case of the MCC design the key element in estimating the area lies in estimating the area occupied by a two input, two output switch. These switches can then be connected in a mesh to form the CB network. Padmanabhan gives a PLA implementation of this switch with a one bit wide data path. He shows that the implementation would occupy a rectangular area of dimensions approximately 100 lambda by 100 lambda. Assume that the above implementation gives the area of the control logic of a switch with a $W$ bit data path. The area occupied by the data path must now be estimated. The data path consists of $W$ lines traversing the switch from left to right and from top to bottom. In addition, $W$ control lines for each set of data lines must be routed. Assuming that separation between lines, including area for driving and control buffers, is 10 lambda, the dimensions of the rectangular area occupied by the switch becomes $100 + 20W$. Hence the area of the MCC realization consisting of $N^2$ switches is:

| | N - Crossbar Subnetwork Size | | | | |
|---|---|---|---|---|---|
| W | 16 | 18 | 20 | 22 | 24 |
| 1 | 72 | 80 | 89 | 97 | 106 |
| 2 | 106 | 119 | 132 | 144 | 157 |
| 4 | 174 | 196 | 217 | 239 | 260 |
| 8 | 311 | 350 | 388 | 427 | 465 |

Table 2: The number of pins per chip necessary, $N_p$, for different values of subnetwork size $N$ and data path width $W$.

$$A_{\text{MCC}} = N^2(100 + 20W)^2 \tag{3.5}$$

Next consider the area occupied by the MUX/DMUX realization. The area occupied by such an implementation can be broadly divided into the area occupied by the $N$ 1-to-$N$ demultiplexers and $N$ $N$-to-1 multiplexers, and the area occupied by the $WN^2$ wires which must be routed from the multiplexers to the demultiplexers. The routing of the wires from the demultiplexers to the multiplexers will be done according to the routing presented by Wise [26] which results in identical wire lengths. Let the minimum separation between wires be $d$ and the vertical separation between consecutive wire origins and endings be $h$. The area occupied by such a routing can be shown to be given by:

$$A_{\text{wire}} = (N-1)^4 \frac{h^2W^2d}{\sqrt{4h^2 - d^2}} \tag{3.6}$$

The minimum area is occupied when $h = d$ which results in:

$$A_{\text{wire}} = (N-1)^4 \frac{(Wd)^2}{\sqrt{3}} \tag{3.7}$$

Next estimate the area occupied by the $N$ demultiplexers and $N$ multiplexers. Assume that the demultiplexers are implemented as a binary tree of 1-to-2 demultiplexers. A 1-to-2 demultiplexer with $W$ data bits occupies a rectangular area of dimensions $30W$ by 24 [16]. A tree realization will have a maximum of $N/2$ demultiplexers in the first stage with $\log_2 N$ stages. Thus, the bounding box of this tree is given by $360WN\log N$. The area occupied by $N$ demultiplexers and $N$ multiplexers (assuming a multiplexer occupies the same area as a demultiplexer) is given by:

$$A_{\text{dmux/mux}} = 360WN^2\log N \tag{3.8}$$

The total area of the MUX/DMUX design is given by:

$$A_{\text{DMC}} = (N - 1)^4 \frac{(Wd)^2}{\sqrt{3}} + 360\,WN^2 \log N \qquad (3.9)$$

The area expressions of (3.5) and (3.9) are lower bound estimates. Table 3 gives the maximum size network that can be implemented in a chip satisfying the above area constraints assuming that these estimates are increased by a third (to handle line drivers, etc.). The maximum chip dimensions assumed are 1cm by 1cm with a $\lambda = 1.5\ \mu m$.

Examination of Table 2 shows that the largest network that can be implemented in a chip satisfying the pin constraints is 22x22 with a 4 bit data path. Table 3 indicates that the area constraints limit the network size to about an 18x18 network with the DMC design and a 25x25 network with the MCC design (assume a 4 bit data path). However, we choose a convenient power of 2 network size of 16x16 with a 4 bit data path as our basic subnetwork to be implemented in a chip, satisfying both pin and chip area constraints. Though not detailed here, a CMOS power analysis of such a chip indicates that about 2.5 watts will be dissipated at 50 Megahertz. We next investigate the layout of the network at the board level.

|   | Sub network size $N$ |   |
|---|---|---|
| W | MCC | DMC |
| 1 | 37 | 34 |
| 2 | 32 | 25 |
| 4 | 25 | 18 |
| 8 | 17 | 13 |

Table 3: The largest subnetwork that can be implemented on a 1cmx1cm chip.

### 3.3. Board Area Constraints

Assume that the pin layout on the chip package uses a pin grid array with three rows of pins having a separation between adjacent pins of 100 mils. The size of a package with at least 175 pins is about 2 inches on a side. The use of a 16x16 subnetwork makes a 256x256 network reasonable for implementation on a single board. This network has two stages, each stage consisting of 16 chips. If a stage of 16 chips is lined up on a single side along a board edge then the board length will be about 32 inches. The routing of wires between these two stages will determine the width of the board occupied by the 256x256 network.

To estimate the area occupied by the routing assume that the routing strategy is similar to that adopted for the chip level DMC implementation. The routing in this case is identical to the DMC implementation of a 16x16 crossbar network. The parameters $h$ and $d$ must, however, be estimated differently. $d$ still remains the minimum separation between wire at the board level with a typical value of 50 mils. Assume the board has two signal layers. The total number of wires to be routed is $N^2(W + 1) = 1280$. Each layer then has to route half the total number of wires, that is, 640 wires. Hence the vertical separation between wires is $h = 32000/640 = 50$ mil, the minimum. The area occupied by the wires is then obtained by evaluating (3.7) as 73 square inches. Thus, the width of the layout of the wires is about 3 inches and the longest path on the board is then no more than $32 + 3 = 35$ inches. This layout will be used in Section 6 in examining the design of a 2048x2048 network.

### 3.4. Board Pin Constraints

At the board level, consideration must be given to the routing of signal wires from one board to the next. Implementing a 256x256 network with a 4 bit data path requires the routing of 1280 wires on the input and output sides. In the last section it was shown that the layout of the chips required each edge of the board to be about 32 inches long. If the wires were brought out to the edge of the board on two layers, it was shown that the separation between wires would have to be 50 mil. This is about the minimum separation between wires on the board that keeps crosstalk among wire to acceptable levels. Commercially available connectors are able to connect up to 100 lines from one side of a board and are no more than 4 inches long. Thus with connectors using both sides of the board, eight connectors can be used for the entire 1280 lines which can be lined up on one edge of the board. Thus the pin constraints at the board level can be satisfied.

## 4. Network Delay

In this section expressions are presented for the time to transfer a packet through the network ($T$). This is a one way network delay time and doesn't include memory access time. A best case model is presented in which a lightly loaded network is assumed with no blocking of packets. Packets, in this situation, can thus stream through the entire network from the input to the output being delayed only by chip module setup and pipeline fill times.

For the case of the MCC design, network delay is composed of two components. The first is the time to fill the pipe of crosspoint switches from input to output of the network. The second is the time it takes to transfer the packet once it has arrived at the end of the network. Thus:

$$T_{MCC} = \textit{pipeline fill time} + \textit{packet transfer time} \qquad (4.1)$$

In this design the average number of crosspoint switches per chip that a packet passes through is $N$. The number of stages the packet traverses is $\lceil \log_N N' \rceil$. Thus, the pipeline fill time is $N \lceil \log_N N' \rceil$. The number of bit times associated with a packet transfer is the packet size ($P$) divided by the path width ($W$). Thus, the packet delay time is:

$$T_{MCC} = (N \lceil \log_N N' \rceil + P/W)(1/F_{MCC}) \qquad (4.2)$$

In the DMC design, associated with routing the packet within each chip is a setup time. This time is dependent on the size of the on-chip CB in that at least $\lceil \log_2 N \rceil$ bits must be received by the chip before the path can be established. Given a path width of $W$, the number of clock periods, $M_{ss}$, associated with this setup time for a single chip (or stage) through which the message passes is thus:

$$M_{ss} = \lceil \log_2 N / W \rceil \qquad (4.3)$$

To break up long path delays, this design also assumes that a 1-bit buffer is present at the output of each chip. This acts as a $\lceil \log_N N' \rceil$ pipeline through the network. As in the MCC case, a packet transfer time is also present to account for the time it takes the packet to leave the network when its first bit starts to leave an output port. The resulting overall time is given by:

$$T_{DMC} = \textit{setup time} + \textit{pipe fill time} + \textit{packet transfer time} \qquad (4.4)$$

$$T_{DMC} = (\ [M_{ss} + 1] \lceil \log_N N' \rceil + P/W\ )(1/F_{DMC}) \qquad (4.5)$$

These network delay expressions have been evaluated and are presented in Table 2. Notice that the results indicate that, even at fairly high clock frequencies, the one way delay through the network is substantial when compared to typical memory cycle times. For example, in the DMC model operating at 40 MHz with a path width of 2, the one way network delay is 1.48 $\mu$seconds. Round trip delay, including 200 nanoseconds for memory access, would be 3.16 $\mu$sec. . That is, the remote (through the network) memory access is more than an order of magnitude greater than local memory access. Note also that the delay expressions and associated tables do not indicate whether or not the frequencies suggested are achievable. This is dependent on the logic and path delays, and on clock distribution characteristics (e.g. clock skew). These design parameters are explored in the next section.

## 5. Clock Frequency and Data Rate Expressions

The clock frequency at which a given system can run is determined by a host of factors ranging from effectiveness of the logic design, to chip layout and signal delay consequences of using a particular packaging technology. The final result of these factors are a set of delay parameters which determine the rate at which data can pass from chip to chip. This maximum data rate corresponds to the maximum clock frequency which can be used without errors occurring.

| | CLOCK FREQUENCY (MHZ) | | | | |
|---|---|---|---|---|---|
| W | 10 | 20 | 30 | 40 | 80 |
| 1 | 14.8 | 7.4 | 4.9 | 3.7 | 1.9 |
| 2 | 9.8 | 4.9 | 3.3 | 2.5 | 1.2 |
| 4 | 7.3 | 3.7 | 2.4 | 1.8 | .91 |
| 8 | 6.1 | 3.1 | 2.0 | 1.5 | .76 |

MCC (Mesh Connected Crossbar) MODEL

| | CLOCK FREQUENCY (MHZ) | | | | |
|---|---|---|---|---|---|
| W | 10 | 20 | 30 | 40 | 80 |
| 1 | 11.5 | 5.75 | 3.8 | 2.88 | 1.44 |
| 2 | 5.9 | 2.95 | 1.9 | 1.48 | .74 |
| 4 | 3.1 | 1.55 | 1.03 | .78 | .39 |
| 8 | 1.9 | .95 | .63 | .48 | .24 |

DMC (Demux/Mux Crossbar) MODEL
Table 2: Time Through Network ($\mu$sec)
($P$=100, 512$\leq N' \leq$4096, $N$=16)

The data rate can be expressed in a general form as follows:

$$DR = \frac{1}{max[\textit{information signal delay, clock signal delay}]} \qquad (5.1)$$

Information signal delays consist of logic and memory delays ($D_L$), information signal path delays ($D_p$) and delays due to clock skew ($\delta$). The sum of these delays can be associated with the time for a signal to pass between communicating modules (chips or crosspoint switches). Since this must occur within a single clock cycle, a constraint is placed on minimum clock cycle duration (and thus maximum frequency). Taking the worst case (largest) sum of these delays overall communicating modules leads an overall constraint on data rate and clock frequency [25].

In a similar manner, delays associated with propagation of the clock signal form another basic limitation on clock frequency. Two types of clock distribution schemes are considered here. In the first (the STANDARD CLOCK SCHEME), the entire clock tree is viewed as an equipotential surface which must achieve a single final voltage in each half of the clock cycle. That is, the entire clock tree must be charged and discharged during the clock cycle. Given that such clock trees may present a large capacitive load, and that this load grows as systems become larger, this constraint can be a limiting factor in achieving high frequencies in physically large systems. If $\tau$ is the time required to charge or discharge the clock tree, then the data rate for this clocking scheme is constrained to be less than $1/2\tau$. Based on the above, the data rate for the case of a Standard Clocked Scheme is as follows:

$$DR_{sc} = \frac{1}{max[D_L + D_p + \delta,\ 2\tau]} \qquad (5.2)$$

Notice also that a relationship exists between $\tau$ and $\delta$. That is, as the clock line increases in length, both $\tau$ and $\delta$ increase with $\tau$ being an upper bound on $\delta$. One model that relates $\tau$ and $\delta$ was developed by Wann and Franklin [25]. The model assumes simple exponential rise times to and from the power supply voltage, $V_{dd}$. Variations in material properties can result in variations in rise time which can be expressed in terms of maximum and minimum $\tau$ values, $\tau_{\max}$ and $\tau_{\min}$. Variations in processing can result in variations in FET threshold voltages which can also be expressed in terms of maximum and minimum threshold voltage values, $V_{Tmax}$ and $V_{Tmin}$. The resultant expression is:

$$\delta = \tau_{\min}\, ln(1 - \frac{V_{Tmin}}{V_{dd}}) - \tau_{\max}\, ln(1 - \frac{V_{Tmax}}{V_{dd}}) \qquad (5.3)$$

When dealing with long clock lines, the charge/discharge time constraint and the clock skew can severely limit performance. The charge/discharge time constraint can be partially overcome by treating the clock lines as transmission lines and, using the memory properties of the line, placing multiple pulses on the line at the same time instant thus reducing the delay between clock pulses (MULTIPLE PULSE SCHEME). Clock skew can be reduced by using a global clock and employing phase locked loop techniques (called "dynamic delay adjustment" in reference 1) to perform phase synchronization. These techniques, however, add complexity to the design and are not pursued here.

## 6. An Example

### 6.1. Physical Design

Based on the design constraint and data rate expressions developed in prior sections, we now consider the design of an interconnection network with 2048 inputs and outputs. As stated in section 3, pin constraints on the chip limit the maximum size of the crossbar that can be implemented in a chip. A 16x16 network having path widths of 4 bits appears to be reasonable and is used in this example.

The implementation of a 256x256 network on a single board was shown to be possible, satisfying area and pin constraints on the board. Larger size networks can then be implemented from these boards by racking the boards in three dimensional space to reduce the distances over which wires must be routed. A 2048 input, 2048 output network implementation is shown in Figure 5. The first two stages of the network are implemented from eight 256x256 network boards; the last stage consists of eight boards with each board implementing one eighth of the last stage. If the boards are stacked as shown in Figure 5, the longest wire between any two chips is one that traverses the diagonal of a board. In section 3 we showed that this distance is 35 inches.

### 6.2. Clock Frequency

Expressions for the clock frequency (data rate) were derived in Section 5 based on the logic and memory delays and signal path delays and skews in clock distribution. In this section we estimate the information signal delays and clock delays from a knowledge of the physical layout of the network. A STANDARD CLOCK SCHEME is used.

Information signal delays consist of delay in the logic and in the memory of a finite state machine implementaion. Estimates given in [16] indicate that the logic delay would be about 12 nano seconds whereas the memory delay can be restricted to about 2 nano seconds. This results in $D_L$ = 14 nsec.

Consider the path delay. Since the network is pipelined, only the largest path delay is of significance. This corresponds to the path that has to go off-chip and traverse the longest distance (35 inches) on the board. This results in off-chip delays much larger than the signal rise times, thus off-chip paths must be viewed as transmission lines with on-chip drivers matched to their impedance (50 ohms). The off-chip path delay is determined largely by the speed of signals on the board and this is typically 0.15 nsec/inch. The delay in driving the driver is about 6 nsec. Hence $D_p = 6 + 0.15*35 = 11.3$ nsec.

The clock distribution essentially consists of two parts: one totally internal to a chip, consisting of a tree distributing the clock to the 16x16 network, and the second external to the chips consisting of the clock distribution on the board. Using a H-tree distribution to minimize clock skew for the clock distribution internal to a chip, the clock delay is given by [27]:

$$\tau_{\text{chip}} = (10N^3 - 3)(3 - 2/N)R_0C_0/7 \qquad (6.1)$$

where $R_0$ and $C_0$ represent the resistance and capacitance of the last branch of the tree distributing the clock to a switch. In this design (a 16x16 network occupying a 1cmx1cm chip) we have $R_0C_0 = 0.244$ pico sec. and we get $\tau_{\text{chip}} = 4.1$ nano sec. The delay in the clock distribution on the board is determined in a similar manner as the path delay. It consists of the delay in driving a driver with a 50 ohm output impedance and the delay in driving a line of maximum length of 35 inches. Thus $\tau_{\text{board}} = 11.3$ nanoseconds. Thus, the total clock delay is $\tau = 15.4$ nanoseconds. An expression for the clock skew is given in Section 5. Assuming a 20% variation over the nominal value in both the clock delay and transistor threshold voltages, the clock skew is obtained as:

$$\delta = .8\tau\log(1-2/5) - 1.2\tau\log(1-3/5) = .7\tau = 10.8 \text{ nano sec.} \qquad (6.2)$$

We can now estimate the clock frequencies. Note that due to the pipeline design of the network, only the largest delays are significant and both the MCC and DMC designs resulted in equal clock frequencies. Since, $D_L + D_p + \delta > 2\tau$ the clock frequency is:

$$F = \frac{1}{D_L + D_p + \delta} = 27.7 \ MHz \qquad (6.3)$$

### 7. Summary and Conclusions

This paper has presented design expressions for implementing an $N \log N$ packet passing interconnection network composed of circuit switched CB chip modules. The expressions considered are derived from various physical constraints and network models. Chip level and board level pin and area constraints are considered and expressions are derived which indicate the sort of delay which can be expected through a network at a given frequency, and the design constraints on achieving that frequency.

An example 2048x2048 network design is considered. This example indicates that using aggressive packaging and MOS technology, a rate of about 28 Mhz is achieveable. However, this frequency, with this network design, would result in a one way delay (ignoring blocking and hot spot delays) of about 1 $\mu$second. A read operation from memory requiring a round trip would thus require more then 2 $\mu$seconds. This represents more then an order of magnitude slowdown when compared with accessing strictly local memory and appears to be a major problem in the design of VLSI based, network centered multiprocessor architectures which utilize standard clock destribution designs.

ACKNOWLEDGMENT

We would like to thank Bill Thomas for his help in developing a number of the models presented in the paper.

### REFERENCES

[1] P.D. Bassett, L.A. Glasser and R.D. Rettberg, "Dynamic Delay Adjustment: A Technique for High-Speed Asynchronous Communications," Proc. Fourth MIT Conf. on Advanced Research in VLSI, pp.219-232, Apr. 1986

[2] V.E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, New York, Academic, 1965.

[3] Bolt Beranek and Newman, "Development of a Butterfly Multiprocessor Test Bed," Quarterly Tech. Rpt. No. 1, March 1985.

[4] S. Browning, "The Tree Machine," Ph.D. Thesis, California Inst. of Tech., Pasadena, 1980.

[5] D.M. Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Networks," IEEE Trans. on Comput., Vol. C-30, No. 4, pp. 331-340, April 1981.

[6] A.L. Fisher and H.T. Kung, "Synchronizing Large VLSI Processor Arrays," IEEE Trans. on Comput., Vol. C-34, No. 8, pp.734-740, Aug. 1985.

[7] M.A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communications Networks," IEEE Trans. on Comput., Vol. C-30, No. 4, pp. 283-291, April 1981.

[8] M.A. Franklin, D.F. Wann and W.J. Thomas, "Pin limitations and partitioning of VLSI interconnection networks," IEEE Trans. on Comput., Vol. C-31, No.11, pp. 1109-1116, Nov. 1982.

[9] L.R. Goke and G.L. Lipovski, "Banyan networks for partitioning multiprocessor systems," Proc. 1st Annu. Symp. Comput. Arch., pp. 21-28, 1973.

[10] A. Gottleib, et.al., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. on Comput., pp. 175-189, Feb. 1983

[11] E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI," IEEE Trans. on Comput., Vol. C-30, No. 4, pp. 247-253, Apr. 1981.

[12] P. Kermani and L. Kleinrock, "A Tradeoff Study of Switching Systems in Computer Communications Networks," IEEE Trans. on Comput., Vol. C-29, No. 12, pp.1052-1060, Dec. 1980.

[13] D.J. Kuck, "The Structure of Computers and Computations: Volume 1," Section 6.5 Typical Alignment Networks, John Wiley and Sons, N.Y., 1978

[14] D.H. Lawrie, "Access and Alignment of data in an array processor," IEEE Trans. on Comput., vol. C-24, pp.1145-1155, Dec. 1975.

[15] C.E. Molnar, T.P. Fang and F.U. Rosenberger, "Synthesis of Delay-Insensitive Modules," 1985 Chapel Hill Conf. on VLSI, Computer Science Press, pp. 67-86, March 1985.

[16] K. Padmanabhan, "Multiprocessor Interconnection Networks In A VLSI Environment," M.S. Thesis, Dept. of Elec. Engr., Washington Univ., St. Louis, Mo., Dec. 1981.

[17] J.H. Patel, "Performance of processor-memory interconnection networks for multiprocessors," IEEE Trans. on Comput., Vol. C-30, No. , pp. 771-780, Oct. 1981.

[18] M.C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., vol. C-26, pp.458-473, May 1977.

[19] G.F. Pfister, et.al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," Proc. 1985 Inter. Conf. on Parallel Processing, pp. 764-771, Aug. 1985.

[20] G.F. Pfister and V.A. Norton, "Hot spot contention and combining in multistage interconnection networks," IEEE Trans. on Comput., Vol. C-34, No. 10, pp.943-946, Oct. 1985.

[21] B. Quatember, "Modular crossbar switch for large-scale multiprocessor systems - Structure and implementation," Proc. AFIPS 1981 Nat. Comp. Conf., vol. 50, pp. 125-135, 1981.

[22] C.L. Seitz, "The Cosmic Cube," Comm. ACM, Vol. 28, No.1, pp. 22-33, Jan. 1985.

[23] H.J. Siegel, R.J. McMillen and P.T. Muller, "Computer interconnection structures: Taxonomy, charactertics and examples," Proc. Nat. Comput. Conf., June 1979.

[24] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," Computer, Vol. 15, No. 1, pp. 47-56, Jan. 1982.

[25] D.F. Wann and M.A. Franklin, "Asynchronous and Clocked Control Structures for VLSI Based Interconnection Networks," IEEE Trans. on Comput., Vol. C-32, No. 3, pp. 283-291, March 1983.

[26] D.S. Wise, "Compact layouts of banyan/FFT networks," VLSI Systems and Computations, H.T. Kung et. al. Eds. Rockville, MD: Comput. Sci. Press, 1981.

[27] S.Y. Kung and R.J. Gal-Ezer, "Synchronous vs. asynchronous computation in VLSI array processors," Proc. SPIE, vol. 341, May 1982.

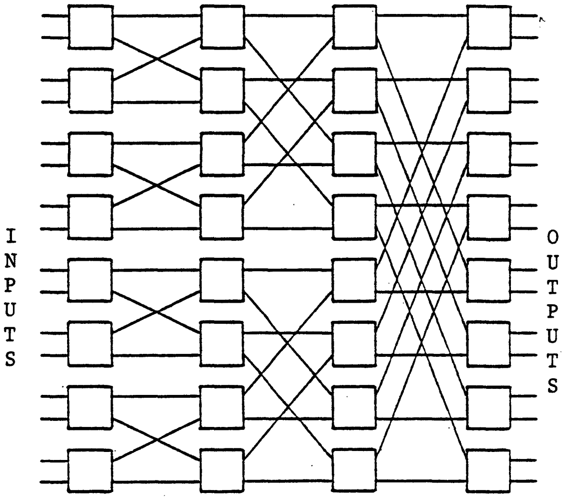


Figure 1: A 16x16 NlogN network built from 2x2 switch modules.

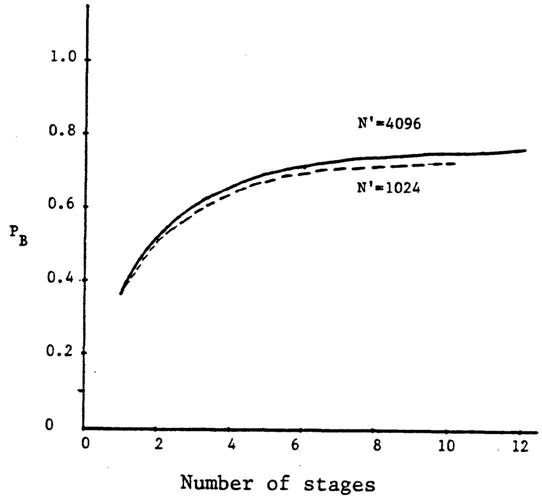


Figure 2: Plot of probability of blocking with the number of stagesin a NlogN network. Each stage consists of switches that are crossbar networks.

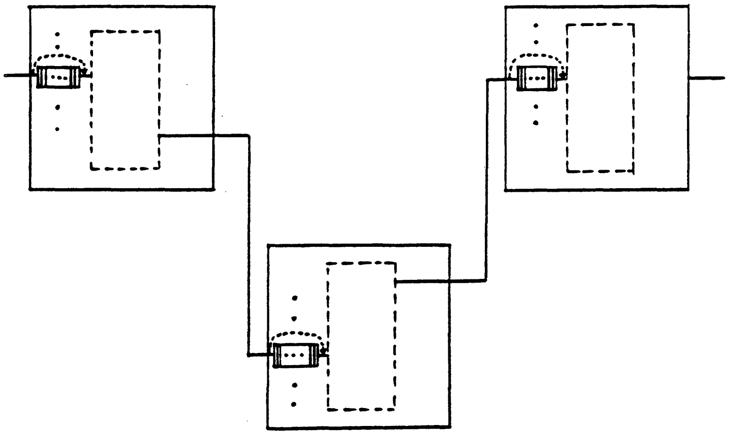
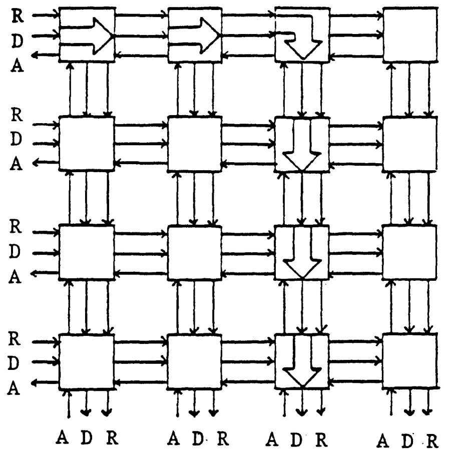


Figure 3: A path from an input to an output in a three stage network. Each input at each stage has a buffer that can be bypassed.



R - request
D - data
A - acknowledge

Figure 4a: The mesh connected crossbar network.

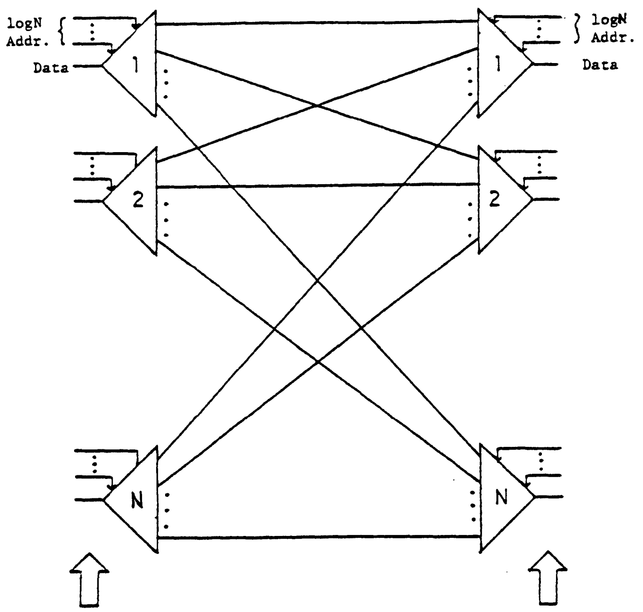


Figure 4b: The DMUX/MUX crossbar network implementation. Each input consists of a 1 to N demultiplexer and each output a N to 1 multiplexer.

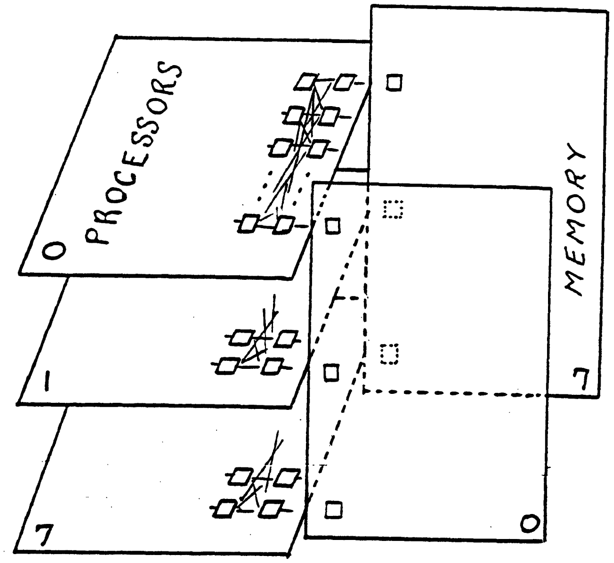


Figure 5: Board racking approach to packaging.

# A GROUP THEORETIC MODEL FOR SYMMETRIC INTERCONNECTION NETWORKS

*Sheldon B. Akers*
Dept. of Electrical & Computer Engg.
University of Massachusetts
Amherst, MA 01003

*Balakrishnan Krishnamurthy*
Computer Research Laboratory
Tektronix Laboratories
Beaverton, OR 97077

### *ABSTRACT*

Symmetric graphs, such as the ring, the $n$-dimensional Boolean hyper-cube and the cube-connected cycles, have been widely used as processor/communication interconnection networks. The performance of such networks is often measured through an analysis of their degree, diameter, connectivity, fault tolerance, routing algorithms, etc. In this paper we develop a formal group theoretic model, called the *Cayley graph* model, for designing, analyzing and improving such networks. We show that this model is universal and demonstrate how the networks mentioned above can be concisely represented in this model.

More importantly, we show that this model enables us to design new networks based on representations of finite groups. We can then analyze these networks by interpreting the group theoretic structure graph theoretically. Using these ideas, and motivated by certain well known combinatorial problems, we develop two new classes of networks, called the *star graphs* and the *pancake graphs*. These networks are shown to have better performance, as measured by the parameters mentioned above, than the popular n-cubes.

## 1. INTRODUCTION

A processor/communication interconnection network is often modeled as an undirected graph, in which the nodes correspond to processor/communication ports, and the edges correspond to communication channels. Communication over such a network is achieved by a message passing protocol, and the delay in communication is measured by the number of edges traversed. Some of the key features of interest in such an interconnection network are its degree, diameter, congestion, symmetry, connectivity, routing algorithms, structure, etc.

A number of interconnection network topologies have been suggested in the literature which address one or more of the above features ([1-7]). These range from simple graphs, such as cycles, complete graphs and stars to more sophisticated graphs such as shuffle-exchange graphs, n-dimensional Boolean cubes and cube-connected cycles. Since there is no single measure to compare these networks, each of the above examples have been justified for one application or another.

A special class of networks, called *symmetric interconnection networks*, have the property that the network viewed from any vertex of the network looks the same. In such a network congestion problems are minimized since the load will be distributed uniformly through all the vertices. Moreover, this symmetry allows for identical processors at every node with identical routing algorithms. It is also very useful in designing algorithms that exploit the structure of the network. it is this class of networks that we address in this paper. With the exception of the star network and the shuffle exchange graphs, all of the networks mentioned above are symmetric networks.

In designing symmetric interconnection networks, the overall objective has been to construct large vertex symmetric graphs with small degree and diameter, high connectivity and offering simple routing algorithms.[1] One attractive network that offers all these properties together with a good degree and diameter is the n-cube, which is a network of $2^n$ vertices, with degree $n$ and diameter $n$. Thus, the n-cube will be used as a standard against which to compare many of the networks constructed in this paper.

Specifically, we shall present a group theoretic model for designing, analyzing and improving symmetric interconnection networks. We show that most symmetric interconnection networks can be represented using this model, and that *every* symmetric interconnection network can be represented by a simple extension of this model. This allows us to provide an algebraic representation of each of the symmetric networks mentioned above. More importantly, this group theoretic model enables us to start with an arbitrary finite group and construct a symmetric network using that group as the algebraic model. This, in conjunction with the vast literature, on finite groups allows us to construct a variety of new interconnection networks.

Another advantage of analyzing such networks in this algebraic setting is that many properties of these networks can be proved for the class as a whole, instead of proving that property for each network independently. For example, all networks derived from a finite group are necessarily vertex symmetric, i.e., the network viewed from any vertex in the network looks the same. We prove a number of other such properties of these networks. Further, even for specific networks constructed using this algebraic model we can often derive properties algebraically and interpret the properties graph theoretically. We will repeatedly use this technique in this paper.

Apart from the algebraic model suggested in this paper we also offer two specific classes of networks that are especially attractive for distributed processing. We call these the *star graphs* and the *pancake graphs*. We describe these networks in some detail and compare them to the $n$-cubes.

The paper is organized into 8 sections. In the second section we define the algebraic model, called the *Cayley graph* model and provide a number of examples as well as some new networks. In Section 3 we prove certain general properties of Cayley graphs. In Section 4 we define *transposition trees* which is a level of abstraction beyond Cayley graphs. A specific Cayley graph resulting from a specific transposition tree, called the *star graph*, is investigated in Section 5. In Section 6 we present some preliminary ideas on designing algorithms on another class of such graphs, called *pancake graphs*. In Section 7 we compose Cayley graphs using a variety of group theoretic operations and study the resulting graphs. The concluding Section 8 reiterates the fertile source of new networks offered by this group theoretic model.

[1] We point out that the problem of constructing large graphs of a given degree and diameter - known as the (d,k) graph problem - is a well studied extremal graph theory problem. However, solutions to the (d,k) graph problem often ignore the more subjective parameters of this problem, such as symmetry, ease of routing and the structure of the graph, which are essential for the design of efficient algorithms on these networks.

0190-3918/86/0000/0216 $01.00 © 1986 IEEE

In the remainder of this section we will state the group theoretic and graph theoretic terminology that we will use in this paper. We will assume basic knowledge of elementary group theory and graph theory. The reader is referred to [14 and 15] for an elementary exposition of the relevant terminology in group theory and graph theory, respectively. Since we will only be considering finite groups we will represent our groups as permutation groups. We will be using a one-row representation for permutations. Thus, the permutation whose cycle representation is (12)(345) will be represented by us as 21453. Recall that transpositions are permutations with exactly one cycle of length 2 and all other cycles of length 1. Our descriptions of permutations will be quite informal. Thus, we will view a transposition as *swapping* a pair of symbols.

As we mentioned in the beginning of the section, we will view interconnection networks as an undirected[2] graph. Thus in the remainder of the paper we will use the term *graph* in place of *interconnection network.* Our graphs will be finite, undirected, loop-free and devoid of multiple edges. We will make special use of cycles, complete graphs, trees, and stars.

## 2. THE CAYLEY GRAPH MODEL

Given a set of generators for a finite group G, we can draw a graph, called the *Cayley graph,* in which the nodes correspond to the elements of the group G and the edges correspond to the action of the generators. That is, there is an edge from an element $a$ to an element $b$ iff there is a generator $g$ such that $ag=b$ in the group G. We require that the set of generators be closed under inverses so that the resulting graph can be viewed as being undirected.

Let us illustrate Cayley graphs with a few examples which will make these ideas clearer. First, we point out that since we will be considering permutation groups the generators are themselves permutations. We will be representing permutations using the symbols 1, 2, 3,..., $n$. For example, consider the generators 1324, 2143 and 4321. Since these generators are permutations of four symbols they must generate either $S_4$ (which itself consists of 24 permutations) or a subgroup[3] of $S_4$. In this case the subgroup generated by these three generators consists of the 8 permutations: 1234, 2143, 2413, 4231, 4321, 3412, 3142 and 1324. The corresponding Cayley graph arising from the above generators is shown in Figure 1. The reader can easily convince himself that the edges of this graph correspond to the action of the generators.

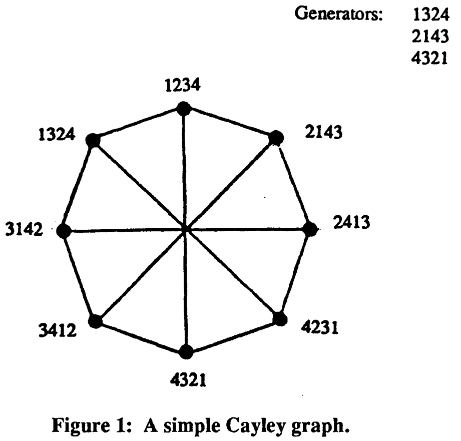


**Figure 1: A simple Cayley graph.**

As another example, consider the generators 213456, 124356 and 123465. The group generated by these generators again contains 8 permutations. These permutations and the corresponding Cayley graph are shown in Figure 2. We might point out that this is, of course, the familiar 3-dimensional cube. Figures 3 and 4 show two additional examples of Cayley graphs, where we have only shown the permutations at a few selected nodes. In Figure 4 we have also shown the generators as labels on the edges. Again we might simply mention that Figure 3 is a 3-dimensional cube-connected cycle, i.e., a 3-dimensional cube with its corners chopped. Figure 4 is an example of a star graph that we will discuss later in this paper.

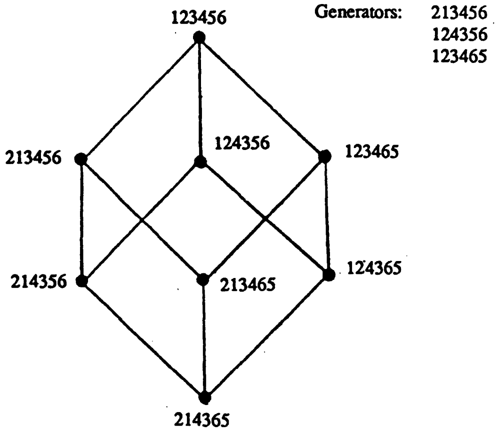


**Figure 2: 3-Cube as a Cayley graph.**

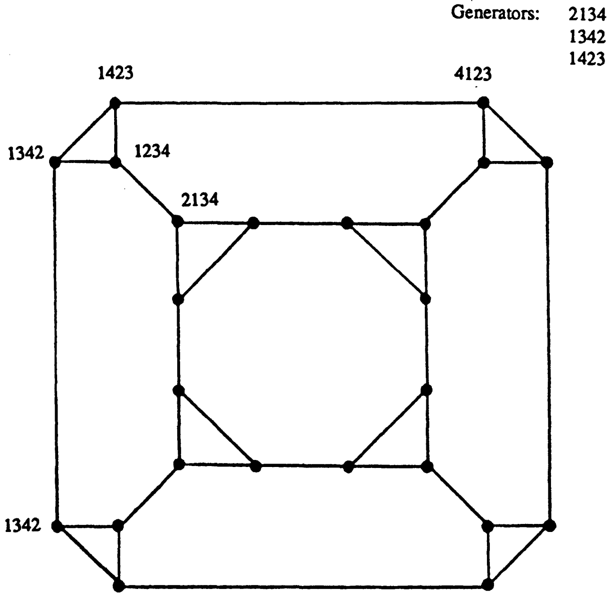


**Figure 3: 3-Dimensional cube-connected cycles as a Cayley graph.**

[2] The techniques of this paper can also be used for directed graphs. However, we will limit our attention to undirected graphs.

[3] The reader unfamiliar with LeGrange's theorem would find it useful to know that in a finite group the order of any subgroup (the number of elements in it) always divides the order of the group.

The reader might have observed that all the Cayley graphs shown so far are vertex symmetric. In fact,

*Theorem 1*: Every Cayley graph is vertex symmetric.

*Proof*: We need to show that given any two vertices in the Cayley graph there exists an automorphism of the graph that maps one vertex into the other. Let $a$ and $b$ be the permutations corresponding to the two vertices. Consider the transformation on the group that maps an arbitrary permutation $x$ into $ba^{-1}x$. Clearly, this maps $a$ into $b$. Further, this transformation is an automorphism of the graph. For, if two permutations $x$ and $y$ were connected by an edge in the graph then there is a generator $g$ such that $xg=y$. Then the images of the two permutations, $ba^{-1}x$ and $ba^{-1}y$, are connected by an edge since $ba^{-1}xg=ba^{-1}y$. Hence the proof. □

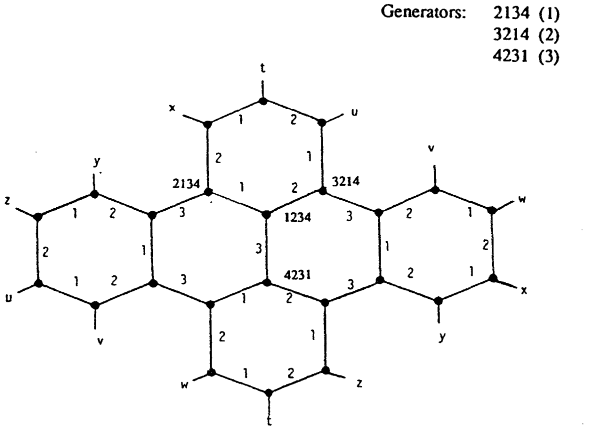


**Figure 4: An example of a star graph.**

As mentioned in Section 1, an attractive feature of vertex symmetric graphs is that routing between two arbitrary nodes reduces to routing from an arbitrary node to a special node. We can now state this more formally. First, observe that in a Cayley graph a path from one vertex to another can be represented by a sequence of generators $g_1,g_2,\ldots,g_p$, where each $g_i$ is a generator of the Cayley graph. Now if $g_1,g_2,\ldots,g_p$ is a path from $x$ to $y$ then it is also a path from $y^{-1}x$ to $I$, the identity permutation. Thus if we want to find a route from $x$ to $y$ we can instead find a route from $y^{-1}x$ to $I$. Consequently the problem of routing reduces to the problem of sorting!

We can now describe a class of Cayley graphs, which we will call the *bubble sort graphs*. Recall that a Cayley graph is completely specified by providing the set of generators. The generators for the $n^{th}$ bubble sort graph is the set of $n-1$ transpositions of the $n$ symbols $1,2,\ldots,n$ that transpose adjacent symbols. Thus, for $n=4$ the generators are: 2134, 1324 and 1243. Notice that a path in this graph is a sequence of adjacent transpositions. Thus, finding a route from a given permutation to the identity permutation is equivalent to sorting the given permutation using the familiar bubble sort algorithm. The reader can verify that the group generated by this set of generators is the symmetric group $S_n$. Consequently, the corresponding Cayley graph has $n!$ vertices. Further, it is easily shown that the corresponding Cayley graph has degree $n-1$ and diameter $\binom{n}{2}$.

As a second example of a family of Cayley graphs we consider an old combinatorial problem, called the *pancake flipping problem* [8]. The problem is to sort a stack of $n$ pancakes of different sizes by repeatedly flipping top sub-stacks with a spatula. For example, consider an arrangement of 5 pancakes represented by the permutation 23514. Let us adopt the convention that the left end of the permutation is the top of the stack. Thus the sorted stack would look like the identity permutation 12345. If we apply a 3-flip, i.e., flip the top (left) three pancakes with the spatula, the 23514 permutation will be transformed into 53214. Notice that such a 3-flip is equivalent to multiplying on the right by 32145. We give below a sequence of flips to sort the given permutation:

$$23514 \rightarrow 53214 \rightarrow 12354 \rightarrow 45321 \rightarrow 54321 \rightarrow 12345$$

We model the pancake flipping problem as a class of Cayley graphs using generators representing the pancake flips. Thus, flipping the top $i$ of $n$ pancakes with a spatula gives rise to the generator $i(i-1)...321(i+1)(i+2)...n$. Clearly, there are (n-1) generators, one for each value of $i$, $1<i\leq n$. It is easy to show that the corresponding Cayley graph has $n!$ vertices with degree $(n-1)$. Finding the diameter of the *pancake graph* is equivalent to finding the maximum number of pancake flips one would need to sort an arbitrary stack of pancakes. This problem is still open, and the best know results can be found in [13].

For this paper we will give a simple routing algorithm that routes in at most (2n-3) steps. Recall that instead of finding a path between two arbitrary permutations, it suffices to find a path from one permutation to the identity, i.e., sort a given permutation. In one step we can bring the symbol $n$ to the left most position using an appropriate flip (generator). In one more step, we can bring $n$ to the $n^{th}$ position using the $n$-flip. Thereafter, we can ignore the symbol $n$ and sort the remainder of the permutation, recursively. This yields a route of $2n$ steps. We can slightly improve this by observing that the last two symbols, i.e., the symbols 1 and 2, do not require two steps each. In fact, when we have moved all the other symbols to their place, 1 and 2 would require at most one more flip. Hence the (2n-3) result. Note that we have in effect proved a (2n-3) upper bound on the diameter of this Cayley graph. We will point out that in contrast to the $n$-cubes, whose diameter and degree grow logarithmically as a function of the number of vertices, the $n$-pancake graphs have degree and diameter that grow slower than logarithmically as a function of its size.

Before we offer a picture of the pancake graph for a small value of $n$, we will first mention some elementary properties of Cayley graphs. This will allow us to use the underlying group theoretic structure of these graphs in analyzing them.

## 3. PROPERTIES OF CAYLEY GRAPHS

Since a Cayley graph is completely specified by a set of $d$ permutations as generators. The degree of the graph will, of course, be $d$. We have already shown in Theorem 1 that every Cayley graph is vertex symmetric. As we had mentioned in Section 1, a vertex symmetric graph has the desirable property that the communication load is uniformly distributed on all the vertices so that there is no point of congestion. A stronger notion of symmetry, called *edge symmetry* requires that every edge in the graph look the same. That is, given two edges of the graph there is an automorphism of the graph that maps one edge into the other. Such a symmetry would ensure that the communication load is uniformly distributed over all the communication links, so that there is no congestion at any one link. It is easy to prove the following necessary condition for a Cayley graph to be edge symmetric. We state it without proof:

*Theorem 2*: Consider a Cayley graph defined by a set of $d$ generators on $n$ symbols $\{1,2,3,...,n\}$. If for every pair of generators there exists a permutation of the symbols that maps one generator into the other then the Cayley graph is edge symmetric.

As a consequence of the above theorem we can conclude that the bubble sort graphs are edge symmetric.

Apart from the symmetry properties of these Cayley graphs there is a very useful decomposition of these graphs that can be seen using elementary group theory. First, we observe that one of the attractive features of the $n$-cube is its recursive decomposition into smaller cubes. Thus, an $n$-cube can be viewed as consisting of two $(n-1)$-cubes, interconnected by edges that are said to lie in the $n^{th}$ dimension. We will now show that this property can be abstracted as a group theoretic property and is possessed by a number of different Cayley graphs.

Consider the 4-pancake graph defined by the 3 generators 2134, 3214 and 4321 on 4 symbols. Let us examine the subgraph of the 4-pancake graph consisting of those vertices (i.e., permutations) that fix the symbol 4 in the $4^{th}$ position. Clearly, there are 3! such permutations and thus six vertices in this subgraph. Further the only edges that connect these vertices in this subgraph are those that correspond to the first two generators, since the last generator will move the symbol 4 from the $4^{th}$ position. Consequently, this subgraph is identical to a 3-pancake graph (on the symbols $\{1,2,3\}$), with the symbol 4 being affixed at the end of each permutation.

More interestingly, let us now examine the subgraph of the 4-pancake graph consisting of the 6 permutations that fix the symbol 3 in the last position. Once again we will find that the subgraph is identical to the 3-pancake graph, but this time on the symbols $\{1,2,4\}$, with the symbol 3 being affixed at the end of each permutation. In this manner we can identify 4 mutually disjoint subgraphs of the 4-pancake graph, each of size $3!=6$, with each subgraph being a copy of the 3-pancake graph. These 4 copies of the 3-pancake graph are then interconnected by edges that correspond to the $4^{th}$ generator, i.e., the generator corresponding to a flip of all four pancakes. We can now offer the informative picture of Figure 5 illustrating the 4-pancake graph.

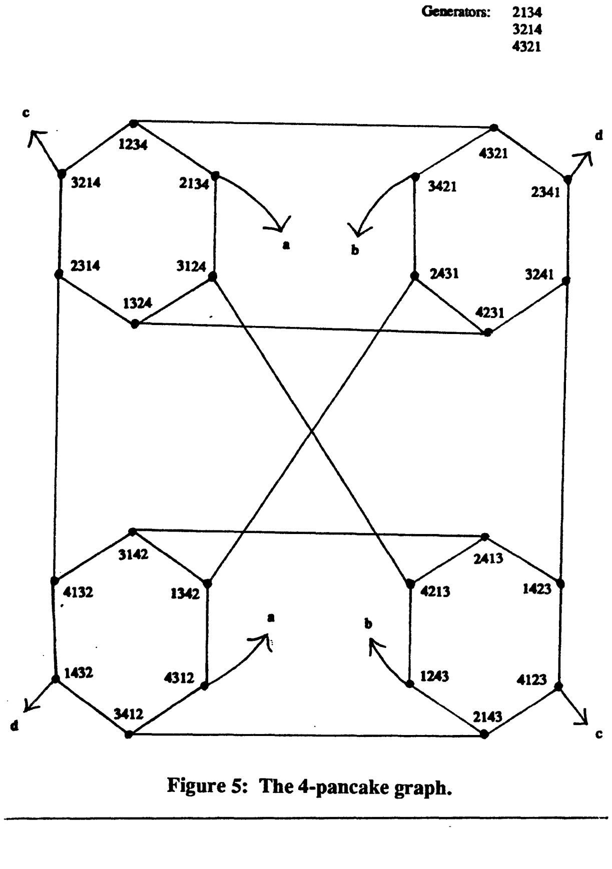


**Figure 5: The 4-pancake graph.**

Finally we make the trivial observation that the 3-pancake graph itself (i.e., the hexagon) is made up of 3 copies of the 2-pancake graph which is itself a line. More generally, an $n$-pancake graph can be viewed as $n$ copies of $(n-1)$-pancake graphs that are interconnected by edges corresponding to the $n$-flip. Further, this decomposition can be carried out recursively, so that each of the $(n-1)$-pancake graphs in turn are made up of $n-1$ copies of $(n-2)$-pancake graphs, and so on.

We can now ask which Cayley graphs have this recursive decomposition property. Notice that in the 4-pancake graph we used the property that the 4-flip cannot be obtained through any combination of 2- and 3-flips. That is, the permutation corresponding to the 4-flip, i.e, 4321, is outside the subgroup generated by the 2- and 3-flips. In fact, that is the only requirement that is needed for such a decomposition. Thus, for convenience we will define a Cayley graph to be *hierarchical*, if its generators can be ordered as $g_1,g_2,\ldots,g_d$, such that for each $i$, $1<i\leq d$, $g_i$ is outside the subgroup generated by the first $i-1$ generators. Under this definition, every hierarchical Cayley graph has such a recursive decomposition structure.

Another important property of interconnection networks is their fault tolerance. The *fault tolerance* of a graph is better defined through the graph theoretic property, called *connectivity*. The *connectivity* of a graph is the minimum number of vertices that need to be removed to disconnect the graph. The fault tolerance is then one less than the connectivity and indicates the maximum number of vertices that can be removed and still have the graph remain connected. Clearly, any graph can be disconnected by removing all the vertices adjacent to a given vertex. Thus its connectivity can be at most its degree.. It has been shown that hierarchical Cayley graphs (with an additional size requirement) are *maximally fault tolerant*. That is, their fault tolerance is exactly one less than their degree.

The last property we will mention in this section is the universality of this model for symmetric graphs. Can every vertex symmetric graph be represented as a Cayley graph? For example, the $n$-cube can be represented as a Cayley graph by generalizing the example shown in Figure 2. We will offer a more formal representation of the $n$-cube in Section 7. We will also provide in that section a representation of the cube-connected cycles as Cayley graphs. While, most symmetric networks considered in the literature can be viewed as Cayley graphs, it remains that there are certain vertex symmetric graphs that cannot be represented as Cayley graphs. A prime example is the Petersen graph, shown in Figure 6.

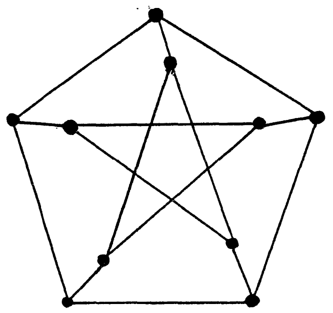


**Figure 6: Petersen graph — An example of a vertex symmetric graph that is not a Cayley graph.**

However, one can extend the Cayley graph model to the *quotient graph* of two Cayley graphs. In this paper we will merely give a rough definition of the quotient and state (without proof) a theorem that establishes the universality of this model. To define the quotient graph of two Cayley graphs we select a subgroup of the group generated by the given set of generators. We then identify the subgroup and all its cosets as subgraphs of the Cayley graph. The quotient graph is obtained by reducing these subgraphs to vertices and connecting two such vertices iff there existed an edge between elements of the corresponding subgraphs.

*Theorem 3*: Every vertex symmetric graph can be represented as the quotient of two Cayley graphs.

Finally, we mention an interesting open conjecture. Recall that a *Hamiltonian path* is a path that visits every vertex exactly once. A *Hamiltonian cycle* is a cycle that forms a Hamiltonian path.

*Conjecture*: Every Cayley graph is Hamiltonian, i.e., has a Hamiltonian cycle. Further, every vertex symmetric graph has a Hamiltonian path.

This conjecture has remained open in the sense that neither has it been proven nor has a Cayley graph been shown to violate it. For specific Cayley graphs it is often easy to establish that it is Hamiltonian. For example, the Hamiltonian property of the $n$-cubes is demonstrated by Gray codes. The Hamiltonian property of the $n$-pancake graphs have been shown in [16].

As we have mentioned, most symmetric interconnection networks that have been suggested in the literature can be represented as Cayley graphs. However, we offer this group theoretic model not only to capture existing networks, but also to design new networks. It is in this vein that we suggest the pancake graphs. We have already pointed out how their degree and diameter (as a function of its size) are more attractive than the $n$-cubes. Later, in Section 6 we will show how we can design some fundamental computational algorithms on the pancake graphs. But first, we will show how even better graphs can be designed using the Cayley graph model.

## 4. TRANSPOSITION TREES

Again we recall that a Cayley graph is completely specified by providing a set of $d$ permutations as generators. We do point out a requirement that this set of permutations be closed under inverses. In the examples of Cayley graphs given above we have not explicitly shown to have met this condition since the generators that we have used have often been involutions, i.e., self-inverses. A special class of involutions are the transpositions — permutations that swap two symbols. For example, 12435 is a transposition that swaps the symbols 3 and 4. All the generators of the bubble sort graphs are transpositions. In this section we provide a model for representing a set of (n-1) transpositions as generators.

Consider a tree on $n$ vertices. We can label the vertices of this tree with the symbols $\{1,2,3,...,n\}$ and interpret the edges as transpositions. For example, the tree on 6 vertices shown in Figure 7, gives rise to 5 transpositions: 321456, 132456, 124356, 123546 and 123654. Thus, we can interpret a tree as a set of transpositions, which in turn give rise to a Cayley graph. We call such a tree, a *transposition tree*. As another example, the path on $n$ vertices gives rise to the bubble sort graph. The following general theorem about Cayley graphs of transposition trees is an indication of the symmetry and structure underlying these graphs that can be readily uncovered by a simple group theoretic analysis:

*Theorem 4*: Let $\Gamma$ be a Cayley graph of a transposition tree of order n. Then,

1. $\Gamma$ has $n!$ vertices. (A well known result attributed to Polya)
2. Both the edge and vertex connectivity of $\Gamma$ are maximal, i.e., equal to its degree.
3. The chromatic index of $\Gamma$ is equal to its degree.
4. $\Gamma$ can be represented as an interconnection of n identical copies of a Cayley graph of a transposition tree of order $n-1$, and hence is hierarchical.

We can view the Cayley graph of a transposition tree as the state diagram of a puzzle. Consider the vertices of the transposition tree to be labeled as suggested above. Now place $n$ markers, each labeled with a symbol from $\{1,2,3,...,n\}$, at the vertices of the tree in any arbitrary way. The puzzle is to move the markers to their appropriate positions by moves consisting of interchanging the markers at the ends of an edge in the transposition tree. The Cayley graph is then the state diagram of such a puzzle. That is, the vertices of the Cayley graph are the possible arrangement of markers at the vertices of the tree. The edges of the Cayley graph correspond to the permissible moves in this puzzle. Finding a path in the Cayley graph corresponds to sorting a permutation, which in turn corresponds to solving this puzzle. It can be shown:

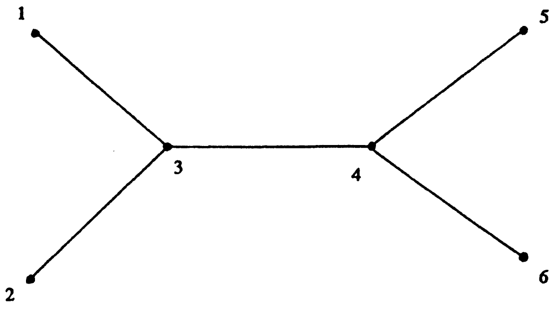


**Figure 7: An example of a transposition tree.**

*Theorem 5*: Let $T$ be a transposition tree on $n$ nodes. Given an assignment of markers as a permutation, $\pi$, of the nodes of $T$, $\pi$ can be sorted in

$$\textit{\# of cycles in } \pi + n + \sum_{i=1}^{n} \delta(i,\pi(i))$$

where, $\delta(i,j)$ is the distance between nodes $i$ and $j$ in $T$.

We remark that the above bound on the diameter of the Cayley graph conforms to the diameter of the bubble sort graph. In fact,

*Theorem 6*: Of all trees on n nodes, the path yields a Cayley graph (i.e., the bubble sort graph) of maximum diameter.

We omit proofs of the above theorems for brevity. Instead we now focus our attention on a specific transposition tree.

## 5. THE STAR GRAPH.

Motivated by Theorem 6, we consider the other extreme tree, namely the star, as a transposition tree. The resulting Cayley graph is called the the *star graph*. Since this is an especially attractive alternative to the n-cube, we provide an informal description of this graph. The nodes of the graph are labeled by permutations of 1 through $n$. A permutation is connected to every other permutation that can be obtained from it by interchanging the first symbol with any of the other symbols. Thus, clearly the degree of the graph is $n-1$. An illustration of the 4-star graph was given in Figure 4.

Let us examine how we might route within this graph. Recall that routing between two vertices in a Cayley graph is equivalent to sorting a permutation. So we need to ask how we might sort a given permutation by exchanging the first symbol with any of the other symbols. For example, consider the permutation 64725831. Let us employ a greedy algorithm where we observe that the symbol in the first position, namely 6, can be moved to its correct position by exchanging 6 and 8. This gives us the permutation 84725631. Again, a greedy move gives us 14725638.

Now we are stuck, since 1 is already at its own position. We now waste one step and move 1 into any position not occupied by the correct symbol. In this case we could interchange 1 and 4 yielding the permutation 41725638. Now following the greedy approach gives us 21745638 and subsequently 12745638. Again we need to waste a step and insert 1 into a position not occupied by the correct symbol. Thus, interchanging 1 with 7 gives 72145638. Resorting back to the greedy step gives 32145678 and the final move to 12345678. Notice that we took 8 moves to sort this permutation. That means that in the Cayley graph we have shown a path of length 8 between 64725831 and the identity.

Of course, this does not establish the diameter of the Cayley graph. For that we must find the permutation that requires the maximum number of steps to sort. Instead, we will derive the exact number of steps required to sort an arbitrary permutation in the following Lemma:

*Lemma 1*: The number of steps required to sort a permutation $\pi$ using the generators of the star graph is given by [4]:

$$n + \#\ \mathit{of\ cycles\ in}\ \pi - 2(\#\ \mathit{of\ invariances\ in}\ \pi) - \begin{cases} 2 & \text{if } \pi(1) \neq 1 \\ 0 & \mathit{otherwise} \end{cases}$$

*Proof*: Follows from the routing algorithm described above. □

*Theorem 7*: The diameter of the $n$-star graph is $\left\lfloor \frac{3}{2}(n-1) \right\rfloor$, and its average diameter is $n+\frac{2}{n}+H_n-4$, where $H_n$ is the $n$th Harmonic number.

*Proof*: The diameter follows by maximizing the formula given in Lemma 1. To establish the average diameter we total the quantity given in Lemma 1 for each $\pi$ and divide by $n!$. It is well known (see [17, p.176]) that the average number of cycles in a permutation of $n$ symbols is $H_n$, the $n^{th}$ harmonic number. Thus, the total number of cycles, over all $n!$ permutations is $n!H_n$. It is also easy to establish that the total number of invariances over all permutations of $n$ symbols is $n!$. Finally, the total number of permutations $\pi$, such that $\pi(1)=1$ is $(n-1)!$. Thus, the average diameter is:

$$\frac{1}{n!} \left[ n(n!) + n!H_n - 2 \cdot n! - 2(n! - (n-1)!) \right]$$

$$= n + H_n - 4 + \frac{2}{n}. \ \square$$

Let us compare this network against the $n$-cube. Recall that the $n$-cube interconnects $2^n$ vertices with degree $n$ and diameter $n$. In contrast, the $n$-star graph interconnects $n!$ vertices with degree $n-1$ and diameter $\left\lfloor \frac{3}{2}(n-1) \right\rfloor$. Notice that both the degree and diameter of the star graph grows slower than a logarithmic function of its size. Thus, asymptotically, the star graphs offer a network with less interconnecting edges and smaller communication delays than the $n$-cubes. Even from a practical point of view, it is evident from Table 1 that purely based on the degree and diameter requirements the star graph is superior. Table 1 provides a comparison of various $n$-cubes against comparable $n$-star graphs.

| A Comparison | | | | | | | |
|---|---|---|---|---|---|---|---|
| *n-cube* | | | | *n-star graph* | | | |
| *n* | *Size*<br>$2^n$ | *Degree*<br>*n* | *Diameter*<br>*n* | *n* | *Size*<br>*n!* | *Degree*<br>*n-1* | *Diameter*<br>$\left\lfloor \frac{3}{2}(n-1) \right\rfloor$ |
| 7 | 128 | 7 | 7 | 5 | 120 | 4 | 6 |
| 8 | 256 | 8 | 8 | 6 | 720 | 5 | 8 |
| 9 | 512 | 9 | 9 | 6 | 720 | 5 | 8 |
| 10 | 1024 | 10 | 10 | 7 | 5040 | 6 | 9 |
| 11 | 2048 | 11 | 11 | 7 | 5040 | 6 | 9 |
| 12 | 4096 | 12 | 12 | 7 | 5040 | 6 | 9 |

Table 1.

Of course, the degree and the diameter are not the only consideration for choosing a specific network. Let us examine some of the other relevant properties. The connectivity of the $n$-cube is $n$, indicating that up to $n-1$ vertices can fail without disrupting the network. Recall that the connectivity is at most equal to the degree of the graph. In the case of the $n$-star graph the degree is $n-1$, And, indeed, its connectivity is also $n-1$. So it is maximally fault tolerant. This is a result from [10]. Actually, such a worst-case fault tolerance measure does not really reflect the practical fault tolerance of the network. Even though the $n$-cube has a connectivity of $n$, the only way to remove $n$ vertices and disconnect the graph is by removing all the $n$ neighbors of any one vertex — a very unlikely event. Likewise, even though the connectivity of the $n$-star graph is $n-1$ the only way to remove $n-1$ vertices and disconnect the graph is to remove all the $n-1$ neighbors of any one vertex. In general both these networks can tolerate a much higher failure.

With regard to symmetry considerations, we recall that the $n$-cube is both vertex and edge symmetric. This alleviates any congestion problems. We know that the star graphs, being Cayley graphs, are vertex symmetric. Further, it is easy to show that the Cayley graph of an edge symmetric transposition tree is itself edge symmetric. Thus, the star graphs, arising from the star as a transposition tree, are also edge symmetric.

We have already noted the recursive decomposition structure of the $n$-cube? Does the star graph possess that property? It is easy to see from the generators that the star graphs are hierarchical. We can infer from that that they can be recursively decomposed. But, better yet, they can be recursively decomposed into $n$ copies of $(n-1)$-star graphs, each of which is in turn further decomposable into smaller star graphs. Such a recursive decomposition can be identified in the diagram of the 4-star graph given in Figure 4. Since the 3-star graph is a hexagon, the 4-star graph consists of 4 copies of the hexagon interconnected by edges corresponding to the $3^{rd}$ generator. Even though a preliminary inspection of Figure 4 suggests 6 hexagons, we should isolate those hexagons whose edges are labeled with 1 and 2 only. The reader will notice 4 such hexagons. These are then interconnected by the edges labeled 3.

Additionally, like the $n$-cube which can be decomposed along any one of its $n$ dimensions, the star graphs can be decomposed along any one of its "$n$ dimensions." Let us explain this further. Observe that an $n$-cube is made up of two copies of an $(n-1)$-cube connected by edges in the $n^{th}$ dimension. However, any one of the edges of the $n$-cube can be viewed as the $n^{th}$ dimension. Similarly, any one of the edges in the star graph can be viewed as the last dimension. So we can break it up into $n$ copies of $(n-1)$-star graphs along that dimension. We do this by observing that if we consider the subgraph consisting of all the permutations that fix the symbol $i$ $(1<i\leq n)$ in the $i^{th}$ position then we are left with $(n-1)!$ vertices that are interconnected by edges that correspond to swapping the first symbol with any one of the other symbols — except the symbol in the $i^{th}$ position.

[4] The number of cycles in a permutation includes the number of invariances.

We have tried to make a case for the star graphs by comparing them to the $n$-cubes. We believe that the Cayley model extracts the attractive properties of the $n$-cubes and formulates it in an abstract setting. This allows us to design other networks that possess similar properties, the star graphs being a prime example. An issue that we have not addressed in this section is how one uses the interconnection structure of these Cayley graphs to develop specific computational algorithms that can be executed on these networks. We address that in the next section using the pancake graphs as our example.

## 6. ALGORITHMS ON THE PANCAKE GRAPHS

Recall that the $n$-pancake graphs are obtained using the $n-1$ possible pancake flips, which we will denote by $f_2, f_3, ..., f_n$, as generators. The $i^{th}$ pancake flip, $f_i$, is a permutation that reverses the prefix of $i$ symbols in the identity permutation. In Section 2 we analyzed the degree and diameter of these graphs and also its recursive decomposition structure.

In this section we show how we might implement certain fundamental algorithms on such a network. First, we should recount similar algorithms on the $n$-cube. Suppose we wish to find the maximal element amongst $2^n$ elements distributed among $2^n$ processors located at the vertices of an $n$-cube. A straightforward algorithm involves $n$ time steps. At the $i^{th}$ step every processor communicates with the processor connected to it along the $i^{th}$ dimension and compares notes on the maximal element that it has encountered so far. We then claim that at the end of the $n$ steps every processor knows the value of the maximal element. The proof of the claim goes as follows: Let $d_i$ represent an edge along the $i^{th}$ dimension and let a word of the form $d_2 d_4 d_3$ represent a path in the $n$-cube starting from a specified vertex. Using these conventions, consider the word $d_1, d_2, \ldots, d_n$. It is easy to see that given any two vertices on the cube, there is a subsequence [5] of the above word that forms a path from the first to the second. Consequently, for every vertex $a$ of the $n$-cube there is a subsequence of the above word that forms a path from the vertex containing the maximal element to $a$. This establishes the claim that the processor at every vertex knows the maximal element.

The reason for detailing the above algorithm is to establish the background necessary to implement a similar algorithm on the pancake graphs. What we need is a word on the alphabet of the generators $\{f_i\}$, such that given any two vertices in the pancake graph there exists a subsequence of the suggested word that forms a path between the two vertices. Actually, it is sufficient to show that we can sort an arbitrary permutation. We provide precisely that. First, we point out that in an algorithm such as the one suggested above for the $n$-cube each processor communicates with only one other processor at each time step. Consequently, the number of processors that know the value of the maximal element can only double at each step. Thus, any such algorithm must take at least as many steps as the logarithm [6] of the number of vertices. In the case of the $n$-pancake graphs, which contain $n!$ vertices, this lower bound is $O(n \log n)$.

Consider the following word (a single word of 21 symbols broken up into many lines) on the generators of an 8-pancake graph:

$$\begin{array}{rl} f_7 f_4 f_2 & f_8 \\ f_6 f_4 f_2 & f_7 \\ f_5 f_4 f_2 & f_6 \\ f_4 f_2 & f_5 \\ f_3 f_2 & f_4 \\ f_2 & f_3 \\ & f_2 \end{array}$$

The above word has been broken up into many lines to emphasize the block structure of the word, and the groupings within each block have been appropriately indicated.

We claim that for every permutation there is a subsequence of the above word that forms a path to the identity, i.e., sorts the given permutation. To sort the permutation we first bring the largest symbol, namely 8, to the first position. This is done using a subsequence of the first three letters in the word (grouped together to indicate that). It should be clear that we can bring 8 to the first position no matter where 8 starts out. Having done that we take 8 to the last position using $f_8$. Thus, the first block of the word is sufficient to position 8 in the correct place. As the reader might have guessed the subsequent blocks of the word are each sufficient to position each of the remaining symbols in their correct places. We have proved the following general theorem:

*Theorem 8*: There is an $O(n \log n)$ algorithm to find the maximal element amongst $n!$ elements distributed among $n!$ processors located at the vertices of an $n$-pancake graph.

*Proof*: It should be clear how to generalize the word suggested above for the 8-pancake graph into a general word for the $n$-pancake graph. Further, we can also similarly prove that every permutation of $n$ symbols can be sorted by a subsequence of that word. The algorithm, then, is merely to execute one letter of that word at each time step. Executing a letter requires that every vertex communicate with the vertex connected to it along the edge corresponding to that generator and compare notes on the maximal element that it has encountered so far. Thus the (parallel) time taken by this algorithm is exactly the length of the word. It is easy to see that the length of the word is $O(n \log n)$. □

In fact, we can use the above technique to construct a binary tree of depth $O(n \log n)$ over the $n!$ vertices of the pancake graph. Every edge of this tree is an edge of the pancake graph. Thus, we can simulate a binary tree. Once we do that we can perform any computation for which the solution on the $n$-cube employs the construction of a binary tree. For example, as the following theorem states, we can compute prefixes over an arbitrary associative binary operation.

*Theorem 9*: Given any associative binary operation $*$ and an assignment of values $x_i$ to each of the $n!$ processors located at the vertices of an $n$-pancake graph, there is an $O(n \log n)$ algorithm that computes (in parallel) the prefix $x_1 * x_2 * \cdots * x_i$ at processor $i$. (The $n!$ vertices must be numbered appropriately.)

*Proof*: Omitted. □

The prefix computation problem is interesting because it is a general formulation of a class of problems. For example, the problem of computing the maximal element can be viewed as the prefix computation problem over the associative binary operation *MAX*. Other trivial applications include computing an $n$-ary associative operation, such as summation. Non-trivial applications include addition of binary numbers, (particularly, computing the carry bits), simultaneous evaluation of polynomials, etc (see [18]). The prefix computation is merely an example of how we might design algorithms on the pancake graphs. Clearly, there is considerable scope for further work in this area.

In the last two sections we have made a strong case for two new interconnection networks: the star graphs and the pancake graphs. We have shown how they compare against the $n$-cubes. We have used the $n$-cubes as our basis for comparison since (we believe) it is the most popular non-trivial interconnection network. However, the $n$-cubes themselves are Cayley graphs, as we mentioned in Section 1. Even some of the variants of the $n$-cubes, such as the cube-connected cycles, can be abstracted as Cayley graphs. However, this requires that we develop an interesting connection between group theoretic products and graph theoretic products. Unfortunately, space limitations prohibit development of that connection in this paper. Instead, we refer the curious reader to the more detailed report [19].

## 7. CONCLUSION

The main conclusion of this paper is that this group theoretic view of symmetric interconnection networks not only abstracts the structure and symmetry properties that make the n-cubes so attractive, but also offers a fertile source of other promising topologies. We can now bring to bear the algebraic tools in analyzing such symmetric networks[7]. The universality of this model, indicated by Theorem 3, allows us to present all symmetric networks in a uniform and comparable framework.

[5] A subsequence need not necessarily be a contiguous subsequence.

[6] All logarithms are in base 2.

[7] An extensive survey of the literature on the use of group theoretic techniques in analyzing symmetries in graphs can be found in [20].

Some of the immediate advantages of using Cayley graphs as a tool for modeling interconnection networks is the conciseness with which a symmetric network can be specified, i.e., by providing the set of generators for the Cayley graph. Furthermore, casting such a network as a Cayley graph immediately allows one to infer all the generic properties of Cayley graphs.

Aside from representing known interconnection networks as Cayley graphs, we have pointed out how this model offers an inexhaustible source of new topologies. In particular, the ability to start from a finite group and construct a Cayley graph is particularly attractive. We believe that we have hardly made a dent in the possible topologies that one could investigate. And even with such a limited investigation we have uncovered many topologies that compare favorably against the $n$-cubes.

The $n$-cubes have a number of attractive features. But many of these features can be interpreted as symmetry properties that many other Cayley graphs could well possess. We have indicated this at many places during the course of our presentation. A case in point is the recursive decomposition structure of Cayley graphs. That is really an analysis of the cosets of the group with respect to a given subgroup.

Finally, we have made an initial attempt at showing how we might design certain fundamental algorithms on such networks as the pancake graphs. We believe that the design of algorithms on such networks could conceivably use the algebraic properties of these graphs. There are definitely many open problems along these lines.

There are also many other features of an interconnection network that we have not at all addressed in this paper. For example, what are the issues involved in laying out these graphs? Can the $n$-star graphs be laid out at least as efficiently as the comparable $n$-cubes? We do conjecture that the $n$-star graphs can be laid out on a surface of genus (n-2).

In summary, we have offered a unified view of symmetric interconnection networks and suggested some new networks — the star graphs and the pancake graphs. We believe that we have done little more than scratch the surface of an obviously fertile field.

## REFERENCES

[1] Pease, M.C., The Indirect Binary $n$-Cube Microprocessor Array, *IEEE Trans. Comput.* C-26 (1977), pp. 458-473.

[2] Preparata, F.P. and Vuillemin, J., The Cube-Connected Cycles: A Versatile Network for Parallel Computation, *CACM* 24 (5) (1981), pp. 300-309.

[3] Samatham, M.R. and Pradhan, D.K., A Multiprocessor Network Suitable for Single-Chip VLSI Implementation, Proc. Eleventh Intl. Symp. Computer Architecture (1984), pp. 328-337.

[4] Seitz, C.L., Concurrent VLSI Architectures, *IEEE Trans. Comput.* (1984), pp. 1247-1265.

[5] Stone, H.S., Parallel Processing with the Perfect Shuffle, *IEEE Trans. Comput.* C-20 (1971), pp. 153-161.

[6] Ullman, J.D., *Computational Aspects of VLSI*, (1984), Computer Science Press, pp. 209-243.

[7] Leighton, F.T., Complexity Issues in VLSI: Optimal Layouts for Shuffle Exchange Graph and Other Networks, (1983), The MIT Press, pp 76-93.

[8] *Amer. Math Monthly* 82 (1) (1975), p. 1010.

[9] Biggs, N.L., *Algebraic Graph Theory*, (1974), Cambridge University Press.

[10] Akers, S.B. and Krishnamurthy, B., Group Graphs as Interconnection Networks, Proc. Fourteenth Intl. Conf. Fault Tolerant Computing, (1984), pp. 422-427.

[11] Leland, W.E. and Solomon, M.H., Dense Trivalent Graphs for Processor Interconnection, *IEEE Trans. Comput.* C-31 (1982), pp. 219-222.

[12] Leland, W.E., et al, High Density Graphs for Processor Interconnection, *Info. Proc. Letters* 12 (1981), pp. 117-120.

[13] Gates, W.H. and Papadimitriou, C.H., Bounds for Sorting by Prefix Reversal, *Discrete Math* 27 (1979), pp. 47-57.

[14] Herstein, I.N., *Topics in Algebra*, (1964), Blaisdell Publishing Co.

[15] Bondy, J.A. and Murty, U.S.R., *Graph Theory with Applications*, (1979), North Holland.

[16] Zaks, S., A new Algorithm for Generation of Permutations, (1981), Technical Report #220, Technion — Israel Institute of Technology.

[17] Knuth, D.E., *The Art of Computer Programming, Vol. 1*, (1973), Addison Wesley.

[18] Fich, F.E., New Bounds for Parallel Prefix Circuits, Proc. Fifteenth Annual ACM Symp. Theory of Computing, (1983), pp. 100-109.

[19] Akers, S.B., and Krishnamurthy, B., A Group Theoretic Model for Symmetric Interconnection Networks, Tektronix Laboratories Technical Report CR-86-29, (1986).

[20] Fuller, E., Krishnamurthy, B., Symmetries in Graphs: An Annotated Bibliography, Tektronix Laboratories Technical Report CR-85-03, (1986).

# RANDOMIZED PARALLEL COMMUNICATIONS

*Debasis Mitra*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*R. Cieslak**

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

*ABSTRACT*

Parallel communication algorithms are central to large scale parallel computing. This paper identifies worst-case source-destination traffic patterns, and proposes a scheme for obtaining relief by means of randomized routing of packets on simple extensions of the well-known omega networks. The communication problem is considered in two separate contexts: the non-renewal, synchronous and the renewal, asynchronous. In the non-renewal context we show that our scheme performs as well as Valiant's. The algorithm extends naturally from the non-renewal to the renewal. First, we explicitly identify the worst-case traffic intensities in the internal links of the extended omega networks over all source-destination traffic specifications which satisfy loose bounds. Second, the benefits of randomization on the stability of the network are identified. Third, exact results, for certain restricted models for sources and transmission, and approximate analytic results, for quite general models, are derived for the mean delays.

## 1. INTRODUCTION

This paper addresses the concern that particular, not rare, source-destination traffic patterns may cause unbalanced usage of the internal links of an interconnection network and thereby severely degrade performance in large parallel computers [1,2]. Worst-case traffic patterns and their effects are identified, and a scheme for obtaining relief by means of randomized routing on simple extensions of the omega network is proposed and analyzed. Another feature of this work is that the main performance results are given as they depend on a parameter $r$ which measures both the degree of extension of the omega network and the degree of randomization. The omega network [3] corresponds to $r = 0$ and the maximum value of $r$ corresponds to the *fully extended omega network* which has one less than twice as many stages as the omega network.

The communication problem with $N$ sources and $N$ destinations is considered in two separate contexts, the *non-renewal* and the *renewal*. The non-renewal context was introduced by Valiant [4-6] and is a model for synchronous communications. Here the task is to complete a *partial h-relation* [4] in which initially each of the sources have at most $h$ packets, with no destination occurring on more than $h$ packets. The assumption is that all link queues are initially empty and that no packets are allowed into the network while the task is in progress. Valiant exhibited an algorithm on a $\log N$-dimensional hypercube which completed the task in time $O(\log N)$ with overwhelming probability. Now, the hypercube requires switches of degree $\log N$ which is unbounded. The algorithm that we give uses switches of fixed degree and also it requires no scheduling [6] — the queue discipline is throughout first-come-first-served. For the fully extended omega network and for the same non-renewal context, we prove that the scheme has as good delay characteristics as has been proven for Valiant's scheme.

The main results of this paper are on the non-renewal context in which packets are generated, and allowed into the network, continually and asynchronously. Here the main interest is in the stationary throughput-delay characteristics [7] for various traffic patterns. The packets at each source are assumed to form stationary renewal processes. The scheme carries over naturally from the non-renewal context. (Valiant's scheme is not defined for this context.) It is not tied to the omega network with 2×2 switches, which we have chosen to be specific, and other pipelined networks with switches of fixed degree can serve just as well. The randomized routing algorithm requires each packet to be given a statistically independent *scattering ticket*. A *distributed switching rule* implements the routing.

The analysis in the non-renewal context considers *partial* $\lambda$*-relations* in which the intensity, or average rate, at each source is no more than $\lambda$ and the intensity at each destination is no more than $\lambda$. We first give an explicit characterization of the extremal traffic on the internal links of the extended omega

* Research supported by JSEP Contract F49620-79-C-0178 and ONR Contract N00014-80-C-0507.

networks. The mean delay is nonlinear in $\lambda$ and rapidly goes to $\infty$ as $\lambda$ approaches a stability boundary. Randomization has the effect of extending the region of stability. The stationary mean time from source to destination is characterized for various stochastic models of the source, link transmission time and the degree of randomization. In particular, on the fully extended omega network this mean time is asymptotically proportional to $\log N$ in the region of stability.

In conclusion, the results of the present worst-case analysis argue strongly for maximum randomization. This paper reports partial results; other results and proofs may be found in [13].

## 2. THE EXTENDED OMEGA NETWORK AND RANDOMIZED ROUTING

The *extended omega network* is obtained by preceding an omega network of $n$ stages ($N = 2^n$) by a *scattering network* with $r$ stages, $1 \leqslant r \leqslant n-1$. See Figure 1 for an example with $n=3$ and $r=1$. At each stage of the scattering network, traffic entering a switch is, on the average, routed to each output port evenly. Scattering is accomplished by assigning a *scattering ticket* to each packet at the source. Each scattering ticket is a binary $r$-tuple $c_r, \ldots, c_1$ in which each element is obtained from the result of a completely independent trial with equiprobable outcomes. A packet with scattering ticket $c_r, \ldots, c_1$ is routed to the output port with the least significant bit of its address given by $c_i$ at the $i^{\text{th}}$ stage of the scattering network. Thereafter the packet is routed through the omega network as usual.

We describe an indexing system for the stages and links in the extended omega network. Of the $n+r$ stages, the one closest to the sources is called the first stage. The ports on both sides of each switch are indexed identically, with all port addresses being $n$-bit binary sequences $a_n, \ldots, a_1$ with zero at the top.

Each link is given an address with the format $(i; a_n, \ldots, a_1)$, where $i$ denotes the stage at which it originates and $a_n, \ldots, a_1$ is the address of the port in the switch at the $(i+1)^{\text{th}}$ stage where the link terminates. The wiring connects output ports $a_n, \ldots, a_1$ at the $i^{\text{th}}$ stage to the input port $a_1, a_n, \ldots, a_2$ at the $(i+1)^{\text{th}}$ stage. The link that makes this connection is, by our previously stated rule, $(i; a_1, a_n, \ldots, a_2)$. The net effect is that a packet on link $(i; a_n, \ldots, a_1)$ on entering a switch in the $(i+1)^{\text{th}}$ stage is switched to output port $b \in \{0,1\}$, and leaves on the link $(i+1; b, a_n, \ldots, a_2)$. Each link is equipped with a queue at its originating port, which for the purposes of this paper is of unlimited capacity.

For each packet originating at source $s_n, \ldots, s_1$ with destination $d_n, \ldots, d_1$ and scattering ticket $c_r, \ldots, c_1$ let

$$\mathbf{R} \triangleq d_n, \ldots, d_1, c_r, \ldots, c_1, s_n, \ldots, s_1 .$$

Now consider a window of width $n$, i.e. it exposes $n$ consecutive numbers of $\mathbf{R}$. The window therefore has $n+r+1$ positions. Number these positions 0 to $n+r$, where position 0 is right most and an increment in position corresponds to sliding the window by one unit to the left. *Fact*: the link used by the packet at the $i^{\text{th}}$ stage, $1 \leqslant i \leqslant n+r$, has address given by the values in $\mathbf{R}$ which are exposed by the window at the $i^{\text{th}}$ position. This Fact is very useful in inferring the traffic carried by a particular link.

The above is consistent with the following distributed switching rule: while the packet is at the switch in the $i^{\text{th}}$ stage, route it to the output port whose address has for its least significant bit

$$\begin{array}{ll} c_i & 1 \leqslant i \leqslant r, \\ d_{i-r} & r+1 \leqslant i \leqslant n+r. \end{array}$$

## 3. TRAFFIC ANALYSIS: THE NON-RENEWAL CASE

The probabilistic bound given here holds for *partial h-relations* [7], see Section 1. The assumption is made in this section that each link transmits one packet in unit time.

The extended omega networks are *not* non-repeating [4] and *not* non-overtaking [8]. That is, it is possible for two packets to use a common link at stage $i_1$, different links at stage $i_2$ and a common link at stage $i_3$, for some $i_1$, $i_2$, $i_3$ where $1 \leqslant i_1 < i_2 < i_3$. Note however that there is partial non-overtaking. To see this split the extended omega network with $(n+r)$ stages into two halves. From the Fact in Section 2 it is easy to see that each isolated half is non-overtaking.

Specifically, with

$$\hat{\imath} \triangleq \left\lfloor (n+r)/2 \right\rfloor, \tag{3.1}$$

each packet's transit through links in stages 1 to $\hat{\imath}$ is called its Phase 1 and its transit from stage $(\hat{\imath}+1)$ to $(n+r)$ is called its Phase 2. The time to complete each phase by all packets in a partial $h$-relation is, in turn, bounded below with the help of the non-overtaking property in each phase.

Consider first Phase 1; Phase 2 is similar. The routes of two packets are said to intersect in Phase 1 if at least one link in stages 1 to $\hat{\imath}$ is common to both routes. Let $X$ be a marked packet. The following facts are proven in [13] to hold regardless of the value of $r$: if for each distinct packet other than $X$ we consider a corresponding trial in which success denotes that the packet intersects the route of $X$ in Phase 1, then these trials are statistically *independent*; the expected number of packets which intersect $X$ in Phase 1 is at most $\sigma h/2$, where

$$\sigma = r - 1 + 2^{1+\left\lfloor (n-r)/2 \right\rfloor}.$$

Note that in the fully extended omega network $\sigma = n$. Then by using standard techniques [4] we prove the following in [13]:

*Theorem 1:* (i) For any $\Delta \geqslant \sigma h/2$,

$$Pr[\text{time to complete Phase 1} \geqslant (n+r)/2 + \Delta] \leqslant Nh \left( \frac{e\sigma h}{2\Delta} \right)^{\Delta} e^{-\sigma h/2}.$$

$$Pr\left[\text{time to complete Phase 2} \geqslant \frac{n+r+1}{2} + h + \Delta\right] \leqslant Nh \left( \frac{e\sigma h}{2\Delta} \right)^{\Delta} e^{-\sigma h/2}.$$

(ii) *Asymptotics:* for $r = n-1$, each phase in any partial $h$-relation is completed in time $O(\log N)$ with overwhelming probability. In particular, for any $K \geqslant e$,

$$Pr[\{\text{time to complete each phase}\} \geqslant (n+r+1)/2+h+Khn]$$

$$\leqslant hN^{-(Kh-1)}$$

For $r = \gamma n$, $\gamma$ fixed and less than unity, each phase in any partial $h$-relation is completed in time $O\left[N^{(1-\gamma)/2}\right]$ with overwhelming probability.

## 4. TRAFFIC ANALYSIS: THE RENEWAL CASE

In the renewal context, at each source $s_n, \ldots, s_1$ the packets to be delivered to destination $d_n, \ldots, d_1$ form a stationary stochastic process with mean rate, or intensity, $\lambda(s_n, \ldots, s_1; d_n, \ldots, d_1)$. A particular case is Poisson processes at the sources. The matrix of these traffic intensities over all sources and destinations constitute the *source-destination traffic specification.* Unlike the treatment in Section 3, here we allow for statistical fluctuations in the time required by each link to transmit a packet, although special consideration is given to the case of constant link transmission time. We assume that when the link transmission times are random, they are, for all links and packets, mutually independent, independent of the traffic and picked from a common distribution with mean 1.

In a partial $\lambda$-relation,

$$\sum_{d_n, \ldots, d_1} \lambda(s_n, \ldots, s_1; d_n, \ldots, d_1) \leqslant \lambda, \quad \forall\, s_n, \ldots, s_1 .$$

$$\sum_{s_n, \ldots, s_1} \lambda(s_n, \ldots, s_1; d_n, \ldots, d_1) \leqslant \lambda, \quad \forall\, d_n, \ldots, d_1 .$$

In full $\lambda$-relations, the above holds with equality. Hereafter, in this section, we exclusively consider full $\lambda$-relations since the results on delay statistics for it are obvious bounds for the partial $\lambda$-relations.

### *4.1 Link Traffic Intensities*

Here $t(i; a_n, \ldots, a_1)$ denotes the traffic intensity, i.e. the mean number of carried packets per unit time, on link $(i; a_n, \ldots, a_1)$. Also, $T(\lambda,n,r)$ denotes the extremal traffic intensity over all links and all traffic specifications satisfying the full $\lambda$-relations, i.e.

$$T(\lambda,n,r) \triangleq \max_{\text{full}\lambda\text{-relation}} \; \max_{i; 1 \leqslant i \leqslant (n+r)} \; \max_{a_n, \ldots, a_1} \; t(i; a_n, \ldots, a_1) .$$

We say that the traffic is *symmetric* if the traffic intensity is identical in all links of the network.

We will find useful the notion of traffic class. Each traffic class is indexed by source, destination and scattering ticket. Each traffic class has an associated traffic intensity and quite obviously the traffic intensity of the class indexed by $s_n, \ldots, s_1; d_n, \ldots, d_1; \quad c_r, \ldots, c_1$ is $\lambda(s_n, \ldots, s_1; d_n, \ldots, d_1)/2^r$.

The traffic intensity $t(i; a_n, \ldots, a_1)$ is obtained by summing the traffic intensity of all classes with routes which include the link $(i; a_n, \ldots, a_1)$. The Fact in Section 2 may be used to calculate this quantity.

The following theorem and its corollary is a summary of our results on the extremal link traffic intensities, both as seen by a packet following a specific route and as seen by an independent observer of traffic in the extended omega network. The proof is in [13].

*Theorem 2:* (i) for full $\lambda$-relations and for all binary $n$-tuples $(a_n, \ldots, a_1)$,

$$t(i; a_n, \ldots, a_1) = \lambda, \quad 1 \leqslant i \leqslant r \text{ and } n \leqslant i \leqslant n+r \text{-(4.1)}$$

If $r = n-1$ then the above specifies the traffic intensities on all links of the extended omega network. If $r < n-1$,

$$t(i; a_n, \ldots, a_1) \leqslant \lambda 2^{\min(i-r,\, n-i)}, \quad r+1 \leqslant i \leqslant n-1 \text{(4.2)}$$

(ii) If $r < n-1$ then there are source-destination traffic specifications satisfying the full $\lambda$-relations such that the traffic intensity on each of the links used in the route followed by a particular class is given by (4.1) and (4.2), with equality holding in (4.2).

*Corollary:* (i)

$$T(\lambda,n,r) = \lambda 2^{\lfloor (n-r)/2 \rfloor} .$$

(ii) The traffic is symmetric for all source-destination traffic specifications satisfying the full $\lambda$-relations if and only if $r = n-1$.

In [13] we construct source-destination traffic specifications for which the traffic intensity on links is as bad as asserted in statement (ii) of the Theorem. The point of the construction is to show that they are not rare.

*Example:* Consider the extended omega network in Figure 1 in which $n=3$, $r=1$. Suppose each source transmits at the average rate of $\lambda$ packets per unit time, and, on the average, source $s_3, s_2, s_1$ sends a fraction $\alpha(s_3,s_2)$ of its packets to destination $s_1, s_2, s_3$ and the remainder to $\bar{s}_1, s_2, s_3$, where the bar denotes bit reversal. The numbers $\alpha(0,0)$, $\alpha(0,1)$, $\alpha(1,0)$, $\alpha(1,1)$ are arbitrary in $[0,1]$.

Note that each destination receives packets at the average rate of $\lambda$ packets per unit time. The resulting traffic intensities on the internal links of the network are as shown on the links in Figure 1. The links of stage 2 either have traffic intensity $2\lambda$ or carry no traffic. For general $n$ and $r$, with $(n+r)$ even, there are many source-destination traffic specifications for which $2^{(n+r)/2}$ links at stage $(n+r)/2$ carry traffic of intensity $\lambda 2^{(n-r)/2}$, and the remaining links of the stage carry no traffic at all.

### *4.2 Stability With Respect to Full $\lambda$-Relations*

The stability condition of interest here is that at each link the traffic intensity is less than the mean link transmission time of packets. Our primary interest is in *stability with respect to full $\lambda$-relations*, i.e. stability for all source-destination traffic specifications which satisfy the full $\lambda$-relation. From the Corollary to Theorem 2, the condition for stability with respect to full $\lambda$-relations is

$$\lambda\, 2^{\lfloor (n-r)/2 \rfloor} < 1 .$$

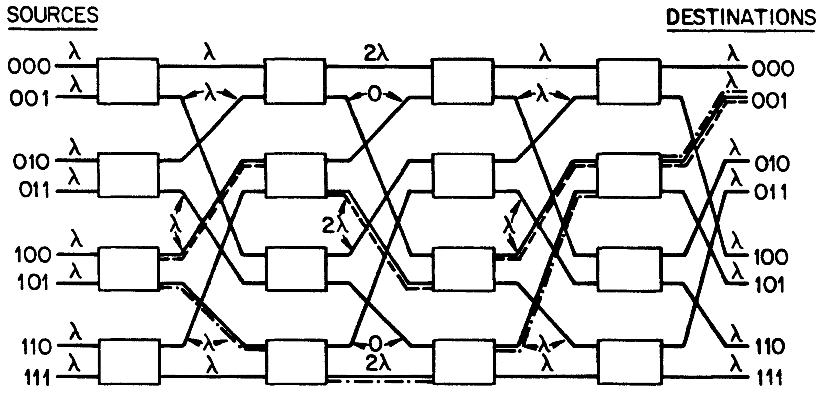


Figure 1: An extended omega network with n=3, r=1. The path from source 101 to destination 001 of packets with scattering ticket $c_1$=1 is shown by chained line, and of packets with $c_1$=0 by dashed line. The inscriptions on links are traffic intensities for Example in Section 4.1.

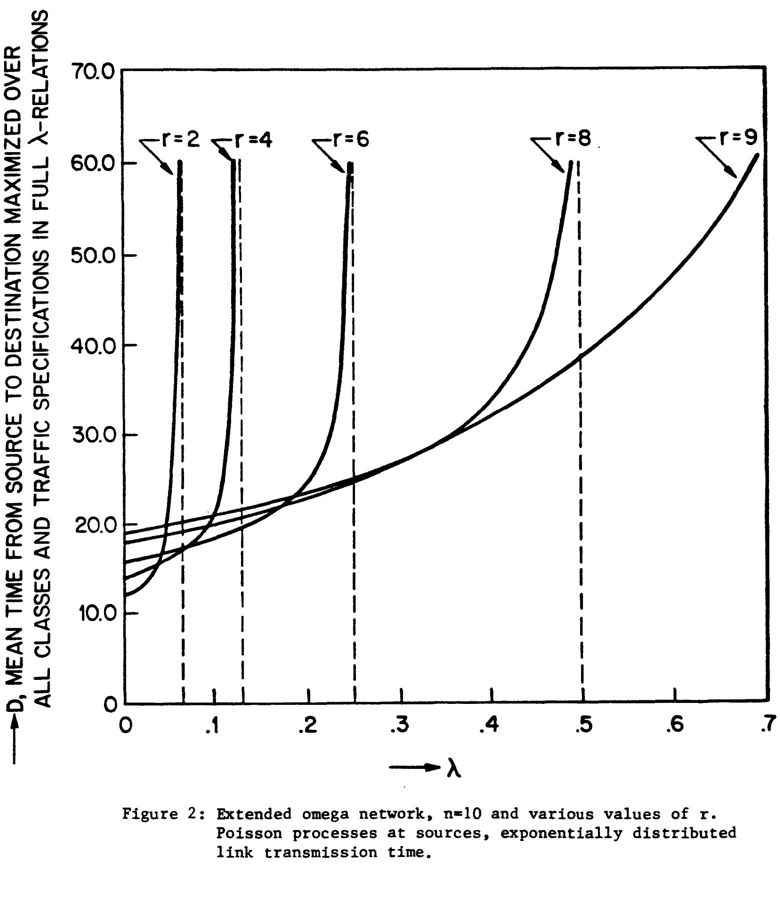


Figure 2: Extended omega network, n=10 and various values of r. Poisson processes at sources, exponentially distributed link transmission time.

For the omega network $r = 0$, and the stability condition is $\lambda < 1/\sqrt{N}$ if $\log_2 N$ is even, and $\lambda < \sqrt{2/N}$ if $\log_2 N$ is odd. For the fully extended omega network $r = n-1$, and the stability condition is $\lambda < 1$.

### *4.3 Mean Delay for Various Degrees of Extensions of the Omega Network.*

Here we make the restrictive assumption that the packets at the sources form Poisson processes and that the link transmission times are random variables with exponential distributions. The exponential has the advantage that exact delay statistics are simple to obtain. The well-known result which makes the problem tractable is that, in equilibrium, the joint distribution of packets of all classes and in all links is of the same form as if the arrival processes to the links are Poisson. Therefore,

$$W(i; a_n, \ldots, a_1) = \text{mean queueing delay at link } (i; a_n, \ldots, a_1)$$

$$= \frac{1}{1 - t(i; a_n, \ldots, a_1)} - 1, \qquad (4.3)$$

where $t(\ )$ is the traffic intensity on the link.

For each class, let

$$D(\text{class}) \triangleq \text{mean time from source to destination for class packets,}$$

$$= \sum_{\text{link} \in \text{class route}} \{W(\text{link}) + 1\}.$$

Finally, let

$$D \triangleq \max_{\text{full}\lambda\text{-relations}} \ \max_{\text{class}} D(\text{class}).$$

From statement (ii) of Theorem 2 we obtain

*Proposition:* With randomized routing in the extended omega network with $n+r$ stages, $1 \leqslant r \leqslant n-1$,

$$D = \frac{2r+1}{1-\lambda} + 2 \sum_{i=1}^{(n-r-1)/2} \frac{1}{1-\lambda 2^i}, \quad \text{if } (n+r) \text{ is odd}$$

$$= \frac{2r+1}{1-\lambda} + \frac{1}{1-\lambda 2^{(n-r)/2}} + 2 \sum_{i=1}^{(n-r-2)/2} \frac{1}{1-\lambda 2^i}, \quad \text{if } (n+r) \text{ is even}$$

Figure 2 plots $D$ against $\lambda$ for various values of $r$ when $n = 10$. This figure indicates that, unless $\lambda$ can be *a priori* restricted to be quite small, the worst-case analysis of this paper argues in favour of maximum randomization.

### *4.4 Approximate Analysis for General Renewal Sources and General Transmission Time Distributions in the Fully Extended Omega Network.*

Here we depart from the preceding treatment of the renewal case by assuming that the packets at the sources form general stationary renewal processes, and the distribution of the link transmission time to be general. Poisson sources and constant link transmission time are, in particular, allowed. However, the analysis is approximate. The program that we follow is based on a proposal of Kuehn [9]. The program decomposes the network into GI/G/1 queues and relates the descriptors of the processes associated with each queue by using prior results [10,11] on GI/G/1 queues. The program undertakes (i) the approximation of the departure process from each link queue as a stationary renewal process, and (ii) the description of each stationary renewal process by two moments, the mean i.e. traffic intensity, and the coefficient of variation of the underlying interarrival distribution. In particular, the distribution of the link transmission time is specified by its mean, which is 1, and its coefficient of variation,

$$c_\ell^2 \triangleq \text{variance of link transmission time.}$$

For constant transmission time $c_\ell = 0$ and for exponential distributions $c_\ell = 1$. Also, the stationary renewal processes at the sources are described by their mean $\lambda$, and the common coefficient of variation $c_s$,

$$c_s^2 \triangleq \frac{\text{variance of interarrival time of packets at source.}}{\lambda^2}$$

For Poisson sources $c_s = 1$. We mention that a large body of validations and corroborations with analytic solutions and simulations have been reported [9-12].

We only consider the fully extended omega network, i.e. $r = n-1$; extensions to $r < n-1$ are easy. In this case traffic is symmetric and we are able to obtain simple, explicit relations, in the form of recursions, for all traffic descriptors in the network. Specifically: (i) the departing process from a link queue in stage $i$ is approximated by a stationary renewal process with coefficient of variation $c_{d,i}$; (ii) this process is split into two stationary renewal processes with means $\lambda/2$ and coefficient of variation $c_{i,i+1}$; (iii) two processes described by $(\lambda/2, c_{i-1,i})$ are superimposed to form the arrival process described by $(\lambda, c_{a,i})$ at a link queue in stage $i$.

We now give three relations to correspond to the above. From [10] we have the approximation

$$c_{d,i}^2 = c_{a,i}^2 + 2\lambda^2 c_\ell^2 - \lambda^2 (c_{a,i}^2 + c_\ell^2)\, g(\lambda, c_{a,i}^2, c_\ell^2),\ i \geqslant 1 \qquad (4.4)$$

where,

$$g(\lambda, c_{a,i}^2, c_\ell^2) = \exp\left\{\frac{-2(1-\lambda)}{3\lambda}\ \frac{(1-c_{a,i}^2)^2}{(c_{a,i}^2 + c_\ell^2)}\right\} \text{ if } c_{a,i} \leqslant 1,$$

$$= \exp\left\{-(1-\lambda)\ \frac{(c_{a,i}^2 - 1)}{(c_{a,i}^2 + 4c_\ell^2)}\right\} \text{ if } c_{a,i} > 1.$$

Decomposition, or splitting, of renewal processes yields [9],

$$c_{i-1,i}^2 - 1 = \frac{1}{2}(c_{d,i-1}^2 - 1),\ i \geqslant 2. \qquad (4.5)$$

Finally, for superposition we use the following approximation [12],

$$c_{a,i}^2 - 1 = \frac{(c_{i-1,i}^2 - 1)}{1 + 4(1-\lambda)^2},\ i \geqslant 2. \qquad (4.6,\text{i})$$

Also, there is the initial condition,

$$c_{a,i}^2 = c_s^2,\ i = 1. \qquad (4.6,\text{ii})$$

Equations (4.4)-(4.6) define a complete recursive system which yields $\{c_{a,i}, c_{d,i}, c_{i,i+1}; i \geqslant 1\}$. Note that the dimension of the network given by $n$ does not appear in the system.

The *mean queueing delay*, $W_i$ for a link queue at stage $i$ (queueing delay excludes transmission time) is given by the following exact relation for GI/G/1 [11]:

$$W_i = \frac{c_{a,i}^2 + 2\lambda^2 c_\ell^2 - c_{d,i}^2}{2\lambda(1-\lambda)}, \quad i \geqslant 1. \tag{4.7}$$

The above together with Little's Law gives the mean queue lengths at each link. Finally, the *mean time from source to destination*, *D*, is given by

$$D = (2n-1) + \sum_{i=1}^{2n-1} W_i\,. \tag{4.8}$$

The first term is from the transmission time and the second is from the cumulative queueing delay. Note that (4.8) is the only point in the procedure where $n$ is used.

Tables 1 and 2 give numerical values for the case of Poisson sources and constant link transmission time. The important observation is that for each $\lambda$, the queueing delay shrinks at the early stages before rather quickly reaching an asymptotic value. For any fixed, positive $\lambda$, $c_s^2$ and $c_\ell^2$, the asymptotic values of $c_{a,i}^2$, $c_{d,i}^2$ and $c_{i-1,i}^2$, as $i \to \infty$, are obtained simply as fixed points of the recursions in (4.4)-(4.6). These may be substituted in (4.7) to obtain the asymptotic value $\overline{W}(\lambda, c_s^2, c_\ell^2)$, where $W_i \to \overline{W}(\lambda, c_s^2, c_\ell^2)$. From (4.8), as $n \to \infty$, we obtain the noteworthy relation

$$\boxed{D \sim 2\{1 + \overline{W}(\lambda, c_s^2, c_\ell^2)\}\, n\,.}$$

The last row of Table 1 gives the computed values of $\overline{W}(\lambda, 1, 0)$ for various values of $\lambda$.

| | Mean Queueing Delay ($W_i$) | | | |
|---|---|---|---|---|
| | $\lambda = 0.2$ | $\lambda = 0.4$ | $\lambda = 0.6$ | $\lambda = 0.8$ |
| Stage 1 | 0.125 | 0.333 | 0.750 | 2.000 |
| Stage 2 | 0.124 | 0.322 | 0.667 | 1.448 |
| Stage 3 | 0.124 | 0.320 | 0.648 | 1.345 |
| Stage 4 | 0.124 | 0.319 | 0.643 | 1.327 |
| Stage $i$, $i > 4$ | 0.124 | 0.319 | 0.642 | 1.324 |

*Table 1*: For $r = n-1$, and full $\lambda$-relations; Poisson processes at sources and unit time for transmission across each link. Calculated from (4.4)-(4.7).

| n | Mean Time from Source to Destination (*D*) | | | |
|---|---|---|---|---|
| | $\lambda = 0.2$ | $\lambda = 0.4$ | $\lambda = 0.6$ | $\lambda = 0.8$ |
| 4 | 7.870 | 9.254 | 11.637 | 17.093 |
| 6 | 12.366 | 14.533 | 18.207 | 26.390 |
| 8 | 16.863 | 19.811 | 24.777 | 35.686 |
| 10 | 21.360 | 25.090 | 31.346 | 44.983 |
| 12 | 25.857 | 30.368 | 37.916 | 54.279 |
| 14 | 30.353 | 35.647 | 44.486 | 63.575 |

*Table 2*: Specification as in Table 1. Calculated from (4.4)-(4.8).

The procedure based on (4.4)-(4.8) with $c_s = c_\ell = 1$ gives correct values for Poisson sources and exponentially distributed link transmission time. In this case, $\overline{W} \equiv W_i = \lambda/(1-\lambda)$. Also, the numerical results for $c_\ell$ in the range [0,1] are bounded by the results for $c_\ell = 0$ and $c_\ell = 1$ in the case of Poisson sources.

### 4.4.1 *Accuracy of Approximation, Simulations*

Comparison with simulations has shown that for constant link transmission time and light traffic, i.e. small $\lambda$, the decomposition overestimates the queueing delay in all stages beyond the first. On the other hand, the approximation is good in heavy traffic. The net effect of the inaccuracies on the mean time from source to destination in the fully extended omega network is small. This is, of course, because in light traffic this time is dominated by the transmission time. Also as expected, the largest errors occur at about $\lambda = 0.6$. Generally, the errors are smaller as we depart from constant link transmission time and add more variability to it.

As previously observed, the procedure in (4.4)-(4.8) yields mean queueing delays which converge quickly, with increasing stages, to asymptotic values. This important fact may be used to simplify simulations. Thus, an omega network of small dimension, say with four stages, may be simulated, and the mean delay observed for the final stage may be taken to apply to all subsequent stages in calculations for large extended omega networks. We have undertaken the program just outlined. The results of the simulation of the four-stage omega network for Poisson sources and unit link transmission time are given in Table 3. The results of the extrapolation to large, fully extended omega networks are given in Table 4, which should be considered a refinement of Table 2.

| | Mean Queueing Delay | | | |
|---|---|---|---|---|
| | $\lambda = 0.2$ | $\lambda = 0.4$ | $\lambda = 0.6$ | $\lambda = 0.8$ |
| Stage 1 | 0.123 | 0.335 | 0.751 | 2.002 |
| Stage 2 | 0.067 | 0.199 | 0.478 | 1.339 |
| Stage 3 | 0.066 | 0.196 | 0.468 | 1.318 |
| Stage 4 | 0.067 | 0.195 | 0.461 | 1.323 |

*Table 3*: From simulations of omega network with 4 stages for Poisson packet traffic at sources and unit time for transmission across each link.

| n | Mean Time from Source to Destination | | | |
|---|---|---|---|---|
| | $\lambda = 0.2$ | $\lambda = 0.4$ | $\lambda = 0.6$ | $\lambda = 0.8$ |
| 4 | 7.529 | 8.512 | 10.544 | 16.954 |
| 6 | 11.801 | 13.292 | 16.390 | 26.249 |
| 8 | 16.072 | 18.073 | 22.235 | 35.544 |
| 10 | 20.344 | 22.854 | 28.081 | 44.839 |
| 12 | 24.615 | 27.635 | 33.927 | 54.134 |
| 14 | 28.887 | 32.416 | 39.772 | 63.428 |

*Table 4*: Extrapolated from data in Table 3. Compare with Table 2.

## ACKNOWLEDGMENTS

It is with gratitude that one of the authors (D.M) acknowledges the benefit of discussions with colleagues at A.T.T. Bell Laboratories, specifically, Dr. P. Semal (visiting from Philips Research Lab., Brussels), Dr. L. A. Shepp, and Dr. A. Weiss. The assistance of Dr. K. G. Ramakrishnan and Ms. J. B. Seery in the simulations is gratefully acknowledged.

## REFERENCES

1. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. Computers, vol. C-32, No. 2, Feb. 1983, pp. 175-189.

2. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," Proc. 1985 Intl. Conf. Parallel Processing, pp. 764-771.

3. G. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor System," 1st Ann. Symp. on Computer Architecture, 1973, pp. 21-28.

4. L. G. Valiant and G. J. Brebner, "Universal Schemes for Parallel Communication," Proc. 13th Annual Symp. on Theory of Computing, 1981, pp. 263-277.

5. E. Upfal, "Efficient Schemes for Parallel Communication," Proc. Symp. on Principles of Distributed Computing, 1982, pp. 55-59.

6. R. Aleliunas, "Randomized Parallel Communication," Proc. Symp. on Principles of Distributed Computing, 1982, pp. 60-72.

7. C. P. Kruskal, M. Snir and A. Weiss, "On the Distribution of Delays in Buffered Multistage Interconnection Networks for Uniform and Nonuniform Traffic," Proc. 1984 Intl. Conf. on Parallel Processing, pp. 215-219.

8. J. Walrand and P. Varaiya, "Sojourn Times and the Overtaking Condition in Jacksonian Networks," Adv. Appl. Prob., vol. 12, 1980, pp. 1000-1018.

9. P. J. Kuehn, "Approximate Analysis of General Queueing Networks by Decomposition," IEEE Trans. Communications, vol. COM-27, no. 1, Jan. 1979, pp. 113-126.

10. W. Kraemer and M. Langenbach-Belz, "Approximate Formulae for the Delay in the Queueing System GI/G/1," Congressbook 8th Intl. Teletraffic Congress, Melbourne, 1976, pp. 235-1/8.

11. K. T. Marshall, "Some Inequalities in Queueing," Oper. Res., vol. 16, 1968, pp. 651-665.

12. W. Whitt, "The Queueing Network Analyzer," Bell System Tech. J., vol. 62, no. 9. part 1, Nov. 1983, pp. 2779-2816.

13. D. Mitra and R. Cieslak, "Randomized Parallel Communications on an Extension of the Omega Network," A.T.&T. Bell Laboratories Memorandum, March 1986.

# A Comparison of Two Synchronization Primitives in an Operating System for Parallel Processing Applications

J. Eric Roskos
Concurrent Computer Corporation
Southern Development Center
2486 Sand Lake Road
Orlando, FL 32809-7642
*jer@peora.CCUR.UUCP*

Abstract — It has been claimed [REED79] that the synchronization primitives termed *eventcounts* and *sequencers* may be used to implement semaphores, and thus that they are "primitives at a lower level than semaphores". However, by expressing the functionality of conventional semaphores, and that of semaphores implemented via eventcounts and sequencers, via a common set of primitives, it may be shown that in fact the eventcount/sequencer pair imposes restrictions on process synchronization that do not exist with the traditional semaphore. Two such restrictions exist. First, the eventcount/sequencer pair imposes an ordering on the activation of suspended processes whereas semaphores activate the processes nondeterministically. Second, it is difficult to implement a "conditional **P**" operation with eventcounts and sequencers because one cannot return a ticket obtained from a sequencer to the ticket pool: a ticket, once obtained, must be used. Thus, the claim that eventcounts and sequencers are lower-level primitives than the traditional semaphore is contradicted by this analysis.

**Introduction**

D. P. Reed and R. K. Kanodia, in their 1979 paper "Synchronization with Eventcounts and Sequencers" [REED79] present a new process synchronization method with two primitives, **advance** and **await**. A process may **await**($E, V$) an eventcount $E$'s reaching a certain value $V$; the eventcount is made to reach this value eventually by other processes performing an **advance**($E$). A separate **ticket**($S$) primitive is defined to give unique, consecutive, increasing integer vaules on each invocation.[1]

The authors observe that eventcounts and sequencers are "primitives at a lower level than semaphores," and that semaphores can be built out of them as a result. They also demonstrate that some more powerful operations on semaphores can be built.

Concurrent Computer Corporation is presently involved in research and development on primitives to be used in synchronizing OS-related operations on general-purpose, symmetrical, parallel processors. In investigating the synchronization primitives available in the literature, we chose Eventcounts and Sequencers as a synchronization primitive that would be compatible with our planned OS and hardware architecture. During the course of the implementation, however, it was found that problems existed which would have not occurred if conventional semaphores had been used, particularly in situations in which entry to a critical section was desired only if such entry would not cause suspension of the process. These situations tend to be common in real-time applications, including our application of I/O with a goal of maximum parallelism in tightly-coupled multiprocessors. In such an application, precise process synchronization is required, but delaying of a process which cannot enter a critical section is not desirable: the process often is able to perform some other task until the critical section becomes available. Certain deadlock avoidance algorithms also needed to function in this way.

As a result of the apparent limitations in eventcounts and sequencers, the mechanisms underlying these primitives were examined using methods previously employed in investigating interprocess communication primitives in multiprocessors [ROSK84]. This paper demonstrates that the implementation of semaphores given in REED79 does not provide the flexibility of Dijkstra's original primitives [DIJK68] due to the combining of two required primitives, those for *token-oriented synchronization* and *sequencing*, into the one **ticket** primitive. This is done by expressing both in terms of comparable primitive operations; the differences then become apparent.

[1] A third eventcount primitive, **read**($E$), returns the current value of the eventcount.

0190-3918/86/0000/0231 $01.00 © 1986 IEEE

### Process Sequencing/Suspending Primitives

We first define two primitives which may be used to effect the process suspending and resumption needed in order to cause process synchronization. There is one suspension primitive, which stops a process, and assigns it a specified integer "tag," and one resumption primitive, which selectively resumes all processes having a given tag.

**stop**($p$,$L$).

Let **stop**($p$,$L$) denote an operation which places the process executing it on a list L, assigning an associated integer "process group tag" $p$. Execution of the process is then suspended. The process group tag is simply an integer which may be used to selectively resume the process or group of processes which have been assigned that tag.

**start**($L$,$p$).

If the set of processes in $L$ with tag $p$ is nonempty, the processes with tag $p$ are removed from $L$. Execution of the processes thus removed are resumed.

### Token-Oriented Primitives

We next define a set of *token-oriented primitives* used to arbitrate access to critical sections of code.

Let T be a set of $n$ **tokens**, $t_i$, $1 \leq i \leq$n.

**g**($T$).

Let **g**($T$) denote a **get** operation defined as follows. If $T \neq \emptyset$, **g**($T$) returns an element $t_j \in T$, with $j$ arbitrary; and indivisibly sets $T \overset{j}{\leftarrow} T - \{t_j\}$.

*If* $T = \emptyset$, **g**($T$) returns the null value $\phi$.

**p**($T$,$t$).

Let **p**($T$,$t$) denote a **put** operation which sets $T \leftarrow T \cup \{t\}$.

**ticket**($T$).

Let the number of tokens in a set $T$ be countably infinite; and let **name**($t$) denote an integer which may be arbitrarily assigned to a token $t \in T$. Then let **ticket**($T$) return **g**($T$), such that if on a given invocation of **ticket**($T$), **name**(**ticket**($T$)) = $i$, then the next consecutive invocation yields **name**(**ticket**($T$)) = $i$+1; and on the initial invocation, **name**(**ticket**($T$)) = 0.

### Comparison of Semaphores with Eventcounts/Sequencers

We can now compare Dijkstra's original semaphore operations [DIJK68] with the implementation of semaphores described by Reed and Kanodia.

Dijkstra's semaphore operations may be defined in terms of the above primitives as follows. Assume for simplicity that $t$ is a global variable in the calling process' private address space, and thus is part of the program state of the process invoking the synchronization primitive.

As is the case for most semaphore implementations, the sequences of primitive operations comprising the P and V operations are themselves critical sections: wherever a process has a copy of the state of some outside object (a pool of tokens, or a counter) in a local variable, the actual state of the object cannot be changed as long as the process will in the future act upon the copy of the state which it has in its posession. This is an instance of the implicit **copy** operation that occurs when a conventional memory is read, as discussed in ROSK84.

```
P(S):
        repeat
            t := g(S.tokens);
            if t=$\phi$ then stop(0,S.queue);
        until t<>$\phi$;

V(S):
        p(t, S.tokens);
        start(0,S.queue);
```

In the above definition, a **V** causes all waiting processes to briefly awaken and attempt to get a token; but only one will succeed for each execution of **V**, so the others remain blocked. An alternative (and more commonly seen) implementation would be only to awaken one arbitrary suspended process for each execution of **V**; but the synchronization results are equivalent, and the latter approach complicates comparison with the eventcount-based semaphores.

The eventcount/sequencer semaphore implementation as presented by Reed and Kanodia may likewise be defined in terms of the above primitives as follows. Assume that prior to the first **V**, *S.count* is initialized to zero.

```
$P_{ES}$(S):
        p := ticket(S.sequencer);
        if S.count < name(p) then
            stop(name(p),S.queue);

$V_{ES}$(S):
        S.count := S.count+1;
        start(S.queue,S.count);
```

Clearly, these are not equivalent semaphore operations. Dijkstra's semaphore primitives do not employ the priority queueing portion of the **stop** primitive, and arbitrate the stop/proceed decision using anonymous tokens which are returned to a common pool of tokens after use. The eventcount/sequencer

[2] This may be generalized by making $t$ a variable denoting a set, with the assignment to t adding a member to the set, and the reference in **P(t,S.token)** removing one member from this set.

implementation does employ priority queueing, such that the processes, if required to stop, are started in the order in which they were stopped; and it employs tokens which are labelled by a sequence number and discarded after use. Furthermore, the actual usage of the tokens differs: in Dijkstra's semaphore, posession of the token indicates the right to proceed; whereas with eventcounts and sequencers, the token merely provides an associated integer value which the process must wait for the eventcount to reach. This latter is fundamental to the distinction between the two kinds of semaphore.

Except for the ordering imposed on processes by the latter version of **P** and **V**, it might nevertheless be argued that the two are equivalent, were it not that a third semaphore operation, the **conditional P**, is possible on Dijkstra's semaphore which is not possible without great difficulty on the eventcount/sequencer pair. The conditional **P** may defined as follows, when used with Dijkstra's semaphore.

```
CP($S$):
        t := g(S.tokens);
        if t=$\phi$ return FALSE;
        else return TRUE;
```

The conditional **P** is equivalent to Dijkstra's **P**, except that it does not wait if no token is available. It is useful in deadlock avoidance; see, for example, BACH84 for an example of its use in a large-scale operating system.

In the eventcount/sequencer implementation, the "token" used to arbitrate access is inseparably joined with the sequencing mechanism built in to the **ticket** primitive. This mechanism causes the priority used in the call to the above **stop** and **start** primitives to be inseparably joined with the tokens used to arbitrate the right to proceed.

Consequently, once the **ticket** is obtained in the process of determining the right to proceed, it is impossible to simply return the ticket to the sequencer; another process may have performed another **ticket** operation in the interim. On the other hand, it cannot be discarded, since it is the responsibility of the process taking a ticket to advance the eventcount past the value on the ticket, to allow the next process to proceed.

Such problems do not exist for **CP** when used with Dijkstra's semaphore, since the nature of the anonymous tokens is such that there is a limited supply, and their absence indicates the need to wait, as compared to the infinite supply of tickets, whose values in relation to an associated eventcount indicate the need to wait (and impose an ordering on the waiting immediately upon taking a ticket).

The distinction between the two definitions of **P** and **V**, while subtle, leads to the above difficulty in backing out of a series of P operations in order to avoid a deadlock. While the authors do provide a simultaneous **P** operation, it implies use of the strategy of deadlock avoidance by locking all resources prior to using any of them. The relative merits of various deadlock avoidance strategies are an issue beyond the scope of this paper; yet it should be borne in mind that strategies other than that of locking all required resources before using any of them may be considered necessary in certain applications.

### Conclusion

The above decomposition demonstrates that fundamental differences exist between Dijkstra's P and V, and Reed and Kanodia's implementation of the same primitives. It further suggests that the sequencer primitives differ from those required to implement P and V in that they combine two underlying primitives: the token primitive presented above, and a sequencing primitive. As such, the P and V presented in REED79 could more accurately be considered sequencing semaphores, since they impose a sequencing on the waiting processes. They also make implementation of the conditional P operation less straightforward, due to the inability to put back a sequenced token without using it.

### *References*

BACH84. M. J. Bach and S. J. Buroff, "Multiprocessor *UNIX* Operating Systems," *AT&T Bell Laboratories Technical Journal* Vol. **63**(8), pp. 1733-1749 (October 1984).

DIJK68. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).

REED79. D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," *CACM* Vol. **22**(2), pp. 115-123 (February 1979).

ROSK84. J. E. Roskos, *Interprocess Communication and Synchronization via Commonly Addressable Memory with Data Movement Primitives*, Ph.D. Dissertation, Vanderbilt University, 1984.

# Operating System Kernel for a Reconfigurable Multiprocessor System*

**Geoffrey Brown and Chuan-lin Wu**
**Department of Electrical and Computer Engineering**
**The University of Texas at Austin**
**Austin, Texas 78712**

**ABSTRACT-- The design of an operating system kernel for Star is presented. Star is a reconfigurable multiprocessor system based upon a multistage interconnection network. The kernel is intended to serve as a basis for further operating system research. The kernel provides mechanisms for process management, local resource allocation, and inter-process communication.**

**A key design goal of the kernel was to minimize loss of transparency in the underlying hardware. This led to the development of a novel communication system to support the broadcast and merging capabilities of the interconnection network. The communication system presented provides for asynchronous multicast communication between processes which simplifies the tasks of creating distributed applications and of synchronizing cooperating processes.**

**The current status of our system is that we have a four node machine running and have implemented the kernel in MODULA-2.**

## 1 Introduction

In this paper we present the design of the Star operating system kernel. This kernel is replicated at each processor node of Star, a reconfigurable multiprocessor system [1]. The kernel represents a common core of software which will be used to investigate the problems of resource management and processor scheduling. The kernel provides the low level primitives to interface the operating system with the hardware and it provides the primitives for process management, interprocess communication (including inter-node communication) and memory management.

This work is motivated by a desire to develop an operating system which can fully exploit the inherent power of reconfigurable computer systems. Achieving this goal will require further research in the areas of resource management and scheduling. Our kernel design is a step towards this goal and has the following important characteristics:

- Loss of transparency in the hardware is minimized; by loss of transparency we mean any operation that is feasible in the underlying hardware but is not feasible using the primitives provided by the kernel.

- All decisions concerning resource management, program models, and scheduling are left to higher levels of the operating system.

- The kernel provides a convenient basis upon which a complete operating system can be built and is designed to simplify this task.

There is a substantial body of previous work in multiprocessor operating systems including: Hydra[2], Medusa[3], Eden[4], Micros [5] and many others. The architectures which these operating systems support vary widely. Star is different from most machines because of its reconfigurable topology; however, there are other reconfigurable systems including: TRAC [6], Cedar [7], PASM [8], and RP3 [9]. All of these reconfigurable systems feature shared memory while Star does not.

A common trend in multiprocessor operating systems is to design the operating system to hide the underlying communication system. Designs like HPC [10] , Eden, and Micros ignore topological issues altogether. In the Star system we are trying to address the topology issue by tailoring our operating system to obtain maximum performance given the very real constraints of a communication network.

Our work was influenced by a desire to ultimately provide the kind of support for distributed programs that Eden and HPC provide; however, while these projects were principally concerned with programability, we are principally concerned with developing a system which will efficiently utilize the interconnection network.

Because many of the design decisions of the overall operating system are topics for future investigation, it was essential that the kernel design be flexible, simple and easily modified. The kernel has been written entirely in MODULA-2 [11]. MODULA-2 is a convenient language for operating system design because it provides the mechanisms for modular design (Modules), the convenience of a higher level language, and reasonable access to the system hardware. MODULA-2 separates the definition of software from the implementation. The implementation of a software module can be modified at will without affecting other parts of the design. By implementing the kernel software in MODULA-2 we have been able to quickly prototype the system with confidence that the design can easily be refined.

Our experimental system has a limited number of processors; however, we are interested in developing application software which could be used on machines with

*This work is supported in part by a grant from IBM corporation.

0190-3918/86/0000/0234 $01.00 © 1986 IEEE

large numbers of processors. One way of facilitating this goal is to assume that application software is structured as a number of communicating processes. Because there may not be enough processors available to allocate a separate processor to each process of an application, the kernel supports the scheduling of multiple processes at a single processor. In addition, we believe that the writer of an application program should not have to be concerned with how the processes of that application might be scheduled for execution.

Because the processes of an application program might or might not share processors, our kernel provides the application programmer with a global communication model which, to communicating processes, appears the same whether the processes are at the same or different nodes.

The Star kernel is level structured and consists of three basic levels [12]:

| | |
|---|---|
| LEVEL 1 | Process Manager |
| LEVEL 2 | Memory Manager |
| LEVEL 3 | Global Communication |

The process manager provides support for multiple processes and the primitives necessary for sharing of local resources among processes which are scheduled at that node. The memory manager controls an important shared resource, local memory, and uses the primitives of the process manager. The global communication layer provides the mechanisms necessary for communication among processes which are not scheduled at the same node. In order to provide a consistent view of the system to application programs, the global communication layer also permits communication among locally scheduled processes.

In this paper we will examine levels 1 and 3. Level 2, the memory manager, provides facilities for managing the storage of a processor node and is not novel. The design decisions which led to this level decomposition are considered in later sections of the paper. In designing the kernel we followed one basic principle; design decisions were delayed as long as was feasible; each layer provides only a simple, basic set of operations while more specific operations were postponed, wherever possible, to higher levels of the operating system.

The organization is as follows. In Section 2 we present a description of the target system, called Star. In Section 3 we present the process manager and in Section 4 the global communication system. In section 5 we discuss building a distributed operating system on top of the kernel and consider the ways in which our design results in loss of transparency.

## 2 Target System

Star , as illustrated in Figure 1, is composed of a collection of N processor elements (PE) and a communication network, called Starnet. N is generally a power of 2. Starnet can be composed of multiple baseline networks [13]. The prototype system, which is currently running, has one baseline network and four processor nodes.

The baseline network, as illustrated in Figure 2, is a multistage interconnection network which has the characteristic of being reconfigurable under the control of the processor nodes and can form such topologies as rings, trees, and meshes [14]. The baseline network does have the property of blocking; that is, the creation of a connection prevents some set of connections that would otherwise be feasible. Therefore, the management of interconnection resources is an important consideration in designing an operating system. Connections in the baseline network originate on the left (in Figure 2) and may connect several destination nodes on the right (called multicasting). In addition the network provides the capability of merging connection requests from the left to a common destination on the right.

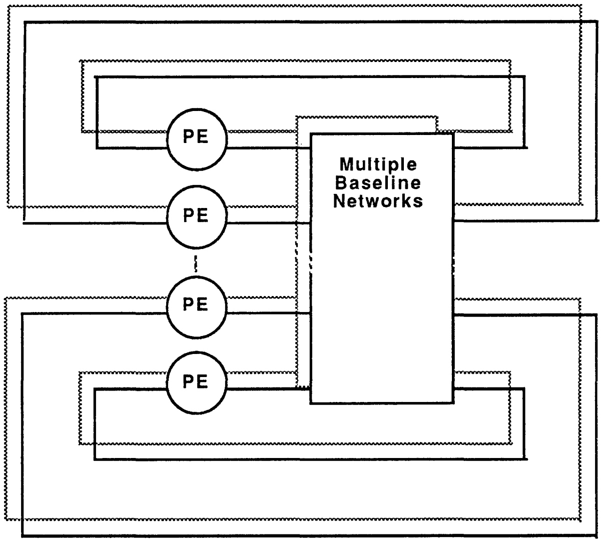


**FIGURE 1 Star System**

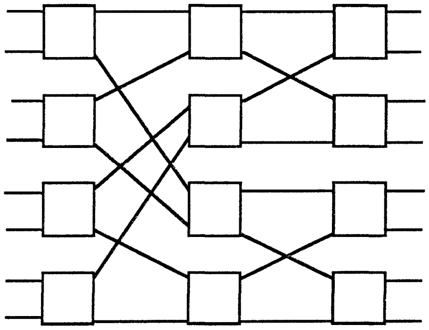


**FIGURE 2 An 8x8 Baseline Network**

Processor nodes are connected to the interconnection network by hardware interface units which have limited buffering capacity and an additional processor to handle communications. The hardware interface is illustrated in Figure 3. The communication processor interfaces to the interconnection network and the main processor exchanges data with the communication processor through an area of shared memory.

The essential characteristic of Starnet is that it is reconfigurable and may, at any moment, be partitioned into a number of private buses connecting two or more processor nodes. We will refer to a group of nodes connected by such a bus as a multicast group.

The details of how the network is reconfigured are immaterial to this paper. Assuming such a multicast group exists, there is, at any moment, exactly one node which is the sender (has the right to send data over the bus). All other nodes are receivers. Receivers may, through hardware control, stop the flow of data and request the right to become the sender. The sender may send data (unless blocked by one of the receivers) or it may relinquish the right to talk. In order for data transmission to occur, all the receivers connected to a bus must accept every word of data transmitted.

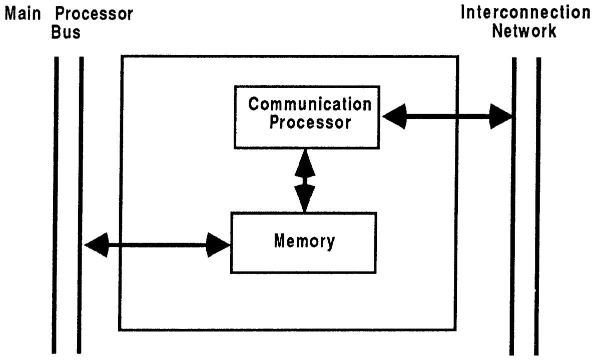


**FIGURE 3 Hardware Interface**

The hardware supports a mode of communication which we will refer to as *reliable multicasting*. This is because message broadcasting within a multicasting group is both error free and order preserving. Reliable multicast is an important mode of communication. Gehani has shown that multicasting simplifies the development of distributed applications [15]. In addition, multicasting greatly reduces the message traffic between processors needed for insuring mutual exclusion of access to critical sections. The distributed mutual exclusion problem was studied by Lamport [16] and Ricart [17]. Ricart showed that in a general message based system which supports broadcast, but not strict message sequencing, N messages (where N is the number of processors) are needed to invoke a critical section. With strict message sequencing only 2 messages are needed to invoke a critical region. Because of the efficiency which reliable multicasting provides in insuring mutual exclusion and the ease of use which Gehani has demonstrated, we concluded that the multicast capability of the hardware is a powerful feature which the kernel must support.

## 3 Process Manager

Support for the process abstraction is provided at the lowest level of the Star kernel by a process manager. The process manager provides mechanisms to create and destroy processes and low level synchronization and communication primitives. The primitives of the process manager may be invoked only by processes which are local to the node. Although these local primitives are not directly used by application programs, they are used in implementing the global communication system provided for application programs and distributed portions of the operating system.

The choice of primitive operations and the major design issues addressed in the process manager have a profound influence on the rest of the operating system. In Section 3.1 we discuss two important design issues: scheduling and memory management. The communication and synchronization primitives provided by the process manager are used in implementing the communication system provided by level 3 of the operating system. In Section 3.2 we discuss the functional characteristics of the process manager.

### 3.1 Design Considerations

Two issues which affected the process manager design involve scheduling and memory management. Although the process manager does not make scheduling decisions, the design had to be flexible enough to permit the implementation of a process scheduler at a later phase of the operating system design. The second issue is whether memory management should have been provided at a level below the process manager.

We chose not to implement the memory manager below the process manager, because we felt that it would be easier to implement using the primitives of the process manager. The process manager provides support for the objects *process* and *semaphore*. Creation and destruction of these two object types does involve the allocation and deallocation of memory. In addition the process manager must manipulate queues of process descriptors. However, in both cases the amounts of memory involved are small and are handled using statically allocated tables.

One of the major responsibilities of the process manager is to maintain a list (the Ready List) of processes which are ready to run. The discipline followed in queuing processes on the ready list and in transferring processor control to a ready process clearly has an impact on the processor scheduling algorithms which may be implemented. It is of great importance that this queuing discipline be sufficiently general to support a wide variety of scheduling policies. We defined the following characteristics of our process manager:

- The process manager supports priorities; operations are provided which permit the process priorities to be examined and modified. Processes are selected for execution based upon these priorities. The priority of the running process is at least as large as that of any ready process.

- The ready list is structured as a priority queue using a FIFO discipline to break ties.

- Although time slicing is not implemented in our kernel, this feature could be added to the process manager without modifying the interface to the process manager.

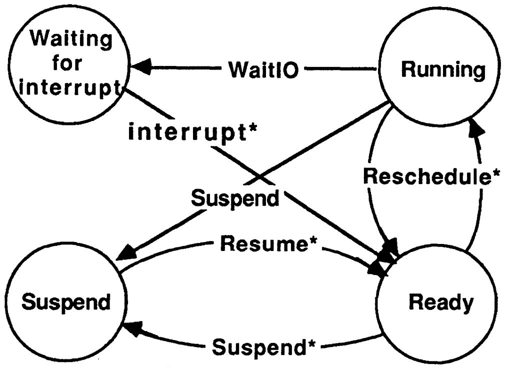
Ruschitzka and Fabry [18] have shown that given these characteristics a function for assigning priorities can be found so that the following scheduling algorithms can be implemented: FIFO, LIFO, preemptive shortest job first. With the addition of time slicing, round robin and feedback algorithms can be implemented. Thus process scheduling disciplines are not unduly restricted by the design of the process manager.

## 3.2 Facilities of Process Manager

The process manager provides support for processes and for communication between processes. The procedures provided by the process manager fall into four catagories: Process Control, Process Synchronization, Process Communication and Utility procedures. The process control procedures include:

| | |
|---|---|
| *CreateProcess* | Create a new process; |
| *DestroyProcess* | Destroy a process; |
| *Suspend* | Prevent a process from executing; |
| *Resume* | Resume a suspended process; |
| *GetPriority* | Get the priority of a process; |
| *SetPriority* | Set the priority of a process; |
| *CurrentProcess* | Get the identity of the current process; and |
| *WaitIO* | Allows a process to wait on a particular hardware interrupt vector. When the interrupt occurs, the process is placed on the ready queue. This permits the structuring of device drivers as separate processes. |

The control procedures lead to the process state diagram shown in Figure 4. A running process may suspend itself, wait on an interrupt, or lose control of the processor through a rescheduling operation. Rescheduling occurs whenever a process with a priority higher than the running process becomes ready to run.



*EXTERNAL EVENTS

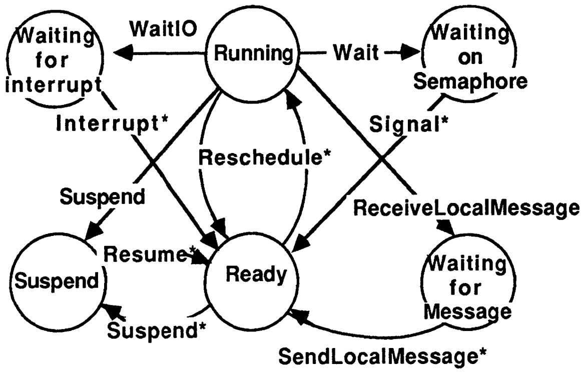
FIGURE 4 Basic Process States

In addition to the process control procedures, the process manager provides abstract data types and procedures for process synchronization and communication.

The synchronization and communication primitives of the process manager were chosen to provide a convenient basis for higher levels of the operating system. In the interest of flexibility, the process manager provides both semaphores and a primitive form of messages.

Semaphores have been widely discussed in the literature [19]; we will say nothing further about them except to list the operations provided by the process manager : *CreateSemaphore, DestroySemaphore, Wait, Signal.*

There are many different message systems and the design possibilities included blocking or non-blocking send, blocking or non-blocking receive, and buffering mechanisms. Because a fully synchronized message system may easily be implemented using semaphores, we concluded that a less tightly synchronized message system was appropriate at this level. Because there is no memory management facility available to the process manager, buffering had to be strictly limited. Process manager level messages (*LocalMessage*) are not intended to be used for transferring large amounts of data, but rather to provide what amounts to a private event flag with state information. In order to provide this capability at a minimum cost, *LocalMessage*s were restricted to positive integers and the following *LocalMessage* primitives provided: *SendLocalMessage* (non-blocking), *ReceiveLocalMessage* (blocking), *LocalMessageFlush* (non-blocking receive). Because buffer space is limited (to a single *LocalMessage*) and *SendLocalMessage* is non-blocking, overflow *LocalMessages* are simply discarded. The process synchronization and communication primitives lead to the process state diagram shown in Figure 5. In addition to the state transitions shown in Figure 4, a running process may also lose control of the processor if it waits on a semaphore or waits for a local message.



*EXTERNAL EVENTS

FIGURE 5 Complete Process State Diagram

A significant issue in the design of the process manager was the question of process termination. In general, the destruction of processes may affect the higher levels of the operating system. This is because higher levels of the operating system may associate further object types with processes and may need to perform some bookkeeping when a process is destroyed. For this reason, the process manager provides a general facility for installing termination procedures. The process manager maintains a list of termination procedures, each of which is called when a process is destroyed. These procedures enable higher levels of the operating system to be notified at process destruction time so that they may perform any necessary bookkeeping operations. This feature is both powerful and dangerous; however, some protection is provided by dictating that only the initial process of the node may install these termination procedures.

## 4 Global Communication Layer

As mentioned in Section 3, the process manager provides some local communication primitives. These primitives are provided to facilitate the implementation of higher levels of the kernel. In a distributed system such as Star a more general interprocess communication system is needed which can cross processor boundaries. In order to provide a coherent view of the system to application programs, the global communication system may also be used for local communications. The global communication system of the kernel is a message passing system supporting reliable multicasting. The design criteria for the low level communication system were:

- To support hardware communication modes;
- To support location independence of processes. The user of the system should not have to modify his code depending upon whether or not two communicating processes are to be scheduled at the same node; and
- To keep the primitive system both simple and efficient.

Because this is the first level of software above the underlying communication hardware, it was essential that it closely adhere to a communication model which the underlying hardware can efficiently support. In particular, we were concerned that the communication software fully support the multicasting capability of the hardware. The communication system provides for location independence of processes by providing a unified model of communication between locally executing and distributed processes.

### 4.1 Global Communication System Definition

The global communication system for Star supports the multicasting capability provided by multicast and merge in the hardware. The communication system is based upon the following abstractions. All communication occurs asynchronously between privately owned *ports* . Two *ports* may communicate if they are attached to the same *channel. Channels* are multiway, multidirectional communication paths. An arbitrary number of *ports* may be connected to a single *channel.* Communication between two *ports* occurs when a process writes a message to one of its *ports* and a receive operation is performed by a process on a *port* connected to the same *channel.* The *channel* model was chosen to closely match the multicasting buses which are supported by the interconnection network.

All messages sent by a port are received, in the order sent, by all ports connected to the same channel. There are two types of channels, local and distributed. A local channel may connect ports which reside on a single processor node. Distributed channels may connect ports on different nodes. Distributed channels exist only where hardware connections between nodes exist. More specifically, distributed channels are realized by a particular configuration of the underlying interconnection network.

### 4.2 Ports and Channels

In this section we will consider what ports and channels are. There is a single interface for both local and distributed channels; however, the implementation of the two is different. When discussing implementation details we mean, for the moment, local channel. We will consider the implementation of distributed channels in Section 4.4.

A channel is essentially a shared data structure consisting of a record for each connected port (the PortList), a fixed buffer space, and a pointer into the buffer space indicating where the next message should be deposited. The set of operations which can be performed on a channel is:

| | |
|---|---|
| *CreateChannel* - | Create a new channel. With a specified name and buffer size. |
| *DestroyChannel*- | A channel with no attached ports may be destroyed. |
| *ConnectPort* - | Connect the specified port to the channel. Does not block if the named channel does not exist. |
| *DisconnectPort* -- | Disconnect the specified port from the channel. |

Ports may be connected to or disconnected from channels at any time. In theory, an arbitrary number of ports may be connected to a channel (in practice this is limited by our implementation). Monitoring of communication can be achieved without introducing delay by attaching a monitor port to the channel.

A port is a privately owned object. Only the owner of a port may read data from or write data to a port; however, a port may be connected to or disconnected from a channel by a process other than its owner. The communication system supports the notion of a process waiting to receive data from any subset of its ports. This is implemented using LocalMessages. Any port which is connected to a channel may enable or disable a notification mode. If the notification mode of the port is enabled, a LocalMessage is sent to the owner of the port whenever a message is added to the buffer of the channel to which the port is connected.

The following operations may be performed upon a port.

| | |
|---|---|
| *ReadFromPort---* | Read next message. This is non-blocking and returns NIL if no message is available. |
| *SendToPort--* | Send a message through a port. If the port is not connected to the channel, the message will be lost. |
| *EnableNotification--* | When the notification mode of a port is enabled, a LocalMessage is sent to the owner of the port whenever a message is added to the buffer of the channel to which the port is attached. |
| *DisableNotification--* | Disable notification mode. |

The ports connected to a channel may read the messages in the channel buffer at their own pace. We may view each port as having a private pointer into a shared list of messages. This private pointer is maintained as part of the channel data structure and is directly accessed only by the channel software. In Figure 6 the channel buffer model and the function of these private pointers is illustrated. In this figure, three ports are connected to the channel. Each port is represented by a private pointer into the channel buffer.

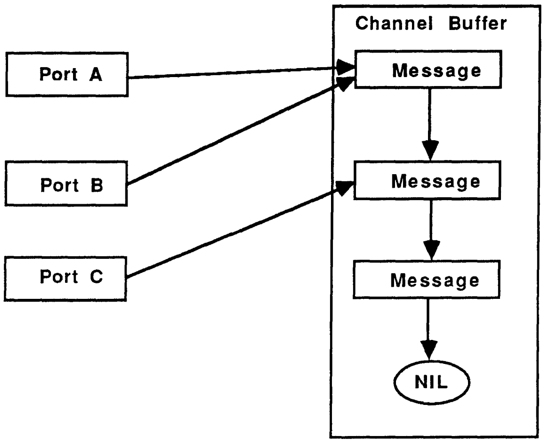


**FIGURE 6 Channel Buffer**

Because buffer space for a given channel is fixed, we can view the buffer as being circular with new messages eventually overwriting old ones. Although this event is not an error it is important to know when it has occurred. For this purpose a *tail pointer* is maintained by the channel, which indicates the oldest message which has not been read by all attached ports. The channel data structure also consists of some status flags including a *note flag* indicating whether any of the attached ports have enable their notification mode, a *nil flag*, indicating whether any of the attached ports have read the most recent message.

The PortList of a channel consists of a record for each connected port. This record includes the port name, a pointer the the next message to be read, a *notificationmode flag*, a *nil flag*, and an *overflow flag*. The *notificationmode flag* indicates whether the port should be notified when a new message is written, the *overflow flag* indicates whether a message has been lost since the last read, the *nil flag* indicates whether there are any messages to read.

## 4.3 Buffering

The choice of asynchronous communication implied that some buffering capability must be built into the communication system. Because buffering is, of necessity, limited, we had to either institute flow control or accept the potential loss of data due to buffer overflow. We regarded the introduction of flow control at this level as a poor idea because it introduces additional overhead for all communication when some applications may not need it. This is particularly true if the communication protocol of an application is inherently bounded (can have a finite limit on the number of outstanding messages) and the bound is sufficiently small that overflow is impossible. Flow control may be easily introduced at a higher level.

The question remained -- how much buffer capacity should be provided. Reid [20] solved this by providing a single message buffer. Each successive message overwrites the previous message. This approach has a serious flaw in that it is impossible to use broadcast to provide synchronization because we cannot be certain that a reply message did not overwrite the original message before all ports had read it. Any fixed buffering scheme has the limitation that it renders some communication protocols either impossible or difficult and provides excess capacity for others.

In solving this problem we made the following assumption: a particular channel is shared by ports which use the same communication protocol. The protocol used must either be bounded or must handle potential message loss. In either case it is possible to allocate a fixed buffer space when the channel is first created. The buffer size is specified in the initial allocation request. Obviously it is possible that this request may not be met; however, this event will be discovered at the outset rather than at some point well into the execution of an application.

## 4.4 Implementing Distributed Channels

We have delayed discussing the implementation of distributed channels until now because their implementation is significantly different from local channels, although they respond to exactly the same set of operations. Distributed channels are realized through the coordinated efforts of two or more connected processor nodes.

In the local channel model we can view a channel as a shared data structure with a number of attached ports. Each port is uniquely owned by a process. Unfortunately this model is much more complicated in the distributed case. First because the distributed channel cannot be implemented through shared memory and second because the communicating ports are on different nodes.

In order to realize the channel model in the distributed case we maintain, at each connected node, a copy of the message buffer. The other parts of the channel data structure are provided for maintaining port status and are only maintained locally. Thus each node which is connected maintains separate data for the channel; part of which is replicated and part of which is unique.

Recall that the multicasting buses realized in the interconnection network have the capability of controlling data flow across the interconnection network and there is always a unique sender on the bus. The capability of controlling data flow insures that a receiving node can process messages as they are received. The fact that there is a single sender insures that only one node will be able to transmit messages at any moment. Updates to a copy of the channel buffer occur in only two ways: the node is the current sender and a send is performed by one of its connected ports or the node is a receiver and a message is received in the buffer of the hardware interface.

In practice multiple distributed channels can be supported over each multicasting bus. The multiple distributed channels share the bandwidth provided by the multicasting bus.

## 5 Discussion

In this section we discuss how the kernel can be used to build a complete operating system, how the design avoids loss of transparency, and the current status of the system

### 5.1 Building a Complete Operating System

The kernel is not an operating system; however, it provides the facilities needed to build a distributed operating system. A distributed operating system for Star would consist of a copy of the kernel and some statically defined set of initial processes at each processor of the system. It would need a file system, some user interface, and a resource manager. An early version of a resource manager for Star has been described in [21].

When the system is initialized, the hardware is not configured and there are no communication channels. Therefore, it must be possible for the set of statically defined processes to create a set of communication channels so that they may cooperate to perform the operating system functions. The feasibility of this initialization step is illustrated by a simple example. Imagine that there is a process **x** residing at node **X** and a process **y** residing at node **Y**. The following code fragments demonstrate a method in which the two processes can become connected to a common channel.

```
(* code fragment for x *)

CONST    buffersize = ?;
VAR      myport : port; success : BOOLEAN;

BEGIN
   CreateChannel("channelone",{X,Y} ,buffersize);
   success := ConnectPort(myport, "channelone");
   ........


(* code fragment for y *)

VAR      myport : port; success : BOOLEAN;

BEGIN
   REPEAT
      success := ConnectPort(myport, "channelone") ;
   UNTIL success
   ........
```

Additional protocol steps involving message passing between **x** and **y** are needed so that the two processes may reach an agreed upon state. Although this initialization process may seem to be a great deal of trouble, it is only necessary for initializing the higher levels of the operating system and should not be necessary for application programs.

### 5.2 Loss of Transparency

In Section 1 we mentioned that minimization of loss of transparency was a key design goal. We have achieved this by tailoring our global communication system to the capabilities of the interconnection network. The channel model supports the multicast connections among multiple processors. In addition the channel maintains the strict message sequencing provided by the underlying hardware.

We have introduced some loss of transparency by forcing channel buffers to be fixed at creation; this introduces the possibility of message loss which does not exist in the hardware; however, we have argued that with a proper choice of communication protocols message loss can be avoided.

### 5.3 Concluding Remarks

In this paper we have presented the operating system kernel for Star, a reconfigurable multiprocessor system. The kernel is intended to serve as a basis for further operating systems research. It provides process management, local resource allocation, and inter-process communication mechanisms. The design is simple and avoids loss of transparency in the hardware by supporting the communication modes provided by the interconnection network.

The kernel supports multiple processes and, at the process manager level, provides simple, yet powerful inter-process communication mechanisms. In addition the process manager provides for the implementation of a wide range of process scheduling algorithms.

The global communication mechanism was designed to closely support the communication modes provided for by the underlying hardware. It supports the general notion of merging by allowing additional communicating entities (ports) to join into a conversation (a channel) at will. In addition the communication mechanism maintains the capability of the hardware to provide reliable multicasting of messages.

The present status of our system is that we have a four node machine running. We have implemented and tested the process manager, memory manager and local portion of the distributed communication system. In addition we have implemented a linking loader which allows separately compiled MODULA-2 programs to be loaded, linked to the operating system and executed as independent processes. The implemented code constitutes about two thousand lines of MODULA-2. We plan to extend our work to provide greater support for distributed programs.

## References

[1] C. Wu, et al, "Prototype of Star Architecture- A Status Report," Proceedings of NCC 1985 , July 1985, pp. 191-201.

[2] W. Wulf, E. Cohen, W. Corwin, et. al, "HYDRA : The Kernel of a Multiprocessing Operating System," CACM, vol. 17, no. 6, June 1974, pp. 337-345.

[3] J. Ousterhout, D. Scelza and P. Sindhu, "Medusa: An Experiment in Distributed System Structure," CACM, vol. 23, no. 2, Feb. 1980, pp. 92-105.

[4] G. T. Almes, A. P. Black and E. D. Lazowska, "The Eden System: A Technical Review," IEEE Transactions on Software Engineering, vol. SE-11, no. 1, Jan. 1985, pp.43-58.

[5] L. Wittie, and A. van Tilborg, "MICROS: A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," IEEE Transactions on Computers, vol. C-29, no. 12, Dec. 1980, pp. 1135-1144.

[6] J. C. Browne, "TRAC: An Environment for Parallel Computing," COMPCON Spring 1984 , pp. 294-298.

[7] D. Gajski, D. Kuck, L. Duncan and A. Sameh, "Cedar -- A Large Scale Multiprocessor," 1983 International Conference on Parallel Processing, pp. 524-529.

[8] H. Siegel, et al, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," IEEE Transactions on Computing, vol. C-30, no. 12, Dec. 1981, pp. 934-946.

[9] G. F. Pfister, W. C. Bentley, et al, "The IBM Research Processor Prototype (RP3): Introduction and Architecture," 1985 International Conference on Parallel Processing, pp. 764-771.

[10] T. J. LeBlanc. and S. A. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," 5th International Conference on Distributed Computing, 1985, pp. 26-34.

[11] N. Wirth, Programming in Modula 2, 2nd edition, Springer-Verlag 1983.

[12] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," CACM, vol. 11, no. 5, May 1968, pp. 341-346.

[13] C. Wu, and T. Feng, "On a Class of Multistage Interconnection Networks, " IEEE Transactions on Computers, vol. c-29, Aug. 1980, pp. 694-702.

[14] W. Lin and C. Wu, "Design of Configuration Algorithms of Commonly Used Topologies for a Multiprocessor--Star," Proceedings of the 1985 International Conference on Parallel Processing, August 1985, pp. 734-741.

[15] N. H. Gehani, "Broadcasting Sequential Processes (BSP)," IEEE Transactions on Software Engineering, vol. se-10, no. 4, July 1984, pp. 343-351.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, vol. 21, no. 7, July 1978, pp. 558-565.

[17] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," CACM, vol. 24, no. 1, Jan. 1981 pp. 9-17.

[18] M. Ruschitzka and R. S. Fabry, "A Unifying Approach to Scheduling," CACM, vol. 20, no. 7, July 1977, pp. 469-477.

[19] J. Peterson and A. Silbershatz, Operating Systems Concepts, Addison-Wesley, 1983.

[20] L. G. Reid, Control and Communication in Programs, UMI Research Press, 1982.

[21] W. Lin and C. Wu, "An Object-based Resource Management Mechanism for a High-Performance Distributed Computing System--Star," Proceedings of the 1984 Computer Software and Application Conference, Nov. 1985, pp. 208-216.

# THE INTERFACE BETWEEN DISTRIBUTED OPERATING SYSTEM AND HIGH-LEVEL PROGRAMMING LANGUAGE

Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY 14627

**Abstract** — A distributed operating system provides a process abstraction and primitives for communication between processes. A distributed programming language regularizes the use of the primitives, making them both safer and more convenient. The level of abstraction of the primitives, and therefore the division of labor between the operating system and the language support routines, has serious ramifications for efficiency and flexibility. Experience with three implementations of the LYNX distributed programming language suggests that functions that can be implemented on either side of the interface are best left to the language run-time package.

## Introduction

Recent years have seen the development of a large number of distributed programming languages and an equally large number of distributed operating systems. While there are exceptions to the rule, it is generally true that individual research groups have focused on a single language, a single operating system, or a single language/O.S. pair. Relatively little attention has been devoted to the relationship between languages and O.S. kernels in a distributed setting.

Amoeba [15], Demos-MP [16], Locus [26], and the V kernel [7] are among the better-known distributed operating systems. Each by-passes language issues by relying on a simple library-routine interface to kernel communication primitives. Eden [5] and Cedar [24] have both devoted a considerable amount of attention to programming language issues, but each is very much a single-language system. The Accent project at CMU [17] is perhaps the only well-known effort to support more than one programming language on a single underlying kernel. Even so, Accent is only able to achieve its multi-lingual character by insisting on a single, universal model of interprocess communication based on remote procedure calls [11]. Languages with other models of process interaction are not considered.

In the language community, it is unusual to find implementations of the same distributed programming language for more than one operating system, or indeed for any *existing* operating system. Dedicated, special-purpose kernels are under construction for Argus [14], SR [1,2], and NIL [22,23]. Several dedicated implementations have been designed for Linda [6,10]. No distributed implementations have yet appeared for Ada [25].

If parallel or distributed hardware is to be used for *general-purpose* computing, we must eventually learn how to support multiple languages efficiently on a single operating system. Toward that end, it is worth considering the division of labor between the language run-time package and the underlying kernel. Which functions belong on which side of the interface? What is the appropriate level of abstraction for universal primitives? Answers to these questions will depend in large part on experience with a variety of language/O. S. pairs.

This paper reports on implementations of the LYNX distributed programming language for three existing, but radically different, distributed operating systems. To the surprise of the implementors, the implementation effort turned out to be substantially easier for kernels with low-level primitives. If confirmed by similar results with other languages, the lessons provided by work on LYNX should be of considerable value in the design of future systems.

The first implementation of LYNX was constructed during 1983 and 1984 at the University of Wisconsin, where it runs under the Charlotte distributed operating system [3,9] on the Crystal multicomputer [8]. The second implementation was designed, but never actually built, for Kepecs and Solomon's SODA [12,13]. A third implementation has recently been released at the University of Rochester, where it runs on BBN Butterfly multiprocessors [4] under the Chrysalis operating system.

Section 2 of this paper summarizes the features of LYNX that have an impact on the services needed from a distributed operating system kernel. Sections 3, 4, and 5 describe the three LYNX implementations, comparing them one to the other. The final section discusses possible lessons to be learned from the comparison.

## LYNX Overview

The LYNX programming language is not itself the subject of this article. Language features and their rationale are described in detail elsewhere [19,20,21]. For present purposes, it suffices to say that LYNX was designed to support the loosely-coupled style of programming encouraged by a distributed operating system. Unlike most existing languages, LYNX extends the advantages of high-level communication facilities to processes designed in isolation, and compiled and loaded at disparate times. LYNX supports interaction not only between the pieces of a multi-process application, but also between separate applications and between user programs and long-lived system servers.

Processes in LYNX execute in parallel, possibly on separate processors. There is no provision for shared memory. Interprocess communication uses a mechanism similar to remote procedure calls (RPC), on virtual circuits called **links**. Links are two-directional and have a single process at each end. Each process may be divided into an arbitrary number of threads of control, but the threads execute in mutual exclusion and may be managed by the language run-time package, much like the coroutines of Modula-2 [27].

### Communication Characteristics

(The following paragraphs describe the communication behavior of LYNX processes. The description does not provide much insight into the way that LYNX programmers think about their programs. The intent is to describe the externally-visible characteristics of a process that must be supported by kernel primities.)

At the University of Wisconsin, this work was supported in part by NSF grant number MCS-8105904, DARPA contract number N0014-82-C-2087, and a Bell Telephone Laboratories Doctoral Scholarship. At the University of Rochester, the work is supported in part by NSF grant number DCR-8320136 and DARPA contract number DACA76-85-C-0001.

0190-3918/86/0000/0242 $01.00 © 1986 IEEE

Messages in LYNX are not received asynchronously. They are queued instead, on a link-by-link basis. Each link end has one queue for incoming requests and another for incoming replies. Messages are received from a queue only when the queue is **open** and the process that owns its end has reached a well-defined **block point**. Request queues may be opened or closed under explicit process control. Reply queues are opened when a request has been sent and a reply is expected. The set of open queues may therefore vary from one block point to the next.

A blocked process waits until one of its previously-sent messages has been received, or until an incoming message is available in at least one of its open queues. In the latter case, the process chooses a non-empty queue, receives that queue's first message, and executes through to the next block point. For the sake of fairness, an implementation must guarantee that no queue is ignored forever.

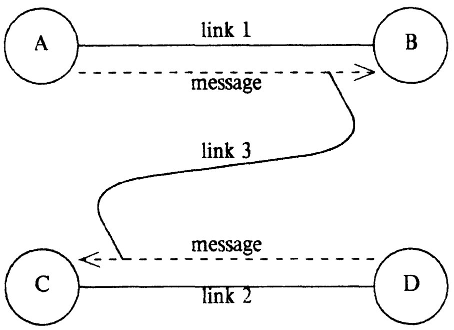


**figure 1: link moving at both ends**

Messages in the same queue are received in the order sent. Each message blocks the sending coroutine within the sending process. The process must be notified when messages are received in order to unblock appropriate coroutines. It is therefore possible for an implementation to rely upon a stop-and-wait protocol with no actual buffering of messages in transit. Request and reply queues can be implemented by lists of blocked coroutines in the run-time package for each sending process.

The most challenging feature of links, from an implementor's point of view, is the provision for *moving* their ends. Any message, request or reply, can contain references to an arbitrary number of link ends. Language semantics specify that receipt of such a message has the side effect of moving the specified ends from the sending process to the receiver. The process at the far end of each moved link must be oblivious to the move, even if it is currently relocating its end as well. In figure 1, for example, processes A and D are moving their ends of link 3, independently, in such a way that what used to connect A to D will now connect B to C.

It is best to think of a link as a flexible hose. A message put in one end will eventually be delivered to whatever process happens to be at the other end. The queues of available but un-received messages for each end are associated with the link itself, not with any process. A moved link may therefore (logically at least) have messages inside, waiting to be received at the moving end. In keeping with the comment above about stop-and-wait protocols, and to prevent complete anarchy, a process is not permitted to move a link on which it has *sent* unreceived messages, or on which it owes a reply for an already-received request.

## Kernel Requirements

To permit an implementation of LYNX, an operating system kernel must provide processes, communication primitives, and a naming mechanism that can be used to build links. The major questions for the designer are then 1) how are links to be represented? and 2) how are RPC-style request and reply messages to be transmitted on those links? It must be possible to move links without losing messages. In addition, the termination of a process must destroy all the links attached to that process. Any attempt to send or receive a message on a link that has been destroyed must fail in a way that can be reflected back into the user program as a run-time exception.

# The Charlotte Implementation

## Overview of Charlotte

Charlotte [3,9] runs on the Crystal multicomputer [8], a collection of 20 VAX 11/750 **node** machines connected by a 10-Mbit/second token ring from Proteon Corporation.

The Charlotte **kernel** is replicated on each node. It provides direct support for both **processes** and **links**. Charlotte links were the original motivation for the circuit abstraction in LYNX. As in the language, Charlotte links are two directional, with a single process at each end. As in the language, Charlotte links can be created, destroyed, and moved from one process to another. Charlotte even guarantees that process termination destroys all of the process's links. It was originally expected that the implementation of LYNX-style interprocess communication would be almost trivial. As described in the rest of this section, that expectation turned out to be naive.

Kernel calls in Charlotte include the following:

MakeLink (var end1, end2 : link)
    Create a link and return references to its ends.

Destroy (myend : link)
    Destroy the link with a given end.

Send (L : link; buffer : address; length : integer; enclosure : link)
    Start a **send activity** on a given link end, optionally enclosing one end of some other link.

Receive (L : link; buffer : address; length : integer)
    Start a **receive activity** on a given link end.

Cancel (L : link; d : direction)
    Attempt to cancel a previously-started send or receive activity.

Wait (var e : description)
    Wait for an activity to complete, and return its description (link end, direction, length, enclosure).

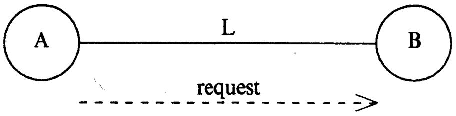
All calls return a status code. All but *Wait* are guaranteed to complete in a bounded amount of time. *Wait* blocks the caller until an activity completes.

The Charlotte kernel matches send and receive activities. It allows only one outstanding activity in each direction on a given end of a link. Completion must be reported by *Wait* before another similar activity can be started.

## Implementation of LYNX

The language run-time package represents every LYNX link with a Charlotte link. It uses the activities of the Charlotte kernel to simulate the request and reply queues described in section 2.1. It starts a send activity on a link whenever a process attempts to send a request or reply message. It starts a receive activity on a link when the corresponding request or reply queue is opened, if both were closed before. It attempts to cancel a previous-started receive activity when a process closes its request queue, if the reply queue is also closed. The multiplexing of request and reply queues onto receive activities was a major source of problems for the implementation effort. A second source of problems was the inability to enclose more than one link in a single Charlotte message.

**Screening Messages.** For the vast majority of remote operations, only two Charlotte messages are required: one for the request and one for the reply. Complications arise, however, in a number of special cases. Suppose that process **A** requests a remote operation on link **L**.



Process **B** receives the request and begins serving the operation. **A** now expects a reply on **L** and starts a receive activity with the kernel. Now suppose that before replying **B** requests another operation on **L**, in the reverse direction (the coroutine mechanism mentioned in section 2 makes such a scenario entirely plausible). **A** will receive **B**'s request before the reply it wanted. Since **A** may not be willing to serve requests on **L** at this point in time (its request queue is closed), **B** is not able to assume that its request is being served simply because **A** has received it.

A similar problem arises if **A** opens its request queue and then closes it again, before reaching a block point. In the interests of concurrency, the run-time support routines will have posted a *Receive* with the kernel as soon as the queue was opened. When the queue is closed, they will attempt to cancel the *Receive*. If **B** has requested an operation in the meantime, the *Cancel* will fail. The next time **A**'s run-time package calls *Wait*, it will obtain notification of the request from **B**, a message it does not want. Delaying the start of receive activities until a block point does not help. **A** must still start activities for *all* its open queues. It will continue execution after a message is received from exactly *one* of those queues. Before reaching the *next* block point, it may change the set of messages it is willing to receive.

It is tempting to let **A** buffer unwanted messages until it is again willing to receive from **B**, but such a solution is impossible for two reasons. First, the occurrence of exceptions in LYNX can require **A** to cancel an outstanding *Send* on **L**. If **B** has already received the message (inadvertently) and is buffering it internally, the *Cancel* cannot succeed. Second, the scenario in which **A** receives a request but wants a reply can be repeated an arbitrary number of times, and **A** cannot be expected to provide an arbitrary amount of buffer space.

**A** must return unwanted messages to **B**. In addition to the **request** and **reply** messages needed in simple situations, the implementation now requires a **retry** message. *Retry* is a negative acknowledgment. It can be used in the second scenario above, when **A** has closed its request queue after receiving an unwanted message. Since **A** will have no *Receive* outstanding, the re-sent message from **B** will be delayed by the kernel until the queue is re-opened.

In the first scenario, unfortunately, **A** will still have a *Receive* posted for the reply it wants from **B**. If **A** simply returned requests to **B** in retry messages, it might be subjected to an arbitrary number of retransmissions. To prevent these retransmissions we must introduce the **forbid** and **allow** messages. *Forbid* denies a process the right to send requests (it is still free to send replies). *Allow* restores that right. *Retry* is equivalent to *forbid* followed by *allow*. It can be considered an optimization for use in cases where no replies are expected, so retransmitted requests will be delayed by the kernel.

Both *forbid* and *retry* return any link end that was enclosed in the unwanted message. A process that has received a forbid message keeps a *Receive* posted on the link in hopes of receiving an allow message.[1] A process that has sent a forbid message remembers that it has done so and sends an allow message as soon as it is either willing to receive requests (its request queue is open) or has no *Receive* outstanding (so the kernel will delay all messages).

**Moving Multiple Links.** To move more than one link end with a single LYNX message, a request or reply must be broken into several Charlotte messages. The first packet contains non-link data, together with the first enclosure. Additional enclosures are passed in empty **enc** messages (see figure 2). For requests, the receiver must return an explicit **goahead** message after the first packet so the sender can tell that the request is wanted. No *goahead* is needed for requests with zero or one enclosures, and none is needed for replies, since a reply is always wanted.

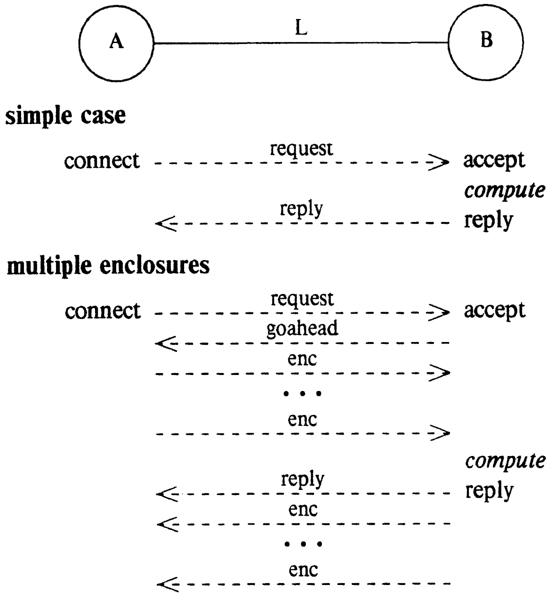


**figure 2: link enclosure protocol**

One consequence of packetizing LYNX messages is that links enclosed in unsuccessful messages may be lost. Consider the following chain of events:

a) Process **A** sends a request to process **B**, enclosing the end of a link.

b) **B** receives the request unintentionally; inspection of the code allows one to prove that only replies were wanted.

c) The sending coroutine in **A** feels an exception, aborting the request.

d) **B** crashes before it can send the enclosure back to **A** in a forbid message. From the point of view of language semantics, the message to **B** was never sent, yet the enclosure has been lost. Under such circumstances the Charlotte implementation cannot conform to the language reference manual.

The Charlotte implementation also disagrees with the language definition when a coroutine that is waiting for a reply message is aborted by a local exception. On the other end of the link

[1] This of course makes it vulnerable to receiving unwanted messages itself.

the server should feel an exception when *it* attempts to send a no-longer-wanted reply. Such exceptions are not provided under Charlotte because they would require a final, top-level acknowledgment for reply messages, increasing message traffic by 50%.

## Measurements

The language run-time package for Charlotte consists of just over 4000 lines of C and 200 lines of VAX assembler, compiling to about 21K of object code and data. Of this total, approximately 45% is devoted to the communication routines that interac' with the Charlotte kernel, including perhaps 5K for unwanted messages and multiple enclosures. Much of this space could be saved with a more appropriate kernel interface.

A simple remote operation (no enclosures) requires approximately 57 ms with no data transfer and about 65 ms with 1000 bytes of parameters in both directions. C programs that make the same series of kernel calls require 55 and 60 ms, respectively. In addition to being rather slow, the Charlotte kernel is highly sensitive to the ordering of kernel calls and to the interleaving of calls by independent processes. Performance figures should therefore be regarded as suggestive, not definitive. The difference in timings between LYNX and C programs is due to efforts on the part of the run-time package to gather and scatter parameters, block and unblock coroutines, establish default exception handlers, enforce flow control, perform type checking, update tables for enclosed links, and make sure the links are valid.

# The SODA Implementation

## Overview of SODA

As part of his Ph. D. research [12, 13], Jonathan Kepecs set out to design a minimal kernel for a multicomputer. His "Simplified Operating system for Distributed Applications" might better be described as a communications protocol for use on a broadcast medium with a very large number of heterogeneous nodes.

Each node on a SODA network consists of two processors: a **client processor**, and an associated **kernel processor**. The kernel processors are all alike. They are connected to the network and communicate with their client processors through shared memory and interrupts. Nodes are expected to be more numerous than processes, so client processors are not multi-programmed.

Every SODA process has a unique **id**. It also **advertises** a collection of **names** to which it is willing to respond. There is a kernel call to generate new names, unique over space and time. The **discover** kernel call uses unreliable broadcast in an attempt to find a process that has advertised a given name.

Processes do not necessarily *send* messages, rather they **request** the transfer of data. A process that is interested in communication specifies a name, a process id, a small amount of out-of-band information, the number of bytes it would like to send and the number it is willing to receive. Since either of the last two numbers can be zero, a process can request to send data, receive data, neither, or both. The four varieties of request are termed **put**, **get**, **signal**, and **exchange**, respectively.

Processes are informed of interesting events by means of software interrupts. Each process establishes a single **handler** which it can close temporarily when it needs to mask out interrupts. A process feels a software interrupt when its id and one of its advertised names are specified in a request from some other process. The handler is provided with the id of the requester and the arguments of the request, including the out-of-band information. The interrupted process is free to save the information for future reference.

At any time, a process can **accept** a request that was made of it at some time in the past. When it does so, the request is completed (data is transferred in both directions simultaneously), and the requester feels a software interrupt informing it of the completion and providing it with a small amount of out-of-band information from the accepter. Like the requester, the accepter specifies buffer sizes. The amount of data transferred in each direction is the smaller of the specified amounts.

Completion interrupts are queued when a handler is busy or closed. Requests are delayed; the requesting kernel retries periodically in an attempt to get through (the requesting user can proceed). If a process dies before accepting a request, the requester feels an interrupt that informs it of the crash.

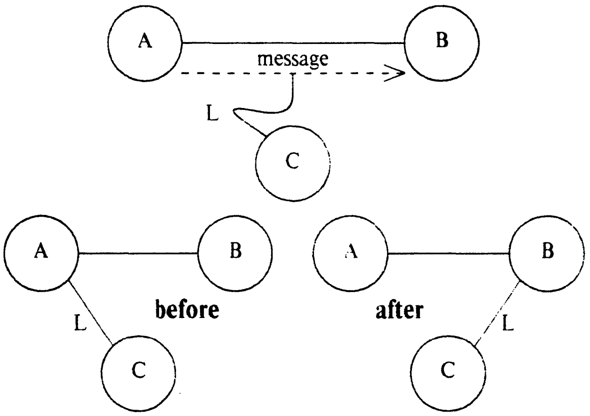
## A Different Approach to Links

A link in SODA can be represented by a pair of unique names, one for each end. A process that owns an end of a link advertises the associated name. Every process knows the names of the link ends it owns. Every process keeps a **hint** as to the current location of the far end of each of its links. The hints can be wrong, but are expected to work most of the time.

A process that wants to send a LYNX message, either a request or a reply, initiates a SODA *put* to the process it thinks is on the other end of the link. A process moves link ends by enclosing their names in a message. When the message is SODA-accepted by the receiver, the ends are understood to have moved. Processes on the fixed ends of moved links will have incorrect hints.

A process that wants to receive a LYNX message, either a request or a reply, initiates a SODA *signal* to the process it thinks is on the other end of the link. The purpose of the signal is allow the aspiring receiver to tell if its link is destroyed or if its chosen sender dies. In the latter case, the receiver will feel an interrupt informing it of the crash. In the former case, we require a process that destroys a link to accept any previously-posted status *signal* on its end, mentioning the destruction in the out-of-band information. We also require it to accept any outstanding *put* request, but with a zero-length buffer, and again mentioning the destruction in the out-of-band information. After clearing the *signals* and *puts*, the process can **unadvertise** the name of the end and forget that it ever existed.

Suppose now that process **A** has a link **L** to process **C** and that it sends its end to process **B**.



If **C** wants to send or receive on **L**, but **B** terminates after receiving **L** from **A**, then **C** must be informed of the termination so it knows that **L** has been destroyed. **C** will have had a SODA request posted with **A**. **A** must accept this request so that **C** knows to watch **B** instead. We therefore adopt the rule that a process that moves a link end must accept any previously-posted SODA request from the other end, just as it must when it destroys the link. It specifies a zero-length buffer and uses the out-of-band information to tell the other process where it moved its end. In the above example, **C** will re-start its request with **B** instead of **A**.

The amount of work involved in moving a link end is very small, since accepting a request does not even block the accepter. More than one link can be enclosed in the same message with no more difficulty than a single end. If the fixed end of a moving link is not in active use, there is no expense involved at all. In the above example, if C receives a SODA request from B, it will know that L has moved.

The only real problems occur when an end of a dormant link is moved. If our example, if L is first used by C after it is moved, C will make a SODA request of A, not B, since its hint is out-of-date. There must be a way to fix the hint. If each process keeps a cache of links it has known about recently, and keeps the names of those links advertised, then A may remember it sent L to B, and can tell C where it went. If A has forgotten, C can use the *discover* command in an attempt to find a process that knows about the far end of L.

A process that is unable to find the far end of a link must assume it has been destroyed. If L exists, the heuristics of caching and broadcast should suffice to find it in the vast majority of cases. If the failure rate is comparable to that of other "acceptable" errors, such as garbled messages with "valid" checksums, then the heuristics may indeed be all we *ever* need.

Without an actual implementation to measure, and without reasonable assumptions about the reliability of SODA broadcasts, it is impossible to predict the success rate of the heuristics. The SODA *discover* primitive might be especially strained by node crashes, since they would tend to precipitate a large number of broadcast searches for lost links. If the heuristics failed too often, a fall-back mechanism would be needed.

Several absolute algorithms can be devised for finding missing links. Perhaps the simplest looks like this:

- Every process advertises a **freeze** name. When C discovers its hint for L is bad, it posts a SODA request on the freeze name of every process currently in existence (SODA makes it easy to guess their ids). It includes the name of L in the request.
- Each process accepts a freeze request immediately, ceases execution of everying but its own searches (if any), increments a counter, and posts an **unfreeze** request with C. If it has a hint for L, it includes that hint in the freeze accept or the unfreeze request.
- When C obtains a new hint or has unsuccessfully queried everyone, it accepts the unfreeze requests. When a frozen process feels an interrupt indicating that its unfreeze request has been accepted or that C has crashed, it decrements its counter. If the counter hits zero, it continues execution. The existence of the counter permits multiple concurrent searches.

This algorithm has the considerable disadvantage of bringing every LYNX process in existence to a temporary halt. On the other hand, it is simple, and should only be needed when a node crashes or a destroyed link goes unused for so long that everyone has forgotten about it.

**Potential Problems.** As mentioned in the introduction, the SODA version of LYNX was designed on paper only. An actual implementation would need to address a number of potential problems. To begin with, SODA places a small, but unspecified, limit on the size of the out-of-band information for *request* and *accept*. If all the self-descriptive information included in messages under Charlotte were to be provided out-of-band, a minimum of about 48 bits would be needed. With fewer bits available, some information would have to be included in the messages themselves, as in Charlotte.

A second potential problem with SODA involves another unspecified constant: the permissible number of outstanding requests between a given pair of processes. The implementation described in the previous section would work easily if the limit were large enough to accommodate three requests for every link between the processes (a LYNX-request *put*, a LYNX-reply *put*, and a status *signal*). Since reply messages are always wanted (or can at least be discarded if unwanted), the implementation could make do with two outstanding requests per link and a single extra for replies. Too small a limit on outstanding requests would leave the possibility of deadlock when many links connect the same pair of processes. In practice, a limit of a half a dozen or so is unlikely to be exceeded (it implies an improbable concentration of simultaneously-active resources in a single process), but there is no way to reflect the limit to the user in a semantically-meaningful way. Correctness would start to depend on global characteristics of the process-interconnection graph.

### Predicted Measurements

Space requirements for run-time support under SODA would reflect the lack of special cases for handling unwanted messages and multiple enclosures. Given the amount of code devoted to such problems in the Charlotte implementation, it seems reasonable to expect a savings on the order of 4K bytes.

For simple messages, run-time routines under SODA would need to perform most of the same functions as their counterparts for Charlotte. Preliminary results with the Butterfly implementation (described in the following section) suggest that the lack of special cases might save some time in conditional branches and subroutine calls, but relatively major differences in run-time package overhead appear to be unlikely.

Overall performance, including kernel overhead, is harder to predict. Charlotte has a considerable hardware advantage: the only implementation of SODA ran on a collection of PDP-11/23's with a 1-Mbit/second CSMA bus. SODA, on the other hand, was designed with speed in mind. Experimental figures reveal that for small messages SODA was three times as fast as Charlotte.[2] Charlotte programmers made a deliberate decision to sacrifice efficiency in order to keep the project manageable. A SODA version of LYNX might well be *intrinsically* faster than a comparable version for Charlotte.

## The Chrysalis Implementation

### Overview of Chrysalis

The BBN Butterfly Parallel Processor [4] is a 68000-based shared-memory multiprocessor. The Chrysalis operating system provides primitives, many of them in microcode, for the management of system abstractions. Among these abstractions are **processes**, **memory objects**, **event blocks**, and **dual queues**.

Each process runs in an address space that can span as many as one or two hundred memory objects. Each memory object can be mapped into the address spaces of an arbitrary number of processes. Synchronization of access to shared memory is achieved through use of the event blocks and dual queues.

An event block is similar to a binary semaphore, except that 1) a 32-bit datum can be provided to the *V* operation, to be returned by a subsequent *P*, and 2) only the **owner** of an event block can **wait** for the event to be **posted**. Any process that knows the name of the event can perform the post operation. The most common use of event blocks is in conjunction with dual queues.

A dual queue is so named because of its ability to hold either data or event block names. A queue containing data is a simple bounded buffer, and enqueue and dequeue operations proceed as

[2] The difference is less dramatic for larger messages; SODA's slow network exacted a heavy toll. The figures break even somewhere between 1K and 2K bytes.

one would expect. Once a queue becomes empty, however, subsequent dequeue operations actually *enqueue* event block names, on which the calling processes can wait. An enqueue operation on a queue containing event block names actually posts a queued event instead of adding its datum to the queue.

## A Third Approach to Links

In the Butterfly implementation of LYNX, every process allocates a single dual queue and event block through which to receive notifications of messages sent and received. A link is represented by a memory object, mapped into the address spaces of the two connected processes. The memory object contains buffer space for a single request and a single reply in each direction. It also contains a set of flag bits and the names of the dual queues for the processes at each end of the link. When a process gathers a message into a buffer or scatters a message out of a buffer into local variables, it sets a flag in the link object (atomically) and then enqueues a notice of its activity on the dual queue for the process at the other end of the link. When the process reaches a block point it attempts to dequeue a notice from its own dual queue, waiting if the queue is empty.

As in the SODA implementation, link movement relies on a system of hints. Both the dual queue names in link objects and the notices on the dual queues themselves are considered to be hints. Absolute information about which link ends belong to which processes is known only to the owners of the ends. Absolute information about the availability of messages in buffers is contained only in the link object flags. Whenever a process dequeues a notice from its dual queue it checks to see that it owns the mentioned link end and that the appropriate flag is set in the corresponding object. If either check fails, the notice is discarded. Every change to a flag is eventually reflected by a notice on the appropriate dual queue, but not every dual queue notice reflects a change to a flag. A link is moved by passing the (address-space-independent) name of its memory object in a message. When the message is received, the sending process removes the memory object from its address space. The receiving process maps the object *into* its address space, changes the information in the object to name its own dual queue, and *then* inspects the flags. It enqueues notices on its own dual queue for any of the flags that are set.

Primitives provided by Chrysalis make atomic changes to flags extremely inexpensive. Atomic changes to quantities larger than 16 bits (including dual queue names) are relatively costly. The recipient of a moved link therefore writes the name of its dual queue into the new memory object in a non-atomic fashion. It is possible that the process at the non-moving end of the link will read an invalid name, but only *after* setting flags. Since the recipient completes its update of the dual-queue name *before* inspecting the flags, changes are never overlooked.

Chrysalis keeps a reference count for each memory object. To destroy a link, the process at either end sets a flag bit in the link object, enqueues a notice on the dual queue for the process at the other end, unmaps the link object from its address space, and informs Chrysalis that the object can be deallocated when its reference count reaches zero. When the process at the far end dequeues the destruction notice from its dual queue, it confirms the notice by checking it against the appropriate flag and then unmaps the link object. At this point Chrysalis notices that the reference count has reached zero, and the object is reclaimed.

Before terminating, each process destroys all of its links. Chrysalis allows a process to catch all exceptional conditions that might cause premature termination, including memory protection faults, so even erroneous processes can clean up their links before going away. Processor failures are currently not detected.

## Preliminary Measurements

The Chrysalis implementation of LYNX has only recently become available. It consists of approximately 3600 lines of C and 200 lines of assembler, compiling to 15 or 16K bytes of object code and data on the 68000. Both measures are appreciably smaller than the respective figures for the Charlotte implementation.

Message transmission times are also faster on the Butterfly, by more than an order of magnitude. Recent tests indicate that a simple remote operation requires about 2.4 ms with no data transfer and about 4.6 ms with 1000 bytes of parameters in both directions. Code tuning and protocol optimizations now under development are likely to improve both figures by 30 to 40%.

## Discussion

Even though the Charlotte kernel provides a higher-level interface than does either SODA or Chrysalis, and even though the communication mechanisms of LYNX were patterned in large part on the primitives provided by Charlotte, the implementations of LYNX for the latter two systems are smaller, simpler, and faster. Some of the difference can be attributed to duplication of effort between the kernel and the language run-time package. Such duplication is the usual target of so-called **end-to-end arguments** [18]. Among other things, end-to-end arguments observe that each level of a layered software system can only eliminate errors that can be described in the context of the interface to the level above. Overall reliability must be ensured at the application level. Since end-to-end checks generally catch *all* errors, low-level checks are redundant. They are justified only if errors occur frequently enough to make early detection essential.

LYNX routines never pass Charlotte an invalid link end. They never specify an impossible buffer address or length. They never try to send on a moving end or enclose an end on itself. To a certain extent they provide their own top-level acknowledgments, in the form of goahead, retry, and forbid messages, and in the confirmation of operation names and types implied by a reply message. They would provide additional acknowledgments for the replies themselves if they were not so expensive. For the users of LYNX, Charlotte wastes time by checking these things itself.

Duplication alone, however, cannot account for the wide disparity in complexity and efficiency between the three LYNX implementations. Most of the differences appear to be due to the difficulty of adapting higher-level Charlotte primitives to the needs of an application for which they are almost, but not quite, correct. In comparison to Charlotte, the language run-time packages for SODA and Chrysalis can

(1) move more than one link in a message

(2) be sure that all received messages are wanted

(3) recover the enclosures in aborted messages

(4) detect all the exceptional conditions described in the language definition, without any extra acknowledgments.

These advantages obtain precisely because the facilities for managing virtual circuits and for screening incoming messages are *not* provided by the kernel. By moving these functions into the language run-time package, SODA and Chrysalis allow the implementation to be tuned specifically to LYNX. In addition, by maintaining the flexibility of the kernel interface they permit equally efficient implementations of a wide variety of other distributed languages, with entirely different needs.

It should be emphasized that Charlotte was not originally intended to support a distributed programming language. Like the designers of most similar systems, the Charlotte group expected applications to be written directly on top of the kernel. Without the benefits of a high-level language, most programmers probably would prefer the comparatively powerful facilities of Charlotte to the comparatively primitive facilities of SODA or Chrysalis. With a language, however, the level of abstraction of underlying software is no longer of concern to the average programmer.

For the consideration of designers of future languages and systems, we can cast our experience with LYNX in the form of the following three lessons:

**Lesson one**: Hints can be better than absolutes.

The maintenance of consistent, up-to-date, distributed information is often more trouble than it is worth. It can be considerably easier to rely on a system of hints, so long as they usually work, and so long as we can tell when they fail.

The Charlotte kernel admits that a link end has been moved only when all three parties agree. The protocol for obtaining such agreement was a major source of problems in the kernel, particularly in the presence of failures and simultaneously-moving ends [3]. The implementation of links *on top of* SODA and Chrysalis was comparatively easy. It is likely that the Charlotte kernel itself would be simplified considerably by using hints when moving links.

**Lesson two**: Screening belongs in the application layer.

Every reliable protocol needs top-level acknowledgments. A distributed operating system can attempt to circumvent this rule by allowing a user program to describe *in advance* the sorts of messages it would be willing to acknowledge if they arrived. The kernel can then issue acknowledgments on the user's behalf. The shortcut only works if failures do not occur between the user and the kernel, and if the descriptive facilities in the kernel interface are sufficiently rich to specify precisely which messages are wanted. In LYNX, the termination of a coroutine that was waiting for a reply can be considered to be a "failure" between the user and the kernel. More important, the descriptive mechanisms of Charlotte are unable to distinguish between requests and replies on the same link.

SODA provides a very general mechanism for screening messages. Instead of asking the user to *describe* its screening function, SODA allows it to provide that function itself. In effect, it replaces a static description of desired messages with a formal subroutine that can be called when a message arrives. Chrysalis provides no messages at all, but its shared-memory operations can be used to build whatever style of screening is desired.

**Lesson three**: Simple primitives are best.

From the point of view of the language implementor, the "ideal operating system" probably lies at one of two extremes: it either provides everything the language needs, or else provides almost nothing, but in a flexible and efficient form. A kernel that provides some of what the language needs, but not all, is likely to be both awkward and slow: awkward because it has sacrificed the flexibility of the more primitive system, slow because it has sacrificed its simplicity. Clearly, Charlotte could be modified to support all that LYNX requires. The changes, however, would not be trivial. Moreover, they would probably make Charlotte significantly larger and slower, and would undoubtedly leave out something that some other language would want.

A high-level interface is only useful to those applications for which its abstractions are appropriate. An application that requires only a subset of the features provided by an underlying layer of software must generally pay for the whole set anyway. An application that requires features *hidden* by an underlying layer may be difficult or impossible to build. For general-purpose computing a distributed operating system must support a wide variety of languages and applications. In such an environment the kernel interface will need to be relatively primitive.

## Acknowledgments

Much of the research described in this article was conducted in the course of doctoral studies at the University of Wisconsin under the supervision of Associate Professor Raphael Finkel.

## References

[1] G. R. Andrews, "The Distributed Programming Language SR — Mechanisms, Design and Implementation," *Software — Practice and Experience* 12 (1982), pp. 719-753.

[2] G. R. Andrews and R. A. Olsson, "The Evolution of the SR Language," TR 85-22, Department of Computer Science, The University of Arizona, 14 October 1985.

[3] Y. Artsy, H.-Y. Chang, and R. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Computer Sciences Technical Report #554, University of Wisconsin – Madison, August 1984.

[4] BBN Laboratories, "Butterfly® Parallel Processor Overview," Report #6148, Version 1, Cambridge, MA, 6 March 1986.

[5] A. P. Black, "Supporting Distributed Applications: Experience with Eden," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 181-193.

[6] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, p. 160. Abstract only; full paper to appear in *ACM TOCS*.

[7] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 128-139. In *ACM Operating Systems Review* 17:5.

[8] D. J. DeWitt, R. Finkel, and M. Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," Computer Sciences Technical Report #553, University of Wisconsin – Madison, September 1984.

[9] R. Finkel, M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project," Computer Sciences Technical Report #502, University of Wisconsin – Madison, October 1983.

[10] D. Gelernter, "Dynamic Global Name Spaces on Network Computers," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August 1984, pp. 25-31.

[11] M. B. Jones,, R. F. Rashid, and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985, pp. 225-235.

[12] J. Kepecs, "SODA: A Simplified Operating System for Distributed Applications," Ph. D. Thesis, University of Wisconsin – Madison, January 1984. Published as Computer Sciences Technical Report #527, by J. Kepecs and M. Solomon.

[13] J. Kepecs and M. Solomon, "SODA: A Simplified Operating System for Distributed Applications," *ACM Operating Systems Review* 19:4 (October 1985), pp. 45-56. Originally presented at the *Third ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, 27-29 August 1984.

[14] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS* 5:3 (July 1983), pp. 381-404.

[15] S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," Report CS-R8418, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1984.

[16] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 110-118. In *ACM Operating Systems Review* 17:5.

[17] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 14-16 December 1981, pp. 64-75.

[18] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM TOCS* 2:4 (November 1984), pp. 277-288.

[19] M. L. Scott and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August 1984, pp. 395-401.

[20] M. L. Scott, "Design and Implementation of a Distributed Systems Language," Ph. D. Thesis, Technical Report #596, University of Wisconsin – Madison, May 1985.

[21] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," TR 183, Department of Computer Science, University of Rochester, January 1986. Revised version to appear in *IEEE Transactions on Software Engineering*, December 1986.

[22] R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 73-82. In *ACM SIGPLAN Notices* 18:6 (June 1983).

[23] R. E. Strom and S. Yemini, "The NIL Distributed Systems Programming Language: A Status Report," *ACM SIGPLAN Notices* 20:5 (May 1985), pp. 36-44.

[24] D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann, "The Structure of Cedar," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, 25-28 June 1985, pp. 230-244. In *ACM SIGPLAN Notices* 20:7 (July 1985).

[25] United States Department of Defense, "Reference Manual for the Ada® Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983.

[26] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 49-70. In *ACM Operating Systems Review* 17:5.

[27] N. Wirth, *Programming in Modula-2*, Texts and Monographs in Computer Science, ed. D. Gries, Springer-Verlag, Berlin, Third, Corrected Edition, 1985.

# DISTRIBUTED PROCESSING UNDER THE DRAGON SLAYER OPERATING SYSTEM

Carl B. Friedlander
The BDM Corporation
McLean, Virginia 22102 / U.S.A.

Horst F. Wedde
Computer Science Department
Wayne State University
Detroit, Michigan 48202 / U.S.A.

## Abstract

DRAGON SLAYER is a distributed operating system for microprocessor based networks. It is currently implemented on a Carrier Sense Multiple Access (CSMA) network of 16 Leading Edge PCs. It provides message passing and uses a standard client/server model. A unique aspect of the system is the lack of any global process existence, global system kernel or central authority. In this paper, we explain our use of broadcast messaging and our mechanism for consistent and unique process naming. We achieve fair resource scheduling without any assumption on global management or control using a novel distributed algorithm derived from constructions using the Theory of Interaction Systems. Further, we address the usefulness of DRAGON SLAYER for dealing with data integrity in distributed data bases, for distributed vision processing applications, and for providing cheap, efficient, robust solutions for real-time process control problems in automatic manufacturing (factory-of-the-future).

## Introduction

An increasing number of operating systems for distributed applications on microprocessor networks have been developed during the past five or six years. If one considers as typical and established examples the Stanford V-Kernel [6,7] or the INRIA CHORUS [1,2,9] systems, one finds that in order to achieve a high modularity and flexibility at design and run time, all communication is based on message passing mechanisms. These systems create and maintain global name tables at every site through a remainder central service, a global kernel. Such an architecture makes sense in a local area network, with a central node responsible for message handling, name tables, process scheduling, managing large data files, etc., and with a standard connection (Ethernet). In a network with peer authority at all nodes, there should be no asymmetrical solution to handling control and management problems. Asymmetric solutions can cause undesirable message overheads in view of envisioned large distances between nodes of future distributed systems. Also, systems built on a master-slave relationship are vulnerable to failure since a master breakdown immediately halts all communication.

With DRAGON SLAYER, a successful attempt has been made to develop and implement a distributed operating system without any form of centralized service or global kernel. We are not aware of any other system of this type. In DRAGON SLAYER, resource management has been distributed to the process level, requiring that we solve the general conceptual problem of fair distributed resource scheduling. Since resources or server processes are normally available at several sites, we insure that a set of resources requested by a process will eventually be assigned so long as such resources are available anywhere in the system. This restriction holds even if server processes initially dedicated to providing the requested resources fail (e.g. through node or link communication failures). Our system is therefore extremely **robust**. Robustness and peer authority at all nodes were major design goals for DRAGON SLAYER. In our paper, we describe the main ideas and structures of this development.

After explaining the hardware configuration for our current implementation, we mention our integration of available software (MS-DOS) and the principles behind our system construction. We discuss the particularities of our message passing mechanism, broadcast messages and distributed naming scheme. A detailed outline is given of our distributed process scheduling and resource allocation handling algorithm.

Acknowledgements

This work was partially supported by the US Army (DAAE07-83-C-R094), under DARPA grant no. MDA 903-85-C-0056, and by the State of Michigan (IPPRI-85-140625).

In order to construct the needed distributed algorithm, we made use of formal results of the Theory of Interaction Systems, enhancing a deadlock-free, but not necessarily fair, algorithm by J. Winkowski [19] which was already in a suitable format and which was the first algorithm presented for this purpose which placed no unrealistic restrictions on the behavior of the processes involved. This algorithm and the formal background mentioned are specified in an appendix. Finally, the conceptual and developmental achievements of the DRAGON SLAYER operating system are discussed in light of existing solutions and of similar conceptual results. We point out the advantages of completely distributed operating systems like DRAGON SLAYER for future distributed applications in real-time processing (factory-of-the-future). We also discuss how DRAGON SLAYER provides for cheap, efficient, robust implementations of distributed algorithms for canonical vision tasks.

## 1. The Hardware Environment

DRAGON SLAYER was targeted for use on a network of Intel 8088 or 8086 based systems. It currently runs on a Carrier Sense Multiple Access (CSMA) network of 16 Leading Edge Personal Computer systems. These systems each contain 512K of local memory, a memory mapped character display and two 320k byte floppy disk drives (see figure 1). The systems are interconnected with a 1 Megabit per second network. Two of the systems have printers connected to them, two of the systems are connected via 9600 BAUD lines to the university mainframe host and one of the systems has an additional 10 megabyte disk drive (see figure 2). None of the individual systems is truly dedicated to any particular function, and the operating system kernel resident on any given system has not been tailored to augment particular functions. However, the systems to which the printers are connected must function as print servers at some time, the systems connected to the mainframe host must serve as gateways and the system connected to the winchester disk must serve as a file server. Our intention has been to allow each system to fulfill its service function without requiring dedication to that particular function.

## 2. The Software Environment

The DRAGON SLAYER system functions within and augments the operation of the MS-DOS or PC-DOS operating system, a single user operating system not designed to allow process communication, multiprocess execution, multitasking, concurrency or parallel program execution. Of chief concern during the design and construction of DRAGON SLAYER has been the philosophical imperative that MS-DOS functions would be unchanged and available to user processes. We have deviated from that philosophy as little as possible. Maintaining the MS-DOS functions has allowed development of applications under existing MS-DOS program development tools. Most user programs that currently can run under MS- DOS continue to run under DRAGON SLAYER's guidance. No disk or other hardware support facilities, other than network message facilities have been added. One clear advantage to using this approach was our ability to add a few procedures to TURBO PASCAL, a commercially available version of PASCAL for Personal Computer systems, and create a usable multiprogramming language in which to develop our application base.

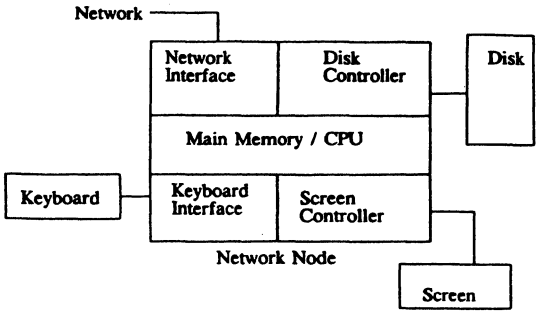


Figure 1
Configuration of a single network node

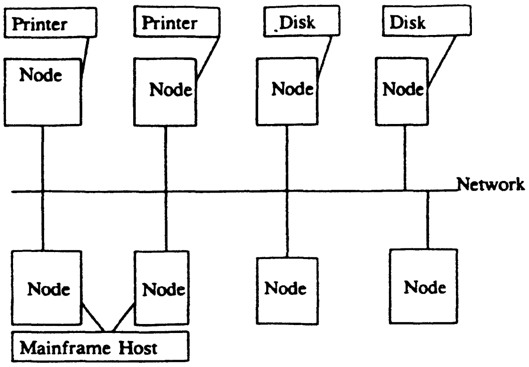


Figure 2
Network Configuration

## 3. Principles of System Construction

The DRAGON SLAYER System is based on the client/server model which is common to many of today's distributed systems[1,2]. System resources are provided to the client processes through server processes while the operating system attempts to ensure fairness of resource allocation and efficiency of operation. When a client process wishes to use a

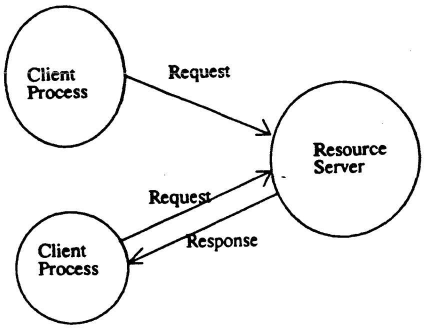


Figure 3.
Client/server Model

particular system resource it must request use of that resource from the resource server and then become a client of the resource server (see figure 3). The use of self scheduling servers without global control of resources introduces a conceptual problem of synchronization and fair resource scheduling that is not present in systems which feature centralized scheduling strategies. Our system requires that processes follow a strict model for resource request and use.

The resource client is insulated from the details of handling a particular resource by the server which presents a well defined interface, an abstract or virtual resource, to the client process. In the DRAGON SLAYER system, the client and server are further insulated by the necessity of communicating through use of the message passing primitives provided by the operating system. All server interfaces are defined in terms of the messages sent to the server by client processes.

The DRAGON SLAYER system has been developed without any form of central control. As a result, client and server processes act with a maximum amount of process autonomy and a large share of the responsibility for allowing the system to function properly. As in other message passing environments, DRAGON SLAYER processes, compete for access to resources which are available only for one of them at a time, and can be considered as members of temporarily groups of processes requesting overlapping sets of resources. However, we have no main process or group manager, and the group members do not have information about other group members other than information available from the resources over which, peers may compete. The sequence of activities through which a process accesses a mutually exclusive set of resources needed for a particular process step or **critical section** are as follows:

1. Establish a list of servers on the system.
2. Request commitment from servers in the resource list.
3. Initiate processing, once the resources/services are available.
4. Return resources and
5. Free the servers.

Once a process reaches step five, it may either terminate existence or return to step two and initiate another processing cycle in which it utilizes a new set of resources. The main process may look for new resource servers to add to its list at any time.

## 4. Message Passing

The DRAGON SLAYER operating system incorporates novel concepts in process addressing and organization. It is a message passing operating system based on a standard client/server model. The sending mechanism is displayed in figure 4. Unique aspects of this work include the lack of use of any global process existence, name or address table, and the use of only asynchronous non-blocking messages. Each processor on the network has a unique name assigned in hardware on the network interface board. Each active, memory resident process has a unique name composed of the name of the processor on which it resides and the segment-offset address of the first byte of executable code within the process. (In this way unique process naming is an indirect product of a process coming into existence at a node.) The mechanism utilized to locate a process is the universal broadcast primitive of the message passing function. This primitive replicates a transmitted message to all processes troughout the network system. As every process instantiation will receive a copy of such a message, locating the process becomes the act of sending an "are you there" message and receiving a "yes I am" reply. Once two processes have exchanged addresses they can communicate directly. The lack of any centralized numbering or naming convention

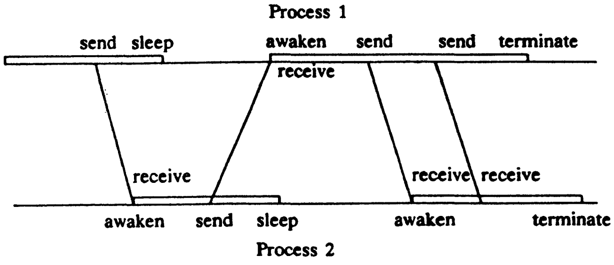


Figure 4
Interprocess Communication

allows the user complete flexibility in the broadcast messages which he wishes to use and to which he wishes to respond. (Obviously, if a user wishes to communicate with a printer he will have to obey the semantics of the messagesof printer servers.) Message passing and interprocess communication is handled through the functions of **message buffer request**, **message buffer free**, **message send** and **message retrieve**. Message buffer request is used to obtain a free message buffer from the operating system. This buffer may be used to transport information from one process to another. Message buffer free is used to return a used buffer to the operating system. Message send is used to request that a message buffer be transported from the current creating process to the process or processes identified as recipients of that message. Message buffer retrieve is used by a process to examine any message buffers that have been sent to it. Messages consist of a destination address, sender's address, control information and text information. High level interface functions remove the possibility of loss of message buffers by providing composite functions that properly integrate requests for, and release of, buffers. Message buffers are maintained in independent local pools. Each node maintains its local pool.

The sender's address is always the three part **segment number / offset number / processor number** address of the sending process. The destination address is either the specific segment / offset / processor address of the desired process or may be specified as a broadcast message to be delivered to all processes on all systems.

## 5. Process Scheduling

Each processor is engaged in scheduling the execution of the processes that are locally memory resident. Ready-to-run processes are scheduled for execution in a round-robin order. A process is deemed ready to run if its timer has been decremented to zero or if it has a message pending. A key problem in process scheduling is **fairness,** sometimes called starvation-freeness. Requested resources must be available after some finite time. Because a server can join more than one group simultaneously it is possible that processes in each of two groups may simultaneously attempt to utilize their shared resources. We achieve fairness in deciding conflicts between clients using a distributed algorithm which can be outlined as follows:

For entering one of its critical sections, say *cs(1)*, a process P must proceed through the sequence of activities mentioned in section 3. Communication, broadcast or direct, would have to occur in these activity sections. For short, we call "establish a list of servers on the system" *rg* ("registration"), the types of resources needed in order to execute *cs(1)* would be R(1),...,R(n). cl(1), cl(2),...,cl(n) will be names for the phases in which the servers providing access to R(1),...,R(n), respectively, would have been freed. Every section different from the critical one is accessible. The critical section is free for access only if none of the servers in the servers' list is serving another process. The **remainder section** (with respect to *cs(1)* is everything outside the activity sequence used to attach the particular set of resources needed in *cs(1)* (see section 3).

The behavior of process P in order to enter and execute a critical section is given by the flow chart scheme in figure 5 which explains the structure for making a transition from one of the sections specified above, into the next. As part of the algorithm processes will reach a state of competition in which they play a distributed game which is developed from an algorithm by J. Winkowski [19] and which is deadlock-free in the sense that one of the players wins after a finite time. This game is not fair. Fairness is achieved as follows: The servers/resource managers are responsible for communication about resources. In particular, they are responsible for communication with or among competing processes. In phase cl(i), the given process P is subject to a locking influence from a neighbor competing with P about R(i) if this neighbor were in *rg*. In *rg*, P exerts a similar influence on its neighbors when these have just released R(i). While P has not yet left *cs(1)* it is subject to a temporarily locking influence from any neighbor being in a state where it has released the shared resource but has not yet entered its remainder section. The corresponding communication is performed through the resource managers involved in the particular competition. Thus, it is really the partial knowledge of these managers alone on which the progress of the involved processes depends.Every process proceeds according to the scheme given in figure 5, and all servers react to the best of their knowledge. In this way, we guarantee that P will, after a finite time, be able to enter its critical section *cs(1)*.

The process and communication mechanism displayed in figure 5 for one process P, for entering one of its critical sections *cs(1)*, looks fairly straightforward because it is supposed only to give an idea about the two separate steps of communication needed for P to leave one section and enter the next, respectively. As usual for mutual exclusion problems, the correctness proof is not straightforward, even in the special case of only one critical section per process. It is not easy to see that no process can take advantage of a neighbor through a locking influence. The scheme in figure 5 is

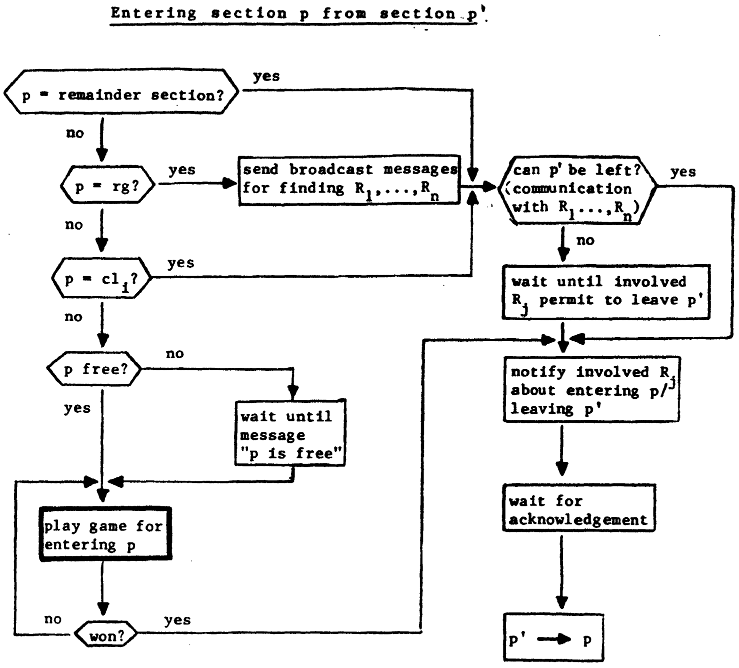

Entering section p from section p'
p = remainder section?
yes
no
p = rg?
yes
send broadcast messages for finding R1,...,Rn
can p' be left? (communication with R1...Rn)
yes
no
wait until involved Rj permit to leave p'
p = cli?
yes
no
p free?
no
yes
wait until message "p is free"
notify involved Rj about entering p/ leaving p'
play game for entering p
wait for acknowledgement
no
won?
yes
p' → p

Figure 5
Critical Section entrance behavior of Process P

the same for cases of multiple critical sections with varying sets and kinds of requested resources. In order to verify this modularity of our design approach, it has been proven [14] that the mechanisms implemented for each critical section do not "interfere". More technical details about the implemented general algorithm and the game, can be found in the appendix.

## 6. Discussion and Future Work

We have briefly described some main features of the DRAGON SLAYER operating system. It allows fully distributed control of interacting and competing processes, without central control and without use of distributed relics of central control such as distributed name tables, group identifiers or multigroup identifiers. Through implementation, we have demonstrated the viability of our construction. The DRAGON SLAYER design principles feature novel aspects. Under completely decentralized control we use broadcast messages as universal communication primitives, and for resource scheduling we implemented a general and fair distributed algorithm.

Broadcast messaging is a necessary function to avoid use of central control for establishment of direct communication between processes. Rather than multicast, broadcast messaging was incorporated in DRAGON SLAYER and provides a means for any process to communicate with all other processes without invoking the operating system as a message filter. In a single broadcast network like ours, the network load for broadcast is the same as for multicast messaging. Under our scheme, the operating system need not maintain any tables with information regarding resident processes. Admittedly, the use of broadcast messages requires that all processes be awakened so that they may actively ignore broadcast messages. On the other hand, the use of multicast messages requires that the operating system be aware of all local elements of all multicast groups introducing a kind of centralized control. We have avoided the use of such distributed lists in our system.

A major focus of our current experimentation is on distributed vision applications: distributed scene analysis with moving objects to be located and identified, like in the Autonomous Land Vehicle project (DoD) or in factory-of-the-future environments. In such distributed applications, the amount of local computation is much higher than the amount of communication. Since broadcast messaging in DRAGON SLAYER only occurs when a process locates the resources needed, we can benefit from the high amount of process autonomy without being impacted by the broadcast costs. Currently, we are performing comparative studies using sequential and parallel versions of algorithms for canonical vision tasks – static object location and measurement of their features – as a benchmark test for our network performance. We compare up to 4 processors under

DRAGON SLAYER against a VAX 11/780. Similarly to the results reported in [13], our first results give evidence that the speedup resulting from the parallelization more than outweighs the relatively modest computational power of our micro-computers. A performance of a mid-size computer can be reached or surpassed at minimal hardware and software expense. While the parallel algorithms used for the benchmark tests could easily be realized in hardware, because of the primitivity and repetitivity of the used operations we envision a major practical benefit of our system for dynamic scene analysis, with moving objects and objects coming into, or leaving, a frame. In this application, higher-level software operations would typically be distributed into partitioned search tasks.

The scheduling algorithm we use results from solving a conceptual problem. The existence of such a general algorithm was investigated by using formal tools, within the Theory of Interaction Systems [11,18]. Our results easily allowed specification of another distributed algorithm for the same purpose, but without use of resource managers [14,15]. Recently another such algorithm, also not using resource managers, was found independently [5]. It uses the same idea of local precedence changes as found in [19].

In our implementation, time-out mechanisms prevent processes from being blocked in a situation where a node with resources dedicated to such processes breaks down during performance of its task. In case of such a node failure or link failure, the requesting process will make another request for the desired resources. Through our formal specification method, as an extended fairness requirement for which the proof can be found in [14], a process registered for accessing resources will eventually get them (maybe after several attempts) as long as such resources are available anywhere in the system. Thus, DRAGON SLAYER exhibits an unsurpassed degree of *robustness*, due to the complete lack of centralized control functions. Future research is targeted towards fault tolerance, data integrity, and error recovery at user levels, in the presence of faulty nodes.

Our development provides the basis for innovative forms of distributed real-time processing with peer authority at every node and cheap, long-distance communication, through the envisioned progress in fiber optics technology. Such developments are under discussion in the vendor industry [3]. In order to provide for high-level real-time applications of distributed microprocessor networks a really distributed compiler for a CSP-like language has been developed in Milan/ Italy [4]. It is now in the phase of becoming an industrial product (Olivetti). We are cooperating with this group to implement it on DRAGON SLAYER, as a first step into office automation applications. Another direction for distributed real-time applications research with the DRAGON SLAYER system, then running on a 68000-based network with Ethernet at Wayne State University, is to support a project in real-time, multi-layer distributed production control with distributed information management (factory-of-the-future).

## References

[1] J.S. BANINO, J.C. FABRE, M. GUILLEMONT, G. MORISSET, M. ROZIER, "Some Fault-tolerant Aspects of the CHORUS operating system"; **Proc. of the 5th International IEEE Conference on Distributed Computing Systems,** May 1985

[2] J.S. BANINO, G. MORISSET, M. ROZIER, "Controlling Distributed Processes with CHORUS Activity Messages"; **Proc. of the 18th Hawaii International Conference on System Sciences,** January 1985

[3] L.W. BEERS, "Organizational Influence on Future Information Systems Architecture"; in: **Innovative Computer Developments and Their Impact on Organizations special issue of Systems Research,** (Horst F. Wedde ed.); Vol.2 No.4(1985), Pergamon Press

[4] G. CASTELLI, F. DE CINDIO, G. DE MICHELIS, C. SIMONE, "The GCP Language and its Implementation"; **Proc. of the IFIP workshop "Languages for Automation"**; New Orleans, October 1984

[5] K.M. CHANDY, J. MISRA, "The Drinking Philosophers Problem"; **TOPLAS** Vol.6 No.4(1984)

[6] D.R. CHERITON, THE V KERNEL: "A Software Base for Distributed Systems". **IEEE Software,** April 1984

[7] D.R. CHERITON, M.A. MALCOLM, L.S. MELEN, G.R. SAGER THOTH, "A Portable Real-time Operating System". **CACM** Vol. 22 No. 2(1979), pp 105-115

[8] E.W. DIJKSTRA, **Two Starvation-free Solutions to a General Exclusion Problem**; EWD 625, Plataanstraat 5, 5671 AL NUENEN, The Netherlands

[9] M. GUILLEMONT, "The CHORUS Distributed operating system: Design and Implementation". **Proc. of the IFIP TC 6 International In-Depth Symposium on Local Computer Networks,** 1982

[10] C.A.R. HOARE, "Communicating Sequential Processes"; **CACM** Vol. 21 No. 8(1978)

[11] A. MAGGIOLO-SCHETTINI, H. WEDDE, J. WINKOWSKI, "Modeling a Solution of a Control Problem in Distributed Systems by Restrictions". **Theoretical Computer Science** 13(1981), 61 - 83; North Holland

[12] "MAP: THE TIE THAT BINDS; COMMUNICATION ON THE PLANT FLOOR"; **GM Public Affairs Newsletter,** Vol.14 No.8 (September 1984)

[13] P.G. SELFRIDGE, S. MAHAKIAN, "Distributed Computing for Vision: Architecture and a Benchmark Test"; **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMI-7 No.5 (1985)

[14] H. WEDDE, "Designing Fair Distributed Scheduling Algorithms"; submitted to International Symposium Mathematical Foundations of Computer Science MFCS'86

[15] H. WEDDE, "A General Distributed Graph Algorithm for Fair Access to Critical Sections; invited paper": **First International Workshop "Distributed Algorithms on Graphs"**, Ottawa, August 1985

[16] H. WEDDE, "A Formal Basis for Correct Implementation of Distributed Programming Languages"; **Proc. of the 5th International IEEE Conference on Distributed Computing Systems**, May 1985

[17] H. WEDDE, "Value of Formal Information System Models for a Flexible Reorganization in an Insurance Company"; in: J.C. AGRAWAL, P. ZUNDE (ed.): **Empirical Foundations of Software and Systems Sciences**; Plenum Press 1985

[18] H. WEDDE, "An Iterative and Starvation-free Solution for a General Class of Distributed Control Problems Based on Interaction Primitives". **Theoretical Computer Science** Vol. 24(1983), 1 - 20; North Holland

[19] J. WINKOWSKI, "Protocols of Accessing Overlapping Sets of Resources"; **Information Processing Letters** Vol. 12 No. 5(1981); North Holland

## Appendix

The DRAGON SLAYER resource scheduling algorithm resulted from an exciting discussion about Dijkstra's Dining Philosophers years ago: In 1979, one of the authors was asked to study Dijkstra's most recent contribution to the theme of fair scheduling of distributed processes [8] which turned out not to be fair without substantial assumptions on centralized control. Using the Theory of Interaction Systems, a specification and analysis tool for distributed systems [11,14,18], a starvation-free solution for Dijkstra's generalized distributed process scheduling problem was developed in an **incremental** procedure and proven correct. The main idea behind the formal construction was a suitable enhancement method for any deadlock-free realization of mutual exclusion without central control. Given such an algorithm one could immediately write down a starvation-free or fair distributed algorithm. J. Winkowski presented such an algorithm in 1981 [19] and no problem remained.

Winkowski's algorithm, besides being based on explicit message passing, was the first to make no assumption on boundedness of "hungry" periods (called registration *rg* phases in section 5). In its main outline, the algorithm works as follows:

Each process is assumed to have just one critical section which may be periodically visited. Each process p, at a time when it wants to enter its critical section, needs a set A(p) of resources. In order to make the requests, p sends visiting cards to the resource/server processes in A(p). The requesting processes have a local priority list which contains priority information with respect to neighbor processes, i.e. processes also possibly requesting resources in A(p). The resource processes have queues for storing visiting cards and store the information about the local priorities obtained from their clients. In order to have coherent information, it is assumed that *initially* all local priorities are projections of a global precedence relation. (This is easily achieved in DRAGON SLAYER by using the processor, segment and offset numbers of the requesting processes.)

The algorithm is a game in which a process wins iff it succeeds in having all visiting cards in the top position of each resource queue in A(p).

If p is interested in accessing its critical section, it sends visiting cards while the following conditions hold:

-There are no winning cards of the neighbor players in resource queues.
-There are no visiting cards in resource queues from players with higher priority. P's card in such a queue already preceeds such a card.

P is to stop leaving cards and collect or take back already placed cards when one of the following situations occurs:

-A winning card from another player appears at a resource in A(p).
-A card from another player with a higher local priority is in the resource list before p has succeeded in getting its card enqueued.

After P wins and the other competing processes have removed their cards from the resource lists, p accesses its critical section. Then P reverses all its local priorities and notifies the resource processes which in turn notify p's neighbors in order to cause the corresponding changes. Then p releases the resources. The key idea of the algorithm is that all local priority lists, through the indicated local operations, can still be understood as local projections of a (changed) global precedence relation. No deadlock would occur in the game since one of the competing processes always wins.

The extension of this algorithm into a starvation-free one is constructed by using the Theory of Interaction Systems. The detail of this construction is available in [18]. The main idea is as follows:

The formal objects corresponding to distributed processes are called **parts**, their relevant sections are called **phases**. Two formal relations between phases of different parts are used as primitives to define any kind of interaction or cooperation between distributed processes: a **coupling relation** specifying mutual exclusion of process states, and an **excitement relation** defining basic asymmetric forms of influence between processes.

A concept of global state and of (formal) operational semantics is then developed without making any reference to global time, purely using local information such as local states or phases, process steps or local events. Entering a next phase is **possible** unless specified mutual exclusion requirements would be violated. In Dijkstra's original Dining Philosophers' Problem the 5 philosophers involved had a critical section ("eating") and a remainder section ("thinking"). Figure 6 shows a corresponding Interaction Systems representation in which all events in the remainder sections are possible and where section *e* can be entered only if **both** neighbors do not have access to their sections *e*. Thus figure 6 is a formal framework for a deadlock-free algorithm.

The excitement relations specify asymmetrical local influences from one part to another. If a part b(1), being in phase p(1), excites b(2), being in phase p(2), then b(1) cannot leave p(1) unless b(2) has not left p(2). The effect on b(2) is described by axioms which account for the absence of any form of global control or management. For an elementary example, consider a process P, being in section *r*, which requests a resource R: P cannot leave *r* until R has been allocated to P. The request triggers service processes through which the global resource manager, or supervisor in a conventional environment would eventually make the requested resource available.

This formal theory, though unconventional, has been proven to be a powerful and flexible modeling and analysis instrument for distributed systems [11,14,16,17,18]. In [18] it was used to construct, in an incremental procedure, an easy extension for the solution in figure 6 such that *gradually* all conflicting requirements of the problem were satisfied. The result is shown in figure 7. It has been proven that given the same interaction specification between neighbors in an arbitrary topology, the resulting formal system specifies a fair or starvation-free solution for Dijkstra's generalized resource scheduling problem. The flow diagram in figure 5 is then an immediate translation of this formal specification into an algorithmic one.

In general, processes may have several critical sections, with varying numbers, and different kinds, of resources requested. They also may die or be newly generated. The major advantage of our formal specification method is that it allows for an easy proof of the fact that communication mechanisms constructed for two different critical section problems, with an appropriate formal specification similar to that in figure 7, do not interfere. (The technical details can be found in [14].) Thus, the final general distributed resource scheduling algorithm can, in a modular way, be composed through iterating distributed procedures each of which solves a single section problem and which was a result of directly breaking down a formal specification, after formally proving its correctness.

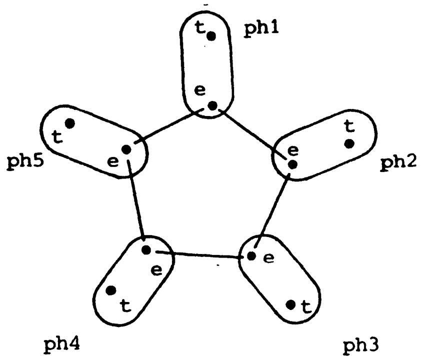


Figure 6
Mutual exclusion between dining philosophers

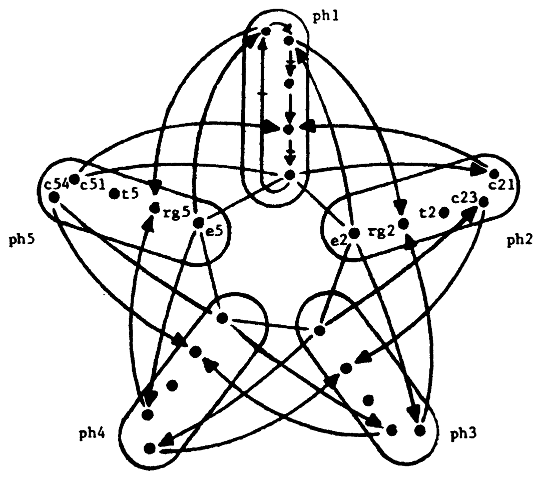


Figure 7
Fair interactions of dining philosophers

# PROGRAMMING SOLUTIONS TO THE ALGORITHM CONTRACTION PROBLEM

Philip A. Nelson, Lawrence Snyder
Computer Science Department, FR35
University of Washington
Seattle, Washington 98195

## Abstract

Algorithms for the parallel solution of problems are usually designed assuming an unlimited number of processors. Physical parallel machines have a fixed number of processors. The algorithm contraction problem arises when an algorithm requires more processors than are available on the physical machine. We present tools for comparing algorithm contractions based on bottle neck communication paths. We apply these tools to minimum, matrix product and sorting.

## Introduction

Algorithms for parallel computers are usually designed assuming an unlimited number of processors. For non-shared memory parallel algorithms, this assumption generally manifests itself by the algorithm utilizing one processor "per point", or some other input size-dependent processor allocation. The physical machine has only a fixed number of processors, of course, which will almost certainly be less than the number required by the algorithm. In order to make the logical processes of the algorithm conform to the physical processors of the machine, we must group processes together into a module to be executed on a single physical machine. This activity is called *contraction*[13]. The way this contraction is performed can have a significant affect on performance.

Consider two examples based on an grid $n \times n$ of processes, *i.e.* the processes communicate with their four nearest neighbors:

(1) There is much process-to-process communication and approximately equal computation required of each process.

(2) There is little process-to-process communication and the amount of computation per process is proportional to its $j$ index, *e.g.* process $i,j$ iterates $j$ times.

Suppose we have only one fourth the required number of processors and now compare two ways of forming contractions of four processes per processor[4]: *Coalescing* groups of adjacent 2×2 subarrays; *folding* groups as if the grid is folded in half and then in half again, *i.e.* $i,j$ $(1 \le i,j \le \frac{n}{2})$ is associated with $i,n-j+1$, $n-i+1,j$ and $n-i+1,n-j+1$. Clearly, algorithm (1) should be contracted by coalescing because the process-to-process communication for the processes sharing the same processor will become intraprocessor communications (*i.e.* fast memory references) rather than slow interprocessor communication; folding would not be as attractive because no communication is saved by locality. Alternatively, algorithm (2) should be contracted by folding because the work is balanced since each processor will perform a matching amount of long and short computations; coalescing would not be as attractive because the processors receiving processes with large indexes will become a bottleneck.

Using the results of Berman and her colleagues[3], an algorithm can be be automatically contracted, and this seems to be the best approach when nothing is known about the algorithm. At the other end of the spectrum, however, the programmer has "complete" knowledge about the algorithm. How should he be guided when performing his own contraction? In this paper we develop some apparatus to guide the programmer who must contract an algorithm. We will provide some case studies of contraction that show an unexpected diversity and we offer some general contraction strategies that can find application in other algorithms. Contraction is a nontrivial problem for parallel programmers[13], and so a secondary goal here is to expose it as an important topic for study and a subject suitable for rigorous analysis.

## Definitions

The generic parallel architecture under consideration in this paper is a non-shared memory model. It is a collection of homogeneous sequential computers operating asynchronously and connected in a communication network that is a bounded degree graph[13]. A single "edge" in the graph provides bidirectional communication between two processors. The CHiP[11] architecture is an example of this generic architecture.

The method used for programming this model consists of defining a sequential program for each processor and a communication graph. We are assuming a configurable architecture. (The problem of mapping a communications graph onto a different processor connection graph is discussed by Berman and Snyder[4] and Bokhari[5].) Communication is explicitly shown in the sequential code by specifying a data value to be sent to the processor connected by a given edge.

The algorithm contraction problem arises when an algorithm that is designed for use on $n$ processors must be mapped to a physical parallel computer with only $p < n$ processors. The programmer must decide which logical processes are to be mapped to the physical processors. Assuming that the logical processes have balanced loads (they run for the same length of time), we would like the physical processors to have balanced loads. This is done by mapping the same number of logical processes to each physical processor. The number of logical processes assigned to two arbitrary physical processors should differ by at most an additive constant $c$. For most contractions, it would be best to have $c=1$.

The contraction induces a communication graph for the $p$ physical processors. This new graph is defined by logical processes needing to communicate with other logical processes not mapped to the same physical processor. We assume that if a logical process in processor $i$ needs to communicate with a logical process in processor $j$, there is a physical edge connecting the two processors in the new graph. The contraction may map many of these logical edges to one physical edge in the new graph. That is, we are allowing only one edge between physical processors. Under the assumption of a bounded degree graph for the generic architecture, this induced graph must also be of bounded degree.

As an example of contraction, let us assume we have an algorithm with a tree graph. Consider the contraction to 5 processors shown in Figure 1. This contraction caused an increase in the degree; for example, the new root vertex has four descendants. Using this kind of a contraction, it can be shown that given $p$ processors, contracting an algorithm with at least $p^2$ logical processes requires degree $p-1$. Figure 2a shows a contraction of the tree to 4 processors. An extension of this method yields a binary tree in the $p$ processors.

Figure 2b gives another contraction to 4 processors. This contraction is derived by the recursive tree construction given by Leiserson[9]. Given two instances of a tree each with an associated free node, we can build a new tree and an associated free node. This produces a linear area layout in the plane with several desirable properties, one of which is the constant number of external edges.

In this paper, we will be considering the contribution of the communication time to the performance of the contracted algorithm. Unless otherwise stated, we assume that a communication between processing elements costs a fixed time $t_c$. During this time, no other communication in

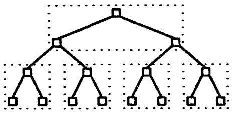
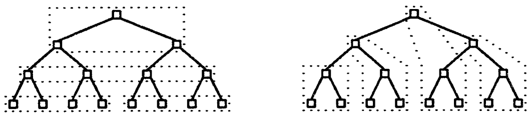


Figure 1: A 5 processor contraction.



(a) (b)

Figure 2: Two 4 processor contractions.

Supported in part by National Science Foundation Grant DCR-8416878
and by Office of Naval Research Contract No. N00014-85-K-0328.

the same direction may take place. We are specifically allowing all edges to have simultaneous communication. Communication internal to a processor costs the fixed time $t_i$. We also assume that $t_c \gg t_i$.

We would like to develop tools for reasoning about the relative merits of different contractions. This includes their communication costs and their execution times. To aid in this objective we give the following definitions.

Let $A=(V,E)$ be an algorithm where $V$ is a set of logical processes ( vertices and associated programs ) and $|V|=n$, $E$ is a set of edges $(V_1,V_2)$, $V_1,V_2 \in V$.

Let $M(A,p)=B$ be a contraction of algorithm $A$ into algorithm $B$ where $B$ uses $p$ processors and $p < |V_A|$. The contraction $M$ maps elements of $V_A$ onto $V_B$ such that the number of elements of $V_A$ mapped to an arbitrary element of $V_B$ differs by no more one from the number of elements of $V_A$ mapped to any other element of $V_B$.

Let $w(e)$, the weight of $e$, for $e=(V_1,V_2)$, be the larger of the number of messages from $V_1$ to $V_2$ and the number of messages from $V_2$ to $V_1$.

Let $K(A) = \underset{e}{MAX}\, w(e)$, for $e \in E$, be the communication "cost" of A. This cost is an estimate of the minimum communication time required for the algorithm. Due to dependancies, the actual communication cost may be more.

Let $T(A)$ be the execution time for $A$.

**PROPOSITION 1:** For a given $A$, $p$, $M_1$, and $M_2$, and $t_c > t_i$, if $K(M_1(A,p)) < K(M_2(A,p))$ then $T(M_1(A,p)) \le T(M_2(A,p))$.

This proposition is formalizing the notion that the bottleneck edge will be a lower bound on the time required for the execution of the mapped algorithm. If the processors have a small amount of computation relative to the communication, the execution time will depend on the communication time. The bottleneck edge of the contraction $M_1$ will require a minimum of $t_c K(M_1(A,p))$ time, which is less than $t_c K(M_2(A,p))$. With a higher minimum communication time, we can not expect $M_2$ to execute in less time than $M_1$. If the processors have a large amount of computation in ratio to the communication, the computation time will dominate, yielding near equal times. Even in this case, $M_1$ uses less time for communication than $M_2$. This proposition then motivates us to map the busiest edges of an algorithm to internal edges.

## Case Studies

We now look at several parallel algorithms and some contractions. We approach these by considering algorithms with similar communication graphs. The three graphs considered are the tree, grid, and binary n-cube.

### Tree algorithms

There are several algorithms that run on complete binary trees (Figure 1) having similar characteristics, like the aggregation operations of minimum and global sum. All processors have a value and we want to compute a global value that depends on all these values. Leaf processors send their value to their parents. Internal processors take the minimum (sums) of their own value and their children's values and then send the result to their parents. The final value will be computed at the root processor in $O(\log n)$ time. The communication in these algorithms requires one message over each edge for each global minimum (sum). For a single minimum we have $K(minimum) = 1$.

Consider the contraction in Figure 2a. Let us call this contraction $M_1(minimum,p)$. Each edge in the original algorithm requires one message. Each edge in the smaller graph has 4 edges from the original graph. Since we have only one connection between the physical processors, we have 4 messages for each edge. For an arbitrary $n$ (size of original algorithm) and $p$ (the number of processors) we have $K(M_1(minimum,p)) = \frac{n}{p}$.

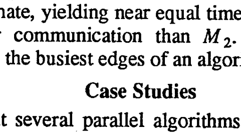


Figure 3: Berman and Snyder tree contraction.

A similar contraction to Figure 2a is touched on by Berman and Snyder[4]. Figure 3 shows this contraction. This is achieved by "folding" the tree. As Berman and Snyder notice, this contraction, $M_2$, has $K(M_2(minimum,p)) = \frac{n}{p}$.

Consider the contraction in Figure 2b. Let us call this contraction $M_3(minimum,p)$. We note that each edge in the smaller graph has at most one edge from the original graph in each direction. For an arbitrary $n$ and $p$ we have $K(M_3(minimum,p)) = 1$.

Proposition 1 tells us that since $M_3$ has a smaller $K$, it is the preferrable contraction. Both $M_1$ and $M_2$ depend on $n$ and $p$ for their cost. But, $M_3$ has a constant cost, regardless of $n$ and $p$. In fact, this contraction is optimum for all tree algorithms that have identical edge weights and unidirectional communication (all toward the root or all toward the leaves).

We first look for a lower bound. Since the tree is connected, the physical processors must be connected. This requires at least one incident edge for each physical processor. The smallest cost $K(M(A,p))$ would be where a maximum of one logical edge was mapped to a physical edge. Therefore, $K(M(A,p)) \ge K(A)$, the cost of the original algorithm.

**LEMMA 2:** For complete binary tree algorithms with balanced processor loads, equal edge weights, and unidirectional communication, algorithm contraction based on Leiserson's binary tree layout technique yields optimum results.

**PROOF:** For the mapping $M_3(A,p)$, each processor contains a complete subtree and an "extra" node. The extra nodes are used in the tree above the subtrees contained in the processors. Therefore, there at most 4 external connections. Of these four, two edges are used to receive(send) data from(to) the children of the extra node, and two edges are used to send(receive) data to(from) the subtree's and the extra node's parents. Since the root of the subtree and the extra node are not at the same level in the tree, edges with data flowing in the same direction can not be connected to the same physical processor. (It is possible to have two of these edges over the same physical edge, but the data moves in opposite directions.) This gives the same weight to the physical edges as the original edges. Therefore, $K(M_3(A,p)) = K(A)$, which is the lower bound. □

Notice that this layout technique will place two logical edges in the same physicial edge for some physical edge. For tree algorithms with bidirectional communication, we then get $K(M_3(A,p)) = 2K(A)$.

To help verify these results, the minimum algorithm was programmed using the Poker parallel programming environment[12]. Both $M_1$ and $M_3$ were programmed. Each contraction was timed using 4 and 16 data items per processor with 4 and 16 processors. The results of these timings are given in Table 1. Each "tick" represents a mircosecond on the 64 processor Pringle.

### Grid algorithms

We next look at algorithms that run on a grid interconnection. Consider the matrix product algorithm for the Wavefront Array Processor(WAP)[8]. It uses $n^2$ processors for the $n \times n$ matrix product $AB = C$. The data is fed in along the top $n$ processors and from the left $n$ processors. The matrix $A$ is arranged to enter column by column, starting with the first column. The matrix $B$ is arranged to enter row by row, starting with the first row. (See Figure 4.) All processors execute identical procedures. The result, $c_{ij}$, is initialized to zero. A loop is executed $n$ times that reads an $A$ value from the left and a $B$ value from above, multiplies them together, and adds the result to $c_{ij}$. The $A$ and $B$ values are sent to the right and down, respectively. This causes the upper left processor to be the first processor to start execution. As the data moves into the array, there is a wavefront of executing processors on the cross diagonal. Each edge is used to send all of one row of $A$ or one column of $B$. For the WAP algorithm we have $K(WAP) = n$.

Consider the contraction in Figure 5. Let us call this contraction $M_1(WAP,p)$. This is the contraction done by cutting the graph into $p$ equal size connected subgraphs and assigning one process from each subgraph selected from corresponding positions to a single processor. The

| Minimum: ticks for n (items) on p (processors) | | | | |
|---|---|---|---|---|
| Contraction | 16 on 4 | 64 on 16 | 64 on 4 | 256 on 16 |
| $M_1$ | 11650 | 20568 | 53496 | 105801 |
| $M_3$ | 4356 | 7682 | 8878 | 12067 |

Table 1: Timings of the minimum algorithm.

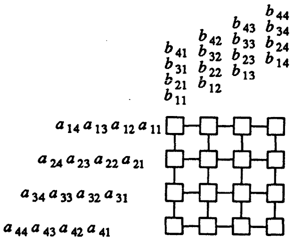

$b_{44}$
$b_{43}$ $b_{34}$
$b_{42}$ $b_{33}$ $b_{24}$
$b_{41}$ $b_{32}$ $b_{23}$ $b_{14}$
$b_{31}$ $b_{22}$ $b_{13}$
$b_{21}$ $b_{12}$
$b_{11}$
$a_{14}\,a_{13}\,a_{12}\,a_{11}$
$a_{24}\,a_{23}\,a_{22}\,a_{21}$
$a_{34}\,a_{33}\,a_{32}\,a_{31}$
$a_{44}\,a_{43}\,a_{42}\,a_{41}$

Figure 4: WAP organization

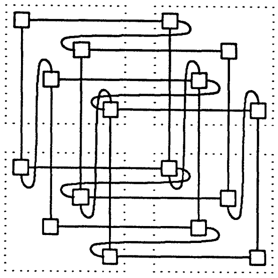


Figure 5: A contraction of 16 logical processes to 4 processors.

physical connection graph, shown in Figure 5, is a grid with end around (i.e. toroidal) connections. For each logical process in a physical processor, there are horizontal and vertical communication paths. Since we have $\frac{n^2}{p}$ logical processes in a processor, the number of logical edges using one processor-to-processor connection is $\frac{n^2}{p}$. Since all horizontal and vertical edges have the same number of messages, $n$, we have $K(M_1(WAP,p)) = \frac{n^3}{p}$.

Consider the contraction in Figure 6. Let us call this contraction $M_2(WAP,p)$. This is the contraction done by cutting the graph into $p$ equal size connected subgraphs and assigning an entire subgraph to a processor. We see that only the perimeter processes have edges that go from processor-to-processor. Also, notice that no end around connections are needed. The number of communication paths over one processor-to-processor connection is $\sqrt{\frac{n^2}{p}}$. Each communication path requires $n$ messages giving $K(M_2(WAP,p)) = \frac{n^2}{\sqrt{p}}$.

Comparing the two contractions, we see that $K(M_2(WAP,p))$ is smaller than $K(M_1(WAP,p))$ by a factor of $\frac{n}{\sqrt{p}}$. Proposition 1 tells us that $M_2$ is the better contraction. We conjecture that $M_2$ is the best contraction that can be achieved for grid algorithms. The basis for this conjecture is that this contraction has the smallest perimeter for a given area, and has been commonly used for contraction in published algorithms, for example for the Jacobi iterative method[1] and for the conjugate gradient method[6].

Both $M_1$ and $M_2$ were programmed using Poker. Table 2 summarizes the results of the timings. As predicted, $M_2$ was the faster contraction, but because the communication time is not the only time consuming part in these algorithms the difference is perhaps not as dramatic as might be seen on a larger problem.

**Binary n-cube algorithms**

We now look at two algorithms for the binary n-cube. The first algorithm is the divide-and-conquer algorithm for matrix product given by Nelson[10]. The other algorithm is Batcher's bitonic sorting algorithm[2].

The matrix product algorithm takes two $n \times n$ matrices, $A$, and $B$, and computes their product $C = AB$. $A$ and $B$ are assumed to be in row major order in the binary n-cube of order $2k$, where $k = \log n$. The algorithm views $A$ and $B$ as a 2x2 matrix of $\frac{n}{2} \times \frac{n}{2}$ matrices. The 2x2 matrix algorithm is then used to multiply the submatrices. Figure 7 shows a order 4 cube layed out in the plane using the CHiP architecture. The numbers in the boxes show the index of the matrix elements initially contained in that processor. We are assuming that the processors are numbered in row major order. The dotted boxes show cubes of order 2. These cubes, which generally have order $2(k-1)$ contain an $\frac{n}{2} \times \frac{n}{2}$ submatrix of both $A$ and $B$. Note that these cubes are constructed by "removing" the edges of order $k$ and $2k$, where and edge of order $k$ connects processors that are $2^{(k-1)}$ distance apart.

To compute the 2x2 matrix product, all processors exchange values of $B$ on the order $2k$ edge and values of $A$ on the order $k$ edge. After the exchange, each cube of order $2(k-1)$ contains 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$. This is all the data that is required for each cube of order $2(k-1)$ to compute its part of the 2x2 matrix product independantly. If the submatrix is not a single element, two matrix products of $\frac{n}{2} \times \frac{n}{2}$ matrices are required. These matrix products are done using the same algorithm. Matrix addition is done element by element. Because corresponding elements of the matrices are contained in the same processor, no communication is required.

To find the cost of this cube matrix multiply, $K(CMM)$, we need to find the edge with the most messages. At the first level of recursion, the order $k$ and $2k$ edges were used to send a message each way. This is the only use of these edges in the algorithm. Therefore, $w(e) = 1$, where $e$ is a order $k$ or $2k$ edge. At the second level of recursion, two matrix products are computed using the order $k-1$ and $2k-1$ edges. Each matrix product sends one message each way on each edge giving $w(e) = 2$, where $e$ is a order $k-1$ or $2k-1$ edge. At level $l$ of the recursion, $w(e) = 2^{l-1}$ messages over the order $k-(l-1)$ and $2k-(l-1)$ edges. The recursion stops when we have order 2 cubes. This is at the $\log n$ level of recursion. There are $\frac{n}{2}$ matrix multiplies done by order 2 cubes. These

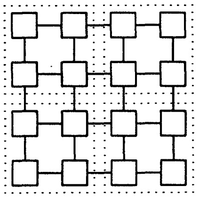


Figure 6: Another contraction of 16 logical processes to 4 processors.

| WAP matrix multiply: ticks for n (items) on p (processors) | | | | |
|---|---|---|---|---|
| Contraction | 16 on 4 | 64 on 16 | 64 on 4 | 256 on 16 |
| $M_1$ | 48854 | 111478 | 400452 | 901543 |
| $M_2$ | 31113 | 73088 | 221545 | 707646 |

Table 2: Timings of the WAP matrix multiply algorithm.

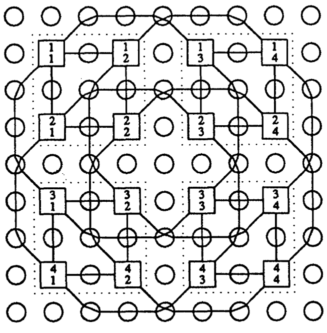


Figure 7: An order 4 binary n-cube.

order 2 cubes use the order 1 and $k+1$ edges. Each matrix multiply sends 1 message each way giving $w(e) = \frac{n}{2}$, where $e$ is a order 1 or $k+1$ edge. Since this is the largest value, $K(CMM) = \frac{n}{2}$.

Consider any contraction, $M(CMM,p)$ where $p = 2^m$ for some $m \leq 2k$. $M(CMM,p)$ will map $\frac{n^2}{p}$ logical processes to every processor. This allows us to put a cube of order $\log \left\lceil \frac{n^2}{p} \right\rceil = 2k-m$ into each processor. The processor-to-processor connection graph is also a cube and is of order $m$. Each processor-to-processor connection supports $\frac{n^2}{p}$ communication paths in the original graph. The real question is which sub-cube do we map to each processor. The cost of the contraction, $K(M(CMM,p))$ will be $\frac{n^2}{p}$ times the maximum $w(e)$, where $e$ is mapped to a physical edge. If $e$ is order 1 or $2k+1$ from the original cube, $K(M(CMM,p)) = \frac{n^3}{2p}$.

Consider the contraction that maps the edges of order 1 through $\left\lfloor \frac{2k-m}{2} \right\rfloor$ and order $k+1$ through $k+\left\lfloor \frac{2k-m}{2} \right\rfloor$ into internal edges. This makes the edge of order $k+\left\lfloor \frac{2k-m}{2} \right\rfloor+1$ the edge with the most messages. This edge is used by level $k-\left\lfloor \frac{2k-m}{2} \right\rfloor$ of the recursion. From before we know that $w(e) = 2^{k-\left\lfloor \frac{2k-m}{2} \right\rfloor-1} = \frac{\sqrt{p}}{2}$. Therefore $K(M(CMM,p)) = \frac{n^2\sqrt{p}}{2p}$. Clearly, this contraction is better in terms of the number of messages over the busiest physical edge than any contraction that does not keep the high traffic logical edges internal to a processor.

By contrast, let us consider the Batcher bitonic merge sort. This sort runs on a order $k$ cube to sort $n = 2^k$ elements. The final sorting will have the smallest element in the first processor and the largest element in the last processor. Figure 8 shows a graphical representation of the algorithm. The arrows represent a data exchange and a compare, leaving the larger number at the end with the arrow and the smaller at the other end. It is obvious from the figure that the order 1 edge has the most messages. Therefore, $K(SORT) = \log n$.

Again, to contract this algorithm, we see that we want to assign a sub-cube into a processor. Consider the contraction $M(SORT,p)$ where the edges of order 1 through order $\log p$ are mapped to internal edges. We are assuming that $p = 2^m$, for some $m \leq \log n$. This contraction assigns the busiest logical edges to be internal edges. These edges carry $\log n - \log p$ messages. Since each processor contains $\frac{n}{p}$ logical processors, $K(M(SORT,p)) = \frac{n(\log n - \log p)}{p}$. Any contraction that does not map these first $\log p$ edges to internal edges will have a higher communication cost. These results agree with and explain the results of Hsiao[7], even though his final algorithm was embedded in a grid instead of another cube.

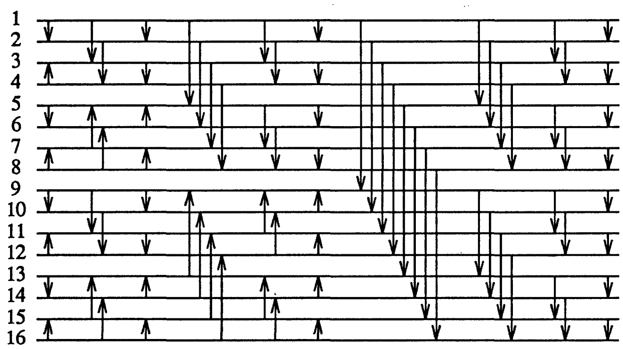


Figure 8: Batcher's bitonic merge sort.

In comparing the contractions for matrix multiply and Batcher's sort, we see that the same size cube is mapped in a different way when mapped to the same number of processors. The busiest edges are different for the two algorithms, thus, the contractions are different.

## Conclusion

The algorithm contraction problem is an important problem for parallel programmers. The way in which an algorithm is contracted can have a significant affect on performance. Processor-to-processor communication can be used as a lower bound on the execution time for an algorithm. It is the processor-to-processor communication that is affected by different contractions.

We have looked at algorithms for the tree, grid, and binary n-cube interconnections. For each algorithm we have compared possible contractions of these algorithms. For trees, we proved that Leiserson's layout technique was the best for contracting tree algorithms such as minimum and sums. For grid algorithms, we conjectured that coalescing by maximizing the area for a given perimeter is optimal for the algorithms with balanced edge loadings. Finally, we showed two algorithms for binary n-cubes that required different contractions to produce the optimal results for the algorithm.

## References

[1] L.M. Adams, *Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers*, Ph.D. Thesis, University of Virginia, Charlottesville, November 1982.

[2] K.E. Batcher, "Sorting Networks and their Applications", *Proceedings of the AFIPS Sprint Joint Computer Confrence*, Vol 32, 1968, pp. 307-314.

[3] F. Berman, M. Goodrich, C. Koelbel, W.J. Robison III, K. Showell, "Prep-P: A Mapping Preprocessor for CHiP Architectures", *Proceedings of the 1985 International Confrence on Parallel Processing*, pp. 731-733.

[4] F. Berman, L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures", *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 307-309.

[5] S.H. Bokhari, "On the Mapping Problem, *IEEE Transactions on Computers*, C-30, No. 3, March 1981, pp. 207-214.

[6] D. Gannon, L. Snyder, J. Van Rosendale, "Programming Substructure Computations for Elliptic Problems on the CHiP System", Ahmed K. Noor (ed.), *Impact of New Computing Systems of Computational Mechanics*, The American Society of Mechanical Engineers, 1983, pp. 65-80.

[7] C.C. Hsiao, *Highly Parallel Processing of Relational Databases*, Ph.D. Thesis, Purdue University, Department of Computer Sciences, December 1982.

[8] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, D.B. Bhaskr Rao, "Wavefront Array Processor: Language, Architecture, and Applications", *IEEE Transactions on Computers*, C-31, No. 11, November 1982, pp. 1-54-1065

[9] C.E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts, 1983.

[10] P.A. Nelson, *A Non-systolic Matrix Product Algorithm*, University of Washington, Department of Computer Science, Technical Report No. 85-11-02, November 1985.

[11] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer", *Computer*, 15(1), January 1982, pp. 47-56.

[12] L. Snyder, "Parallel Programming and the Poker Programming Environment", *Computer*, 17(7), July 1984, pp. 27-36.

[13] L. Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential", *Annual Review of Computer Science*, 1986 (to appear).

# PARALLEL ALGORITHMS FOR 2–D CONVOLUTION

*Zhixi Fang and Xiaobo Li*

Department of Computer Science
Wichita State University
Wichita, KS 67208

*Lionel M. Ni*†

Department of Computer Science
Michigan State University
East Lansing, MI 48824

**Abstract:** Convolution is a basic operation of linear systems theory, and 2-D convolution is a frequently used operation in image processing. However, 2-D convolution is computationally intensive. For an $N$ by $N$ search area and an $M$ and $M$ window, the time complexity of a serial algorithm is $O(N^2M^2)$. This paper presents several parallel convolution algorithms for array processors with $N^2$ processing elements connected by various communication networks. By using inter-PE communication networks efficiently, each PE requires only a small local memory, many unnecessary data transmissions are eliminated, and the time complexity is reduced to $O(M^2)$.

## I. INTRODUCTION

Great efforts have been devoted to developing parallel processing architectures and associated algorithms for many popular mathematical operations [DuLe81, FuIc82, HwFu83, LiNi85, NiHw85, NiJa85]. *Convolution* is a very important operation in linear systems theory and image processing. The convolution of two functions **G** and **W** is a function $C$ of a displacement $y$ defined as

$$C(y) = \int \mathbf{G}(x)\mathbf{W}(x-y)dx.$$

We generalize it to the following form:

$$C(y) = \Omega[\mathbf{G}(x)\bullet\mathbf{W}(x-y)],$$

where $\bullet$ is a binary operator, which could be a multiplication (*), addition (+), inf, logical AND, etc.; and $\Omega$ is a function defined on all values of $[\mathbf{G}(x)\bullet\mathbf{W}(x-y)]$ over the entire region of $x$. $\Omega$ could be an integration (continuous case), summation (discrete case), sup, or maximum.

In two dimensions, one function is "rubbed over" the other [Ball82]. The value of the convolution at any displacement can be defined as the following general form:

$$C(i,j) = \Psi\Phi[\mathbf{G}(i+s,j+t)\bullet\mathbf{W}(s,t)]$$

where $\Phi$ is a function defined over the region of $t$, and $\Psi$ is a function defined over the region of $s$. In the finite discrete case, **G** is defined on a square grid array of size $N^2$ as

$$\mathbf{G} = \left\{ G_{ij} | i,j \in [0,N-1] \right\},$$

and **W** is defined on a finite rectangular grid of size $M$ by $M'$ $(M,M' < N)$,

$$\mathbf{W} = \left\{ W_{st} | s \in [0,M-1], t \in [0,M'-1] \right\}.$$

Hence, the 2-D digital convolution of **G** and **W** can be defined as:

$$C_{ij} = \sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} G_{i+s,j+t} * W_{st}. \tag{1}$$

Equation (1) is also called *image correlation* or *image template matching.*

Many other applications, such as *object labeling* including fuzzy model, discrete model, and linear stochastic model and *morphological operations* including dilation and erosion, also require some forms of 2-D convolution. Their difference lies on the different choices of $\Psi$, $\Phi$, and $\bullet$ functions. For example, the linear stochastic model for object labeling in computer vision requires the computation of the probability that a label $y$ fits object $x$. Let $W_{st}$ denote the probability that label $t$ fits object $s$. Also let $G_{st}$ denote the conditional probability that label $y$ fits object $x$ given that $t$ is the correct label for object $s$. Then the probability that $y$ is the correct label for object $x$ of the $(k+1)th$ iteration is

$$C(i,j) = \Psi\Phi[\mathbf{G}(i+s,j+t)\bullet\mathbf{W}(s,t)],$$

where $i=j=0$, $\bullet$ defines a multiplication, $\Phi$ defines a summation over the range of $t$, and $\Psi$ defines a weighted summation over $s$ with weighting factors $\beta_{xs}$ for different values of $s$. That is,

$$\mathbf{W}^{(k+1)}(x,y) = \sum_s \beta_{xs} [\sum_t [\mathbf{G}(x,y;s,t) * \mathbf{W}^{(k)}(s,t)]],$$

or in a more familiar form [RoHB76]:

$$P_x^{(k+1)}(\lambda_y) = \sum_s c_{xs} [\sum_t [P_{xs}(\lambda_y|\lambda_t)P_s^{(k)}(\lambda_t)]].$$

We are concerned with the communication complexity of 2-D convolutions. Various definitions of $\Psi$, $\Phi$, and $\bullet$ only affect the computational complexity. Equation (1) will be used as a representative case to indicate the computational requirement for a wide variety of window-based image processing tasks. **G** is called the *search area* and **W** is called a *window.* 2-D digital convolution is used to find *sub-search-areas* (*subareas*) from **G** that match closely with **W**. For convenience, each $M$ by $M'$ subarea of **G** can be uniquely referenced by its upper left corner coordinate $(i,j)$. Let $SUB(i,j)$ denote the subarea at location $(i,j)$. There are $(N-M+1)(N-M'+1)$ such subareas for $i \in [0,N-M]$ and $j \in [0,N-M']$ as shown in Fig.1.

The window **W** is matched against every subarea $SUB(i,j)$ in **G**; and the convolution $C_{ij}$ is computed as the *proximity measure* between **W** and each subarea $SUB(i,j)$. This measure is also called *non-normalized correlation*, or cross correlation. The *normalized correlation* is discussed in Section VII. To simplify the discussion, we assume that $M=M'$ and $M$ is a power of 2. We also assume the cross correlation $C_{ij}$ is used. In Section IV, we discuss the general case for array processors where $M \neq M'$ and $M$ is not necessary to be power of 2.

During the convolution computation of all displacements, $(N-M+1)(N-M'+1)$ similarity values are generated, and one of

† This work was supported in part by the DARPA NIMM project

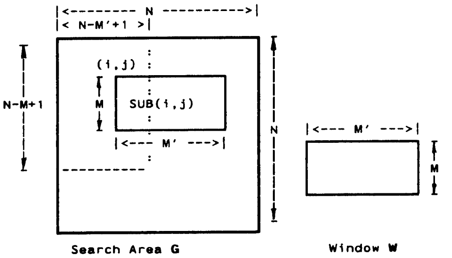


Fig. 1. An $N$ by $N$ search area **G** and an $M$ by $M'$ window **W**

the following results can be obtained:

1. another 2-D array, formed by the similarity values of all subarea locations;
2. all subareas with a similarity which exceeds a given threshold;
3. the subarea with the greatest similarity to a window; and
4. the first subarea whose similarity exceeds a predefined threshold.

Convolution can be used as a simple ***filtering*** method, for which option 1 above is desired. For ***edge detection***, the edge operator is a window and option 2 is produced [FrCh77]. ***Image registration***, or ***scene matching***, is another application of convolution where option 3 or 4 is used [WoHa78].

Our paper discusses parallel algorithms for 2-D convolution, as defined in Eq.(1), on array processors with three different types of interconnection networks. Usually, 2-D convolution operations are extremely time-consuming, involving the time required both to perform the calculations and for communication among the PEs. With a single processor and traditional algorithms, the number of steps for convolution operations is $(N-M+1)(N-M'+1)MM'$, approximately $N^2MM'$ for $M, M' << N$. With multiple processors and parallel processing algorithms, the total time can be significantly reduced. These algorithms can be easily modified for other applications of convolution, such as those listed in Table 1.

Different convolution applications have different computational complexities in their evaluation of different functions $\Phi$, $\Psi$ and $\bullet$. For each application, however, the time complexity of the computation performed in the PEs is exactly the same, regardless of the type of network. It is the time complexity for communication among the PEs that varies widely from network to network. We will compare the communication complexities of three different types of networks, independent of the forms of the functions $\Phi$, $\Psi$ and $\bullet$.

In this paper, parallel algorithms are developed and discussed for 2-D digital convolution on array processors with different types of interconnection networks. Section II introduces a model of an array processor and the three types of interconnection networks. Detailed algorithms for array processors with $N^2$ PEs with a mesh network, a hypercube network, and a shuffle-exchange network are given in Sections III, IV and V, respectively. Section VI provides an explanation of the modifications required to the algorithms for different size of array processors. Section VII discusses the computation of generalized convolution. Section VIII presents our conclusions, including a table comparing the network communication complexities.

## II. MODEL OF AN ARRAY PROCESSOR

An array processor is comprised of $Q=2^q$ processing elements (PEs), each having some local memory [HwBr84]. We assume that the PEs are indexed 0 through $Q-1$ and refer to the $p$th PE as PE($p$). The synchronized PEs execute instructions issued from a control unit. The control unit broadcasts an instruction to all PEs, and all enabled PEs simultaneously execute. the instruction. The enable/disable mask can be used to select a subset of the PEs that are to perform an instruction. The set of enabled PEs can be changed from instruction to instruction.

A network is needed to provide inter-PE communication. While several types of interconnection networks have been proposed [Sieg79], some are topologically equivalent and some are not [WuFe80]. The control transfer algorithm among the equivalent networks has been studied in [Fang84,FaDe85]. Three interconnection networks which are not topologically equivalent are considered in this paper -- the mesh, hypercube, and shuffle-exchange networks.

### Mesh Interconnection Network

In this model, the PEs may be thought of as being physically arranged in a k-dimensional array $A(n_{k-1}, n_{k-2}, \cdots, n_0)$, where $n_i$ is the size of the $i$th dimension and $Q=n_{k-1}*n_{k-2}*\ldots*n_0$. The PE at location $A(p_{k-1},\ldots,p_0)$ is connected to the PEs at locations $A(p_{k-1},\ldots,p_j\pm 1,\ldots,p_0)$, $0\leq j<k$, provided they exist. Data may be transmitted from one PE to another via this interconnection pattern only. The interconnection scheme for 16 PEs with $k=2$ is given in Fig. 2a.

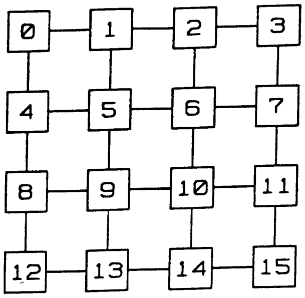


Fig. 2a. A mesh topology with 16 PEs

### Hypercube Interconnection Network

Let $p_{q-1}\ldots p_0$ be the binary representation of $p$, the index of PE($p$), for $p=0,1,\ldots,2^q-1$. Let $p^{(b)}$ be the number with binary representation $p_{q-1}\cdots p_{b+1}\overline{p}_b\, p_{b-1}\cdots p_0$, where $\overline{p}_b$ is the complement of $p_b$ and $0\leq b<q$. In the hypercube model, PE($p$) is directly connected to PE($p^{(b)}$) for $0\leq b<q$. A 4-cube (16 PEs) interconnection network is shown in Fig. 2b. The hypercube has been proposed and used in both SIMD and MIMD computer systems [PrVu79,Seit85].

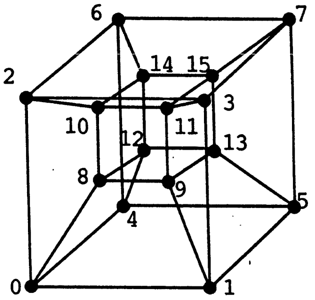


Fig. 2b. A hypercube topology with 16 PEs

### Shuffle-Exchange Interconnection Network

Let $q$, $p$ and $p^{(b)}$ be the same as in the hypercube model. Let $p_{q-1} \cdots p_0$ be the binary representation of $p$. Define SHUFFLE($p$) and UNSHUFFLE($p$) to be the integers with binary representation $p_{q-2}p_{q-3} \cdots p_0 p_{q-1}$ and $p_0 p_{q-1} \cdots p_1$, respectively. In the shuffle-exchange model, PE($p$) is connected to PE($p^{(0)}$), PE(SHUFFLE($p$)) and PE(UNSHUFFLE($p$)). These three connections are called *exchange, shuffle* and *unshuffle*, respectively. Once again, data transmission from one PE to another is possible only via the connection scheme. A shuffle-exchange network of 16 PEs is shown in Fig. 2c.

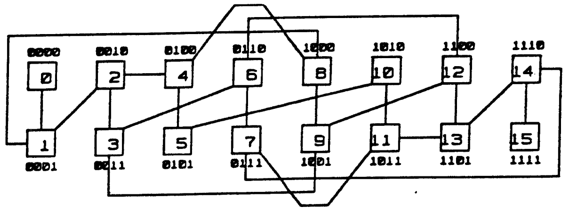


Fig. 2c. A shuffle-exchange topology with 16 PES

It should be noted that the shuffle-exchange model requires at most three connections per PE, while the mesh model requires $2k$ connections per PE, and the hypercube model requires $q$ connections per PE. It should also be emphasized that, in any time instance, only one unit of data can be transmitted along an interconnection line, although all lines can be busy at the same time.

To evaluate the efficiency of parallel algorithms on array processors, one must consider the amount of parallellism that can be exploited in order to fully utilize the large amount of PEs. Such a study on image correlation was conducted in [SiSF82]. In their approach, each PE has a large local memory to handle a number of subimages, and only neighboring pixels need be sent among PEs. In our approach, we use the interconnection networks efficiently to obtain an optimal solution. Each local memory is small, containing only $M$ locations.

In describing our algorithms, we follow the notations and assumptions used in [DeNS81] as stated below. PE($p$) denotes the PE with index $p$. $R(p)$, $A(p)$, $B(p)$, and $C(p)$ are registers in PE($p$). MAR($p$) denotes the local memory address register in PE($p$), and M[MAR($p$)] denotes the local memory location with address in MAR($p$). The symbol "$\leftarrow$" signals an assignment involving data routing between directly connected PEs, while the symbol ":=" indicates an assignment in which all variables in both sides of ":=" are local to the same PE. "$\Leftarrow$" indicates movement involving common data or constants from the control unit memory to the local memories whose address is specified in MAR($p$). For any integer $i$, $i_b$ will denote bit $b$ of the binary representation of $i$, and $i_{s:r}$ will signify the number whose binary representation is $i_s i_{s-1} \dots i_r$. PEs may be enabled by providing a *selectivity function* following a statement. For example, if we want to perform $R(p) := A(p) + B(p)$ for those PEs whose index has bit $b$ equal to 0, the statement will be

$$R(p) := A(p) + B(p),\ (p_b = 0).$$

The communication complexity of an algorithm includes both the time needed to route data from PE to PE, and the time needed to broadcast data from control unit memory to local memories. A *unit-route* is a data transmission from a PE to a directly connected PE. The following unit-route statements will be used in the three interconnection networks in this paper:

(1) In the 2-dimensional mesh model:
$R(p_1-1,p_2) \leftarrow R(p_1,p_2)$ (* go up *);
$R(p_1+1,p_2) \leftarrow R(p_1,p_2)$ (* go down *);
$R(p_1,p_2-1) \leftarrow R(p_1,p_2)$ (* go left *);
$R(p_1,p_2+1) \leftarrow R(p_1,p_2)$ (* go right *).

(2) In the hypercube model:
$R(p^{(b)}) \leftarrow R(p)$ where $0 \leq b < q$.

(3) In the shuffle-exchange model:
$R(p^{(0)}) \leftarrow R(p)$ (* go exchange connection *);
$R(SHUFFLE(p)) \leftarrow R(p)$ (* go shuffle connection *);
$R(UNSHUFFLE(p)) \leftarrow R(p)$ (* go unshuffle connection *).

The above notations will be frequently used in the algorithms in the upcoming sections. The following procedure ROTATE will be extensively used and is demonstrated below. Consider a 2-D search area, $\mathbf{G} = \{G_{ij},\ \text{for } i,j \in [0,N-1]\}$, and an array processor with $Q = 2^q = N^2$ PEs, where the element $G_{ij}$ is stored in register $R(p)$ of PE($p = iN + j$) and $N = 2^n$. For convenience, we may use PE($i,j$) or $R(i,j)$ to denote PE($iN + j$) or $R(iN + j)$. Thus, initially, we have $R(i,j) = G_{ij}$. We may wish to *rotate*, or *end-around shift*, the 2-D search area $2^k$ positions to either

right $(R(i,j) = G_{i,(j-2^k) \bmod N})$,
left $(R(i,j) = G_{i,(j+2^k) \bmod N})$,
up $(R(i,j) = G_{(i+2^k) \bmod N,j})$,
or down $(R(i,j) = G_{(i-2^k) \bmod N,j})$,

for $0 \leq k < n-1$. The rotation procedure for a 2-D mesh machine is straightforward. The procedure to perform a rotate operation in a hypercube array processor is shown below. The rotate procedure for a shuffle-exchange network is explained in Section V. In procedure ROTATE, $s$ indicates the bit position where the bit complement operation begins; $r$ is the bit position where the bit complement ends. Flag is set to 0 for left and up rotate operations; it is set to 1 for right and down rotate operations.

```
procedure ROTATE($R,s,r$,flag);
begin
  $R(p^{(s)}) \leftarrow R(p)$;
  for $b := s+1$ to $r$ do
    if flag=0 then
      $R(p^{(b)}) \leftarrow R(p),\ (p_{b-1}=1\ \&\ \cdots\ \&\ p_s=1)$
    else (* Note that & is a logical AND operator *)
      $R(p^{(b)}) \leftarrow R(p),\ (p_{b-1}=0\ \&\ \cdots\ \&\ p_s=0)$;
end;
```

In the following sections, we shall present detailed convolution algorithms for array processors with three different types of interconnection networks.

## III. MESH MODEL WITH $N^2$ PEs

In this section an $O(M^2)$ convolution algorithm MESH-N2 is developed for a two-dimensional mesh model with $N^2$ PEs, which can handle various sizes of windows. These $N^2$ PEs may be viewed as an $N$ by $N$ 2-D array. We assume that $N=2^n$ and $M=2^m$. Initially, the element $G_{ij}$ is stored in register $A(i,j)$ of PE$(i,j)$ for $i,j \in [0,N-1]$. The $M$ by $M$ window is stored in the control unit memory in column-major order with a starting address $\alpha$. The resulting similarity measures of $C_{ij}$ will be stored in register $C(i,j)$ of each PE.

This algorithm is designed to process one column of the window at a time. For a fixed column $t$, $M$ window elements, $W_{0t}, \cdots, W_{m-1,t}$ will be broadcast to all PEs. Thus, $M$ local memory locations are needed in each PE to hold these elements. If we rotate up the search area **G** one row per step, we can compute a partial sum of $M$ products for a given $t$. Therefore, PE$(i,j)$, which is responsible for evaluating $C_{ij}$, will accumulate the following $M$ products.

$$\sum_{s=0}^{M-1} G_{i+s,j+t} * W_{st} \qquad (2)$$

MESH-N2 matches an $M$ by $M$ window in an $N$ by $N$ search area on a mesh machine with $N^2$ PEs. The initial configuration is $A(i,j) = G_{ij}$.

```
procedure MESH-N2 ($A$, $C$)
begin
  $C(i,j)$ := 0; (*initialization*)
  for $t$ :=0 to $M$-1 do begin
    (*read one window column to local memory*)
    MAR($i,j$) := 0; (*initialize MAR*)
    for $s$ :=0 to $M$-1 do begin
      CMAR := $\alpha+t*M+s$;
      M[MAR($i,j$)] ⇐ CM[CMAR];
      MAR($i,j$) := MAR($i,j$) + 1;
    end; (*end loop $s$ *)
    (*Rotate-multiply-add on the search area segment*)
    MAR($i,j$) := 0;
    $C(i,j)$ := $C(i,j)$+$A(i,j)$*M[MAR($i,j$)];
    (*get first product*)
    for $k$ :=1 to $M$-1 do begin
      $A(i-1,j) \leftarrow A(i,j)$;
      MAR($i,j$) := MAR($i,j$) + 1;
      $C(i,j)$ := $C(i,j)$ + $A(i,j)$*M[MAR($i,j$)];
    end; (*partial summation of $M$ products*)
    for $k$ :=1 to $M$-1 do $A(i+1,j) \leftarrow A(i,j)$;
      (*recover the original value*)
    $A(i,j-1) \leftarrow A(i,j)$;
      (* rotate G left one position *)
  end; (*end loop $t$, process one window column*)
end;
```

Clearly, the communication time complexity of the MESH-N2 algorithm is

$$\sum_{t=0}^{M-1} 3M = 3M^2 = O(M^2)$$

It is trivial to extend MESH-N2 to the general case where either $M \neq M'$ or $M$ is not a power of 2.

## IV. HYPERCUBE MODEL WITH $N^2$ PEs

Now we turn to CUBE-N2, the convolution algorithm for a hypercube array processor with $N^2$ PEs. As with the mesh model, the $M$ by $M$ window is stored in the control unit memory in column-major order with a starting address $\alpha$; and each PE has $M$ locations in the local memory. We logically divide each column of the search area **G** into $N/M$ segments, where each segment has $M$ consecutive elements.

There are four phases involved in order to compute Eq.(2) for a given window column. In phase 1, all $M$ elements of the window column are sent to the local memory of all PEs. These $M$ elements are located in addresses from 0 to $M-1$. In phase 2, we permute the elements within each segment. Then, we multiply the element in register $A$ with the window element, which is stored in local memory. Finally, we add the product to register $C$. These permute-multiply-accumulate (PMA) steps are repeated so that every element of the segment appears exactly once in each position of the segment. Since we only permute elements within the same segment, some positions which need elements from the next segment cannot be satisfied. For instance, the last position of a segment needs $M-1$ elements of **G** from the segment below it. Thus, only half of the products are generated in phase 2 (this will be clear from the later example and discussion).

In phase 3, we first rotate **G** up $M$ positions. Then, we repeat the PMA steps in the same way as in phase 2 in order to generate the remaining half of the products needed in Eq.(2). Before we go to the next phase, we have to rotate **G** down $M$ positions to recover it to the original status. In phase 4, we rotate **G** left one position so that we can start to work on the next window column. After we have processed all $M$ window columns, the final result of $C_{ij}$ is stored in $C(i,j)$.

To simplify our description of the PMA steps, we omit the second subscripts of $j$ and $t$ for **G** and **W**. We use $G_0, G_1, \cdots, G_{m-1}$ to denote the $M$ elements of one segment. Initially, we have $A(i)=G_i$ and $C(i)=0$. After the execution of $C(i)=C(i)+A(i)$*M[MAR($i$)], we have $G_i * W_0$ in register $C$ of every PE.

Now we will discuss the PMA sequence. First, we exchange one element in each segment so that $A(i)=G_j$, where $j=i^{(0)}$. We assign MAR($i$)=1. Only the PE($i$) with $i_0=0$ can perform the multiplication to generate $G_{i+1} * W_1$.

In the second step, we exchange pairs of elements within each segment. If there is a valid local memory address (to be discussed later), only PE($i$) with $i_1=0$ can multiply $A(i)$*M[MAR($i$)]). The product will be added to $C(i)$. Since the second step should generate two terms in Eq.(2), $G_{i+2} * W_2$ and $G_{i+3} * W_3$, in PE($i$) with $i_1=0$, a procedure GRAY (to be defined later) is employed to exchange the elements in each pair within the segment. In the same way, the $(k+1)$th step exchanges $2^k$ element pieces within each segment. Using a valid address, we perform

$$C(i) := C(i) + A(i) * \text{M[MAR}(i)], \quad (i_k = 0).$$

Just as in the second step, the $(k+1)$th step generates $2^k$ terms in Eq.(2), and procedure GRAY executes a permutation subsequence. After executing the $m-1$ steps, each element in a segment should appear in any position of the segment once and only once.

We employ the GRAY sequence concept [Roth79] to generate the permutation subsequence for the small section of $2^k$ elements in each step. Given an integer $i$ in the range $[0,M-1]$, where $i=i_{m-1:0}$, and a given $k$ $(0 \le k < m-1)$, starting with integer $i$, we can change one bit of the binary expression at a time to generate a sequence of $2^k-1$ integers in the range $\{\lfloor i/2^k \rfloor * 2^k, \ldots, (\lfloor i/2^k \rfloor + 1) * 2^k - 1\}$ with $i$ excluded. Let $S_{k-1}$ denote the sequence of bit positions on which the GRAY sequence with $2^k-1$ integers will take a complement operation to generate the corresponding integers. $S_{k-1}$ can be recursively defined as follows:

$$S_0 = 0$$

$$S_{k-1} = S_{k-2},\ k-1,\ S_{k-2}.$$

The length of the GRAY sequence can be obtained by the recursive equation $L(S_{k-1})=2L(S_{k-2})+1$, where $L(S_0)=1$. The following lemmas are useful in developing our parallel algorithms and can be easily proved by induction on $k$.

**Lemma 1**: Given an integer $i=i_{m-1:k}\,i_{k-1}i_{k-2:0}$ in $[0,M-1]$, starting from $i$, the last number in the GRAY sequence of $2^k-1$ numbers is $i^{(k-1)}=i_{m-1:k}\,\bar{i}_{k-1}i_{k-2:0}$.

Let $F$ be a boolean variable, which is changed alternatively between true and false in the generation of numbers of a GRAY sequence. The initial value of $F$ is false. In the sequence of bit positions $S_{k-1}$, each bit position $b$ $(0 \le b < k-1)$ is involved in pairs of complement operations. In each pair, the first complement is from $i_b$ to $\bar{i}_b$ when $F$ is true. The second complement is from $i_b$ to $\bar{i}_b$ when $F$ is false.

**Lemma 2**: During the GRAY sequence, when $i_b$ is to be complemented to generate the next number, MAR($i$) is modified by the following formula.

$$\text{MAR}(i) := \text{MAR}(i)+2^b \text{ if } (F \& i_b=0) | (\bar{F} \& i_b=1)$$
$$\text{MAR}(i) := \text{MAR}(i)-2^b \text{ if } (F \& i_b=1) | (\bar{F} \& i_b=0)$$

**Lemma 3**: After executing $A(i^{(k)}) \leftarrow A(i)$, the MAR($i$) is incremented by $2^k$ if $i_{k-1}=0$. If $i_{k-1}=1$, then MAR($i$) is unchanged.

The recursive procedure GRAY($U$,$F$,phase) is designed to generate the GRAY sequence, modify MAR($i$), and execute the multiply-accumulate operation on $C(i)$.

```
recursive procedure GRAY($U$,$F$,phase)
begin
  if $U=0$ then return else
  begin
    flag:=true;
    GRAY($U-1$,flag,phase);
    $A(p^{(U+n-1)}) \leftarrow A(p)$;
    if $F$ then
      MAR($p$):=MAR($p$)+$2^{U-1}$, $(p_{U+n-1}=0)$;
      MAR($p$):=MAR($p$)-$2^{U-1}$, $(p_{U+n-1}=1)$;
    else
      MAR($p$):=MAR($p$)-$2^{U-1}$, $(p_{U+n-1}=0)$;
      MAR($p$):=MAR($p$)+$2^{U-1}$, $(p_{U+n-1}=1)$;
    end; (* end if-else *)
    if phase=2 then $C(p):=C(p)+A(p)*$M[MAR($p$)], $(p_k=0)$
      else $C(p):=C(p)+A(p)*$M[MAR($p$)], $(p_k=1)$;
    flag:=false;
    GRAY($U-1$,flag,phase);
  end; (* end if-else *)
end;
```

Finally we come to procedure CUBE-N2, a convolution algorithm for a hypercube array processor with $N^2$ PEs.

```
procedure CUBE-N2($A$, $C$)
begin
  $C(p)$:=0; (* initialization *)
  (* process one window column at a time *)
  for $t$:=0 to $M-1$ do begin
    (* Phase 1: read one window column to local memory *)
    MAR($p$):=0; (* initialize MAR *)
    for $s$:=0 to $M-1$ do begin
      CMAR := $\alpha+t*M+s$;
      M[MAR($p$)] $\Leftarrow$CM[CMAR];
      MAR($p$) := MAR($p$)+1;
    end;
    (* Phase 2: PMA on its own segment *)
    MAR($p$):=0;
    $C(p):=C(p)+A(p)*$M[MAR($p$)]; (* get first product *)
    for $k$:=$n$ to $n+m-1$ do begin
      $A(p^{(k)}) \leftarrow A(p)$;
      if $k=n$ then MAR($p$):=MAR($p$)+1
        else MAR($p$):=MAR($p$)+$2^{k-n}$, $(p_{k-1}=0)$;
      $C(p):=C(p)+A(p)*$M[MAR($p$)], $(p_k=0)$;
      $F$:=false;
      $U$:=$k-n$;
      GRAY($U$,$F$,2);
    end (* end loop $k$ *)
    (* Phase 3: PMA on the next segment *)
    (* rotate the image up one segment *)
    $A(p^{(n+m-1)}) \leftarrow A(p)$; (* recover original value *)
    ROTATE($A$,$n+m$,$2n-1$,0);
    (* PMA on the new segment *)
    MAR($p$):=$M$;
    for $k$:=$n$ to $n+m-1$ do begin
      $A(p^{(k)}) \leftarrow A(p)$;
      if $k=n$ then MAR($p$):=MAR($p$)-1
        else MAR($p$):=MAR($p$)-$2^{k-n}$, $(p_{k-1}=1)$;
      $C(p):=C(p)+A(p)*$M[MAR($p$)], $(p_k=1)$;
      $F$:=false;
      $U$:=$k-n$;
      GRAY($U$,$F$,3);
    end (* end loop $k$ *)
    (* rotate G down one segment *)
    $A(p^{(n+m-1)}) \leftarrow A(p)$; (* recover original value *)
    ROTATE($A$,$n+m$,$2n-1$,1);
    (* Phase 4: Rotate G left one position *)
    ROTATE($A$,0,$n-1$,0);
  end; (* end loop $t$ *)
end;
```

Algorithm CUBE-N2 uses $M$ local memory locations for each PE. The inter-PE communication time in procedure GRAY with parameter $U$ is $2^U-1$ unit routes. Therefore, the communication time in phase 2 or phase 3 requires $\sum_{k=0}^{m-1}(2^k-1) \approx M$ unit-routes. To rotate **G** up or down one segment takes $\log_2(N/M)$ unit-routes. Phase 4 takes $\log_2 N$ unit-routes to rotate **G** left one position. The communication complexity is

$$\sum_{t=0}^{M-1}(3M+2(log_2 N-\log_2 M)+\log_2 N)$$
$$= M(3M+\log_2 N-2\log_2 M) \le M*\max(3M,\log_2 N)=O(M^2).$$

Note that the complexity of the traditional computer is $O(N^2*M^2)$.

|  |  | k=n | k=n+1 | U=1 | k=n+2 | U=1 | U=2 | U=1 | recover | rotate up | k=n | k=n+1 | U=1 | k=n+2 | U=1 | U=2 | U=1 | recover | rotate down |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PE(i) | D,G | D,G | D,G | D,G | D,G | D,G | D,G | D,G | G | D,G | D,G | D,G | D,G | D,G | D,G | D,G | D,G | G | G |
| PE( 0) | *0,0 | *1,1 | *3,3 | *2,2 | *6,6 | *7,7 | *5,5 | *4,4 | 0 | 8,8 | 7,9 | 7,11 | 6,10 | 6,14 | 7,15 | 5,13 | 4,12 | 8 | 0 |
| PE( 1) | *0,1 | 1,0 | *1,2 | *2,3 | *6,7 | *5,6 | *3,4 | *4,5 | 1 | 8,9 | *7,8 | 5,10 | 6,11 | 6,15 | 5,14 | 3,12 | 4,13 | 9 | 1 |
| PE( 2) | *0,2 | *1,3 | 3,1 | 2,0 | *2,4 | *3,5 | *5,7 | *4,6 | 2 | 8,10 | 7,11 | *7,9 | *6,8 | 2,12 | 3,13 | 5,15 | 4,14 | 10 | 2 |
| PE( 3) | *0,3 | 1,2 | 1,0 | 2,1 | *2,5 | *1,4 | *3,6 | *4,7 | 3 | 8,11 | *7,10 | *5,8 | *6,9 | 2,13 | 1,12 | 3,14 | 4,15 | 11 | 3 |
| PE( 4) | *0,4 | *1,5 | *3,7 | *2,6 | 6,2 | 7,3 | 5,1 | 4,0 | 4 | 8,12 | 7,13 | 7,15 | 6,14 | *6,10 | *7,11 | *5,9 | *4,8 | 12 | 4 |
| PE( 5) | *0,5 | 1,4 | *1,6 | *2,7 | 6,3 | 5,2 | 3,0 | 4,1 | 5 | 8,13 | *7,12 | 5,14 | 6,15 | *6,11 | *5,10 | *3,8 | *4,9 | 13 | 5 |
| PE( 6) | *0,6 | *1,7 | 3,5 | 2,4 | 2,0 | 3,1 | 5,3 | 4,2 | 6 | 8,14 | 7,15 | *7,13 | *6,12 | *2,8 | *3,9 | *5,11 | *4,10 | 14 | 6 |
| PE( 7) | *0,7 | 1,6 | 1,4 | 2,5 | 2,1 | 1,0 | 3,2 | 4,3 | 7 | 8,15 | *7,14 | *5,12 | *6,13 | *2,9 | *1,8 | *3,10 | *4,11 | 15 | 7 |
| PE( 8) | *0,8 | *1,9 | *3,11 | *2,10 | *6,14 | *7,15 | *5,13 | *4,12 | 8 | 8,16 | 7,17 | 7,19 | 6,18 | 6,22 | 7,23 | 5,21 | 4,20 | 16 | 8 |
| PE( 9) | *0,9 | 1,8 | *1,10 | *2,11 | *6,15 | *5,14 | *3,12 | *4,13 | 9 | 8,17 | *7,16 | 5,18 | 6,19 | 6,23 | 5,22 | 3,20 | 4,21 | 17 | 9 |
| PE(10) | *0,10 | *1,11 | 3,9 | 2,8 | *2,12 | *3,13 | *5,15 | *4,14 | 10 | 8,18 | 7,19 | *7,17 | *6,16 | 2,20 | 3,21 | 5,23 | 4,22 | 18 | 10 |
| PE(11) | *0,11 | 1,10 | 1,8 | 2,9 | *2,13 | *1,12 | *3,14 | *4,15 | 11 | 8,19 | *7,18 | *5,16 | *6,17 | 2,21 | 1,20 | 3,22 | 4,23 | 19 | 11 |
| PE(12) | *0,12 | *1,13 | *3,15 | *2,14 | 6,10 | 7,11 | 5,9 | 4,8 | 12 | 8,20 | 7,21 | 7,23 | 6,22 | *6,18 | *7,19 | *5,17 | *4,16 | 20 | 12 |
| PE(13) | *0,13 | 1,12 | *1,14 | *2,15 | 6,11 | 5,10 | 3,8 | 4,9 | 13 | 8,21 | *7,20 | 5,22 | 6,23 | *6,19 | *5,18 | *3,16 | *4,17 | 21 | 13 |
| PE(14) | *0,14 | *1,15 | 3,13 | 2,12 | 2,8 | 3,9 | 5,11 | 4,10 | 14 | 8,22 | 7,23 | *7,21 | *6,20 | *2,16 | *3,17 | *5,19 | *4,18 | 22 | 14 |
| PE(15) | *0,15 | 1,14 | 1,12 | 2,13 | 2,9 | 1,8 | 3,10 | 4,11 | 15 | 8,23 | *7,22 | *5,20 | *6,21 | *2,17 | *1,16 | *3,18 | *4,19 | 23 | 15 |
|  | phase 1 | phase 2 |  |  |  |  |  |  |  | phase 3 |  |  |  |  |  |  |  |  |  |

Fig. 3. An example shows the permutation and address generation sequences for 16 PEs, where $M=8$

To further illustrate our algorithm, Fig.3 shows an example of processing one window column where $M=8$. The first column of the figure gives the initial configuration, where we list 16 PEs (PE(0,$j$) to PE(15,$j$)). The first value, $D$, associated with each PE, is the value of its MAR, and the second value, $G$, is the index $i$ of $G_{ij}$. Again, subscripts $j$ and $t$ are omitted from Fig.3. The results of all enable operations are followed by the symbol *. The first column in phase 3 demonstrates the rotation of G up one segment (8 positions). The second to last column in both phase 2 and phase 3 shows the restoration of G to its original value (before permutation). The last column in phase 3 shows the effects of rotating G down one segment. Fig.3 demonstrates that, after phase 3, all eight products are available for the evaluation of Eq.(2).

In the general case, a window could be an $M$ by $M'$ rectangle ($M<M'$), and $M$ might not be a power of 2. Let $M=M_1+M_2+\cdots+M_q$, where $M_d$ is an integer power of 2, and $d=1,2,...,q$. Algorithm CUBE-N2 can be modified by looping $k$ from $n$ to $n+m-1$ inside another loop on $d$ from 1 to $q$. For each value of $d$, the variable $m$ takes the value $\log_2(M_d)$. In the outermost loop, $t$ increments from 0 to $M'-1$ to process columns in the window. It is clear that the time complexity of the modified CUBE-N2 algorihm is $O(M{*}M')$.

## V. SHUFFLE-EXCHANGE MODEL WITH N$^2$ PEs

An $O(M^2)$ convolution algorithm for a $N^2=2^q$ shuffle-exchange array processor can be arrived at by simulating the routes in procedure CUBE-N2, using the technique used in [DeNS81]. We shall use the same logically two-dimensional view of PEs and the same initial configuration described in Section III and Section IV. Initially, $A(i,j)=G_{ij}$ for $i,j\in[0,N-1]$, and the $M$ by $M$ window is stored in the control unit memory in column-major order with the starting address $\alpha$. The resulting similarity measures will be in the register $C$.

The basic steps in the shuffle-exchange model convolution algorithm are the same as those used in CUBE-N2. The only difference is the data routing. In the shuffle-exchange model, we have to employ EXCHANGE, SHUFFLE, and UNSHUFFLE to simulate data transmission in the hypercube model. In the hypercube model, PE($p$) is directly connected to PE($p^{(b)}$) and the fundamental data transmission is $R(p^{(b)})\leftarrow R(p)$ for $0\le b<q$. Clearly, the following subroutine in the shuffle-exchange model implements the basic data transmission in the hypercube model.

```
proceudre TRANSMIT ($R$ :register; $b$ :integer)
begin
  for $i$:=1 to $b$ do $R$(UNSHUFFLE($p$)) ← $R(p)$;
    $R(p^{(0)})$ ← $R(p)$;
  for $i$:=1 to $b$ do $R$(SHUFFLE($p$)) ← $R(p)$;
end;
```

Note that the SHUFFLE and UNSHUFFLE connections are complementary operations. If several UNSHUFFLE operations are followed by the same number of SHUFFLE operations, the result will recover the search area to its original value. Since SHUFFLE or UNSHUFFLE rotates the bit position in the binary representation of index $p$, the selectivity function should take the corresponding adjustment if it is necessary.

Procedure ROTATE($R,s,r,flag$), developed in Section II for the hypercube model, should be modified to perform rotation for the shuffle-exchange model. The paprameters in the new procedure PS-ROTATE retain the same meanings as in the procedure ROTATE.

```
procedure PS-ROTATE($R,s,r$,flag);
begin
  for $i$:=1 to $s$ do $R$(UNSHUFFLE($p$)) ← $R(p)$;
    $R(p^{(0)})$←$R(p)$;
  for $i$:=1 to $r-s$ do
    $R$(UNSHUFFLE($p$)) ← $R(p)$;
    if flag=0 then
      $R(p^{(0)})$←$R(p)$, $(p_{q-1}=1$ & $\cdots$ & $p_{q-i}=1)$
    else (* Note that & is a logical AND operator *)
      $R(p^{(0)})$←$R(p)$, $(p_{q-1}=0$ & $\cdots$ & $p_{q-i}=0)$;
    end; (* end of if-then-else *)
  end; (* end loop $i$ *)
  for $i$:=1 to $r$ do $R$(SHUFFLE($p$)) ← $R(p)$;
end;
```

It is important to note that PS-ROTATE requires only twice as many unit-routes as does ROTATE in the hypercube model. Recall that each PE in a shuffle-exchange is connected to up to three 3 PEs, while in a hypercube each PE is connected to $q$ other PEs.

The parallel convolution algorithm SHEX-N2 for a shuffle-exchange network with $N^2$ PEs is obtained by (i) replacing the data transmitting statement $R(p^{(b)})\leftarrow R(p)$ with procedure TRANSMIT; and (ii) substituting PS-ROTATE for ROTATE in both the recursive procedure GRAY and algorithm CUBE-N2 from Section IV. SHEX-N2 also requires $M$ local memory locations for each PE. As mentioned above, PS-ROTATE takes about twice as many unit-routes as does ROTATE in the hypercube model. Thus, it takes $2*log_2(N/M)$ unit-routes to rotate **G** up or down one segment. Similarly, rotating **G** left one column takes $2*\log_2 N$ unit-routes.

Now we need to investigate the number of unit-routes required by procedure GRAY with parameter $U$. By definition of the GRAY sequence in Section IV, $S_U=S_{U-1},U,S_{U-1}$ where $S_U$ represents the sequence of bit positions on which the GRAY sequence with $2^U-1$ integers will take a complement operation to generate the corresponding integers. It is clear that $S_U$ is the same as an inorder traversal list of a complete binary tree with its root labelled $U$, and every other node of level $x$ labelled $U-x+1$. From this fact, an interesting property of the GRAY sequence is described in the following lemma.

**Lemma 4:** In $S_U$, corresponding to the GRAY sequence with parameter $U$, the number of integer $r$ 's, $1\leq r\leq U$, is $2^{U-r}$.

Each integer $r$ in $S_U$ corresponding to a GRAY sequence involves a data transmission $A(p^{(r)})\leftarrow A(p)$ in algorithm CUBE-N2. Therefore, by procedure TRANSMIT, it will take $2r-1$ unit-routes in the shuffle-exchange model. The inter-PE communication time in procedure GRAY with parameter $U$ is

$$\sum_{r=1}^{U}(2r-1)*2^{U-r} = 3*2^U-2*U-3$$

Therefore, phase 2 takes

$$\sum_{k=0}^{m-1}(3*2^k-2*k-3)+\sum_{k=0}^{m-1}(2k+1) = 3*2^m-2m-3\approx 3M$$

unit-routes. The overall time complexity of the SHEX-N2 algorithm is

$$\sum_{t=0}^{M-1}(M+2((3M-2\log_2 M-3)+2\log_2(N/M))+2\log_2 N)$$

$$=M(7M+6\log_2 N)\leq M*max(7M,6log_2 N)=O(M^2).$$

Note that the complexity of algorithm CUBE-N2 is $O(2M^2)$, while SHEX-N2 uses only triple as many unit-routes as does CUBE-N2 in the hypercube model. Each PE in the shuffle-exchange model is connected to up to 3 PEs however, instead of $q$ PEs as is the hypercube model.

## VI. ARRAY PROCESSORS WITH DIFFERENT NUMBERS OF PEs

The algorithms in the previous sections were developed on array processors with the number of PEs same as the size of the search area. We now demonstrate the modification of algorithm CUBE-N2 so that it can be applied to other numbers of PEs. The modification of algorithms on other types of array processors would be similar to this example.

First, let us consider the case of $L^2$ PEs, where $M\leq L\leq N$. Again we assume that $L$ is an integer power of 2. The search area **G** can be partitioned into $(N/L)^2$ equal-sized *blocks*, where each block $\mathbf{G}_{xy}$ is $L$ by $L$ for $0\leq x,y<(N/L)$. For a fixed $y$, we call procedure CUBE-N2 for blocks from $\mathbf{G}_{0y}$ to $\mathbf{G}_{zy}$, where $z=(N/L)-1$. Then we repeat the procedure call from $y=0$ to $y=(N/L)-1$. The only modification we have to make is the procedure ROTATE. In algorithm CUBE-N2, there are three places to call ROTATE: one for rotate up, one for rotate down, and one for rotate left. Instead of performing a rotate or end-around shift for those boundary elements, we must instead shift **G** from the neighboring (either down, up, or left) blocks. ROTATE can be easily modified to add this feature by reading the neighboring elements before performing the rotation. For example, to shift a block left one position, we can add the following statement to the beginning of ROTATE.

```
read block[$x*L:(x+1)*L-1;(y+1)*L+t:(y+2)*L+t-1$]
      ($p_{n-1:0}$=0 · · · 0);
```

Note that the leftmost column of the block to the right of the current block is read. Here $n=\log_2 L$.

An alternative approach does not involve modifying ROTATE. We can read the neighboring elements into another register, for instance, $B$. Suppose that we want to shift **G** up 4 positions. The top four rows of PEs (or $B$s) will be loaded with the element values of the top four rows of the blocks below it. Before we perform the shift on $A$, we exchange the contents of $A$ and $B$. Then, we perform ROTATE to get the desired result.

Now we present algorithm CUBE-L2 for a hypercube with $L^2$ PEs. **G** is initially stored in the I/O system and can be read block by block. The window **W** is stored in the control unit memory in the same way as was specified in CUBE-N2. Once it is obtained, the resulting similarity measure **C** is sent to the I/O system block by block.

```
procedure CUBE-L2
begin
  for $y$:=0 to (N/L)-1 do
    for $x$:=0 to (N/L)-1 do begin
      read block[$x*L:(x+1)*L-1;y*L:(y+1)*L-1$];
      Modified-CUBE-N2($x,y$);
    end;
end;
```

It is simple to show that the time complexity of CUBE-L2 is $O(N^2M^2/L^2)$.

Algorithm CUBE-N2M2, developed in [FaLN85], is a parallel template matching algorithm for an array processor with $N^2M^2$ PEs. By combining algorithms CUBE-N2M2 and CUBE-N2, one can obtain an efficient parallel convolution algorithm, CUBE-N2K2, for a hypercube with $N^2K^2$ PEs, where $1\leq K\leq M$. Algorithm CUBE-N2M2 has a time complexity of $O(log_2 N*\log_2 M)$. Meanwhile, algorithm CUBE-N2K2 has a time complexity of $O(M^2/K^2+\log_2 N*\log_2 K)$. One readily sees that when $K=1$, CUBE-N2K2 works exactly like CUBE-N2, and when $K=M$, CUBE-N2K2 works as CUBE-N2M2. When $K=M/\log_2 M$ and there are $N^2M^2/(log_2 M)^2$ PEs, it is interesting to see that the complexity of the algorithm CUBE-N2K2 is still $O(log_2 N*\log_2 M)$.

## VII. COMPUTATION FOR GENERALIZED CONVOLUTION

For some applications we need to normalize both arrays (search area and window) before the computation of convolution so that the similarity measure is a generalized digital convolution, i.e., normalized correlation. First we define

$$H=\sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} G_{i+s,j+t}$$

$$T=\sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} W_{st}$$

$$Z=\sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} G_{i+s,j+t}^2$$

$$D=\sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} W_{st}^2$$

$$E=\sum_{s=0}^{M-1}\sum_{t=0}^{M'-1} G_{i+s,j+t} * W_{st}$$

The normalized coorelation is thus defined as

$$G_{ij} = \frac{E-\frac{HT}{MM'}}{\sqrt{\left(Z-\frac{H^2}{MM'}\right)\left(D-\frac{T^2}{MM'}\right)}}$$

In each PE($p$), we use four registers MSI($p$), MWD($p$), MSIS($p$) and MWDS($p$) to store $H$, $T$, $Z$, and $D$, respectively. The modifications to algorithm CUBE-N2 include the following:

1. After a window column is read to local memory, cumulate M[MAR($p$)] into MWD($p$), cumulate M[MAR($p$)]$^2$ into MWDS($p$).
2. During the execution of multiplication-addition in both the main loop body and in procedure GRAY, cumulate $A(p)$ and $A(p)^2$ into MSI($p$) and MSIS($p$), respectively.
3. Before exiting, calculate the final result from contents in $C(p)$ and the four new memory locations.

The complexity of the algorithm generating the normalized correlation coefficient is still $O(M*M')$.

## VIII. CONCLUSIONS

Two-dimensional digital convolution, a basic operation in image processing, is computationally time consuming in traditional machines. We have proposed several parallel algorithms on array processors with different types of interconnection networks. PE operations can be carried out between the data movement steps among PEs, eliminating many unnecessary data transmissions. Our approach employs a small local memory requiring only M locations for each PE.

With $N^2$ PEs and $M=M'=2^m$, the communication time complexity of all three networks has the same order of magnitude $O(M^2)$. The degree of communication (connection links) of each PE is 4, $q$, and 3, for mesh, hypercube, and shuffle-exchange, respectively. However, the mesh connection provides the smallest communication complexity as summarized in Table 1. This is due to the inherent topological matching between mesh network and 2-D search area.

Table 1. Communication Time Complexity

| | |
|---|---|
| mesh | $3M^2$ |
| hypercube | $3M^2+M\log_2N-2M\log_2M$ |
| shuffle-exchange | $M*max(7M,6log_2N)$ |

## REFERENCES

[BaBr82] D. Ballard and C. Brown, *Computer Vision*, Prentice-Hall, 1982.

[DeNS81] E. Dekel, D. Nassimi and S.Sahni, "Parallel matrix and graph algorithms," *SIAM J. Comput.*, vol.10, No.4, November 1981.

[DuLe81] M.J.B. Duff and S. Levialdi, *Languages and Architectures for Image Processing*, Academic Press, London, 1981.

[FaDe85] Z. Fang and J. Deogun, "Control function transfer algorithm in multistage interconnection networks," *IEEE Trans. on Computers*, (to appear).

[Fang84] Z. Fang, "Mathematical theory of multistage interconnection networks analysis," *Ph.D. Dissertation*, University of Nebraska, Lincoln, August 1984.

[FaLN85] Z. Fang, X. Li and L.M. Ni, "Parallel algorithms for image template matching on hypercube SIMD computers," *Proc. of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp.33-40, November 1985.

[FrCh77] W. Frei and C.C. Chen, "Fast boundary detection: a generalization and a new algorithm," *IEEE Trans. on Computers*, vol.C-26, pp.988-998, 1977.

[FuIc82] K.S. Fu and T. Ichikawa, *Special Computer Architectures for Pattern Processing*, CRC Press, Florida, 1982.

[HwBr84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1984.

[HwFu83] K. Hwang and K.S. Fu, "Integrated computer architectures for image processing and database management," *Computer*, pp.51-60, 1983.

[LiNi85] X. Li and L.M. Ni, "A pipeline architecture for computing cumulative hypergeometric distributions," *Proc. of the 7th Int'l Symp. on Computer Arithmetic*, pp.166-172, June 1985.

[NiHw85] L.M. Ni and K. Hwang, "Vector reduction techniques for arithmetic pipelines," *IEEE Trans. on Computers*, May 1985.

[NiJa85] L.M. Ni and A.K. Jain, "A VLSI systolic architecture for pattern clustering," *IEEE Trans. on PAMI*, vol.PAMI-7, No.1, pp.80-89, January 1985.

[PrVu79] F. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel computation," *Proc. IEEE Symp. on Foundations of Computer Science*, pp.140-147, 1979.

[RoHB76] A. Rosenfeld, R. Hummel and S. Bucker, "Scene labeling by relaxation operations," *IEEE Trans. on Systems, Man and Cybernetics*, Vol.SMC-6, pp.420-433, June 1976.

[Roth79] C.H. Roth, *Jr., Fundamentals of Logic Design*, 2nd ed., St. Pauls, Minn.: West Publishing Co., 1979.

[Seit85] C. Seitz, "The cosmic cube," *Comm. of the ACM*, 28(1), January 1985.

[Sieg79] H.J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. on Computers*, vol.C-28, pp.907-917, 1979.

[SiSF82] L.J. Siegel, H.J. Siegel and A.E. Feather, "Parallel processing approaches to image correlation," *IEEE Trans. on Computers*, vol.C-31, March 1982.

[WoHa78] R.Y. Wong and E.L. Hall, "Sequential hierarchical scene matching," *IEEE Trans. on Computers*, vol.C-27, pp.359-366, 1978.

[WuFe80] C. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. on Computers*, vol.C-29, pp.694-702, August 1980.

# Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees

( Preliminary Version )

V. K. Prasanna Kumar and Mehrnoosh Mary Eshaghian
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

## Abstract

In this paper we consider the Mesh of Trees organization for several fundamental tasks in computer vision. We illustrate the suitability of this architecture by showing fast parallel algorithms for many tasks in low to medium level vision. We derive poly logarithmic algorithms for many problems on digitized pictures such as identifying and labeling figures in a 0/1 image, drawing digitized straight lines, computing convexity properties of digitized images, determining distances, etc. For all the above problems, the Mesh of Trees organization has superior time performance compared with the pyramid and mesh connected computer of corresponding size.

## I. INTRODUCTION

Mesh Connected Computers (MCC) and Cellular Arrays have been considered suitable for image processing, since images can be naturally mapped onto these so that neighboring pixels are mapped onto neighboring processing elements [MILL 85, ROSE 83]. But solving many image processing problems on a $N \times N$ 2-MCC requires as much as $O(N)$ time. An alternate parallel architecture is the pyramid organization which has logarithmic diameter [TANI 83, MILL 84]. The hierarchical organization of pyramid results in logarithmic time performance for many image processing tasks. However, many other problems require as much as $O(N^{1/2})$ time on a $N \times N$ base pyramid [MILL 84]. In this paper, we explore the suitability of the Mesh of Trees for image processing applications.

The Mesh of Trees organization has been introduced to solve several problems such as sorting, matrix multiplication and some graph problems [LGHT 81, NATH 83]. In this paper we propose this as an alternate architecture to the pyramid computer and evaluate its performance with respect to several fundamental tasks in image processing. For many problems on digitized images, the Mesh of Trees organization is shown to have superior performance compared to either the pyramid or 2-MCC of corresponding size.

The rest of this paper is organized as follows. In the next section, we define the structure of the Mesh of Trees and standard operations on it which will be used throughout the paper. In section III, we present $O(\log N)$ parallel algorithms for the following problems: determining the extreme points of the convex hull of a figure, enumerating and deciding the PEs within the convex hull of a figure, deciding if two sets of processors are linearly separable, finding the PEs along a digitized straight line, determining nearest neighboring figure to a figure, etc. We also derive poly logarithmic algorithms for several problems including labeling 0/1 images, estimating convexity properties for digitized pictures having multiple objects, etc. For several problems, the solution on the Mesh of Trees turns out to be considerably simpler than those on the pyramid. Due to space limitation proof sketches will be provided. Details appear elsewhere [PRAS 86b].

## II. MESH OF TREES

The Mesh of Trees network can be looked upon as an $N \times N$ matrix of processors in which each row and each column of processors forms the leaves of a binary tree [LGHT 81, NATH 83]. The root and the internal nodes of each binary tree are also processors. This structure is called an $(N \times N)$-MOT for short. The $N^2$ leaf processors form the base of the network and are called base processors (BPs). Most of the processing is done by the BPs. The internal processors (IPs) are used for communication between BPs. During the course of this communication, the IPs may also be required to carry out some simple operations such as summing and extracting the minimum on the data. A $(4 \times 4)$-MOT and its layout is shown in figure 1, where the BPs are represented by white circles and the IPs are represented by black circles. Any two adjacent rows or columns of the base are $O(\log N)$ distance apart. Since there are $N$ rows and $N$ columns, the total VLSI area of the layout is $O(N^2(\log N)^2)$, which can be shown to be optimal [LGHT 81]. Notice that, in the standard VLSI model, $(N \times N)$ 2-MCC and $(N \times N)$ pyramid have $O(N^2)$ area requirement.

0190-3918/86/0000/0270 $01.00 © 1986 IEEE

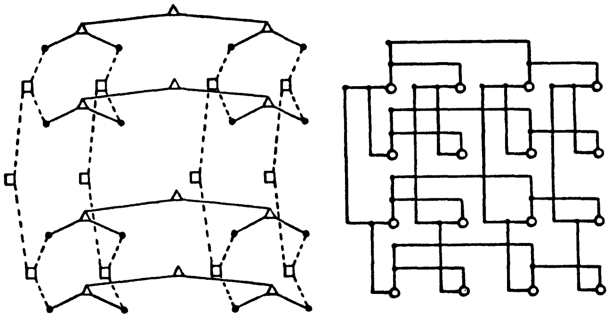


Fig. 1 (4×4)-MOT and it's layout

The following are some of the commonly used communication operations on the Mesh of Trees [NATH 83].

1) ROOTOLEAF-The contents of the data register of the root of the corresponding tree are broadcast to the leaves of the tree,

2) LEAFTOROOT-Selector specifies one BP whose register R contents are sent to the root of the corresponding tree.

3) LEAFTOLEAF-This can be expressed as a sequence of LEAFTOROOT and ROOTOLEAF operations.

The time required for the above operations is $O(\log N)$ which is the height of the tree.

## III. PARALLEL GEOMETRIC ALGORITHMS

In this section we consider several basic tasks on digitized images and derive fast parallel algorithms for these tasks. For all our algorithms the input is an $N \times N$ image with the base PE($i,j$) storing the pixel($i,j$).

### 3.1 Connected Components

Given a 0/1 image a fundamental task is to identify figures in the image. Figures correspond to connected 1's in the image. The labeling problem is to identify and associate an unique ID with the connected 1's in the image. An essential part of this algorithm is:

**Lemma 1:**[NATH 83] Given a $N \times N$ adjacency matrix of a $N$ vertex graph stored in the BPs of an $(N \times N)$-MOT, the connected components can be determined in $O((\log N)^3)$ time.

**Theorem 1:** Given a $N \times N$ 0/1 image, all figures can be labeled in $O((\log N)^4)$ time using an $(N \times N)$-MOT.

**Proof sketch:** The basic idea is to label blocks of size $k \times k$ and merge 4 adjacent blocks to form bigger size blocks. This is repeated until the block size becomes $N \times N$. To merge 4 blocks of size $k \times k$, we look at the boundary of the blocks as shown in figure 2. At the boundary between two blocks we can associate an *adjacency graph* as follows: each figure incident on the boundary corresponds to a vertex. Two figures adjacent to each other at the boundary has an edge between them. There can be $O(k)$ edges and $O(k)$ nodes across the boundary of two adjacent blocks. We then convert the $O(k)$ nodes and edges into adjacency matrix format using the basic data movement operations in $O(\log k)$ time. Now using lemma 1, the connected components of the *adjacency graph* is found in $O((\log k)^3)$ time. Repeating this along the horizontal boundary, we can determine the new labels within a block of size $2k \times 2k$. This information is propagated to the outer boundary of the block of size $2k \times 2k$ as shown in figure 2. Repeating this $\log N$ times, all the figure are labeled. By traversing the above steps in reverse, we can relabel all the figures. □

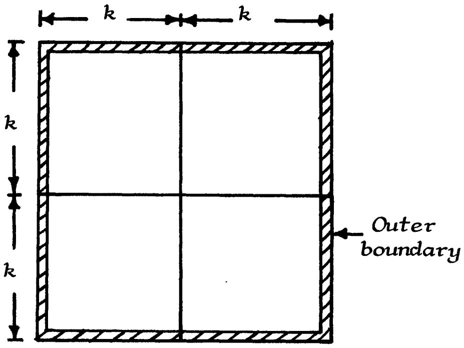


Fig. 2 Merging 4 blocks of size $k \times k$

### 3.2 Convexity Algorithms

Now we consider convexity problems. We use the following definition of convexity: A set of PEs is said to be convex if and only if the corresponding set of integer lattice points is convex. Given a set S of PEs, the convex hull of S, denoted Hull(S), is the smallest convex set of PEs containing S.

**Theorem 2:** In an $(N \times N)$-MOT, in $O(\log N)$ time one can identify the extreme points of 1's in a $N \times N$ image.

**Proof sketch:** The algorithm operates in two steps. It first finds the Right Most (RM) and the Left Most (LM) 1 in each row using the LEAFTOROOT and ROOTOLEAF operations. Then it verifies if these points are extreme points or not. The verification is done by having the $i$th column compute the enclosing angle made by the 1s in the image with the RM of the $i$th row ($RM_i$). This angle is defined to be the smallest angle $\phi_i$ such that all the 1s are enclosed inside the region X $RM_i$ Y as shown in figure 3. This can be easily computed in $O(\log N)$ time, using ROOTOLEAF and LEAFTOROOT operations. $RM_i$ is an extreme point if and only if $\phi_i < 180$ degrees. This step is repeated for LMs. □

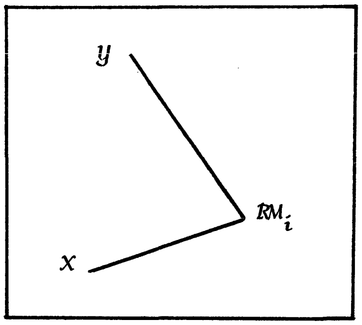


Fig. 3 Enclosing angle X $RM_i$ Y

**Lemma 2:** In an $(N \times N)$-MOT, suppose the extreme points of a set S have been marked. Then,

a. in $O(\log N)$ time the extreme points of S can be enumerated.

b. in $O(\log N)$ time the base PEs within the hull can be marked.

Using Lemma 2 we have:

**Corollary:** Using an $(N \times N)$-MOT , it can be decided if two sets of base processors are linearly separable in $O(\log N)$ time.

**Theorem 3:** Using an $(N \times N)$-MOT,

a. the diameter of a figure can be determined in $O(\log N)$ time.

b. a smallest enclosing box, a smallest enclosing circle for a figure can be determined in $O(\log N)$ time.

**Proof sketch:** a) The algorithm finds the diameter, by finding the maximum of distances between any two points on the boundary. The LM and the RM of each row having an extreme point is broadcast to all the BPs along that row. The $i$th column computes the distance between the extreme points in the $i$th row and the rest of the extreme points. In the next step, the maximum of those values is obtained in $O(\log N)$ time.

b) Given a set S of points in the plane, a smallest enclosing box is a rectangle of least area containing S. This rectangle must contain an extreme point of S on each side, and at least one of it's sides must contain two consecutive extreme points [FREE 75]. Therefore, after finding the extreme points of the convex hull enclosing the points, using theorem 2, each pair of consecutive extreme points is assumed to form the base of a rectangle. In $O(\log N)$ time, the extreme points of the convex hull of the figure lying on the other three sides can be found. Then the area of each of these rectangles is obtained. Using the data movement operations of section II, the rectangle with the minimum area is identified as the smallest enclosing box. Similarly, the smallest enclosing circle is a circle of least area containing S. This can also be computed in $O(\log N)$ time using a property of a smallest enclosing circle [FREE 75], and using the following fact [VOSS 82]: in a $N \times N$ digitized image, a figure can have at most $O(N^{2/3})$ extreme points. Details can be found in [PRAS 86b]. □

By combining adjacent convex hulls we can show:

**Theorem 4:** In an $(N \times N)$-MOT, one can determine and enumerate the extreme points of the convex hull of all figures simultaneously in $O((\log N)^4)$ time.

**Proof sketch:** The idea is to find convex hulls locally within blocks of size $k \times k$, merge the convex hulls within adjacent 4 blocks to construct convex hulls of figures within blocks of size $2k \times 2k$, $1 \leq k \leq \frac{N}{2}$. Notice that given two adjacent blocks of size $k \times k$, there can be at most $O(k)$ figures that run across the boundary between the blocks. Thus, $O(k)$ convex hulls need to be merged. Merging of convex hulls involves finding tangent lines $pq$ and $rs$ as shown in figure 4. This can be done by a binary search on the extreme points of A and B. Using suitable representation for the convex hulls and a basic technique in [OVER 80] we can show:

**Lemma 3:** Given convex hulls within blocks of size $k \times k$, the convex hulls can be merged to yield the convex hull of figures in block of size $2k \times 2k$, in $O((\log N)^3)$ time simultaneously for all the figures.

A basic step in the proof of the above lemma is to gather information about the relationship of a given line $xy$ (joining an extreme point of A with an extreme point of B) to the convex hulls A and B. If this line is tangential to both A and B, then we are done; otherwise we need to identify one of several cases [OVER 80]. A crucial step is to identify this information in parallel for all the $O(k)$ figures. By recursively representing information about the convex hulls and using efficient data movement this can be done in $O((\log k)^2)$ time [PRAS 86b]. □

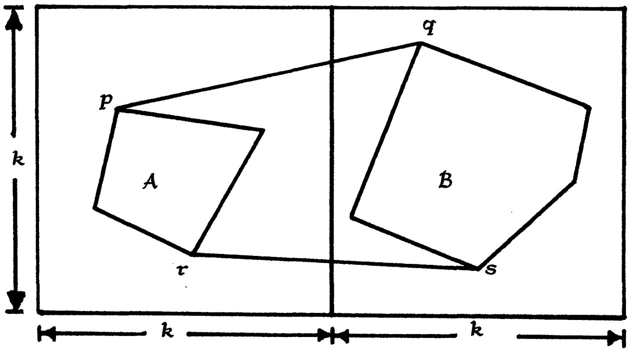


Fig. 4 Merging of convex hulls

### 3.3 Distance Problems

Now we consider several distance problems. In the following discussion we use the $l_1$ metric. However, it can be modified to operate for any $l_k$ metric.

**Theorem 5:** Using an $(N \times N)$-MOT,

a. the nearest neighbor 1 to each PE having a 1 can be found in $O(\log N)$ time.

b. the nearest neighboring figure to a given figure can be found in $O(\log N)$ time.

**Proof Sketch:** a) The algorithm starts by finding the nearest neighboring 1, along the X and Y axis, to each BP. This can be easily implemented by traversing the row and column trees. At the end of this step, each PE has the coordinates of the nearest PE with a 1, along the X and Y axis. Then each PE having a 1 broadcasts it's coordinates to all the PEs within it's nearest neighbor 1 in each direction. Each PE calculates the minimum distance between the broadcast address and it's nearest neighbor 1 along X and Y axis. In the return movement, the minimum of all these distances is sent back to the PE having a 1 which sets its nearest neighbor.

b) The idea of the algorithm is to first find the nearest neighbor for all the boundary PEs of the figure, and then to find the nearest figure, by finding the minimum among all those values obtained for the PEs at the boundaries. Details will appear in the full paper. □

A related problem is the closest pair problem for which efficient solutions are known on the pyramid computer [STOU 85]. This can be easily solved on the Mesh of Trees:

**Corollary:** Given an $(N \times N)$-MOT, the closest pair problem can be solved in $O(\log N)$ time.

## IV. CONCLUSION

In this paper we showed that many of the image processing tasks which required $O(N)$ computation time on an $(N \times N)$ mesh connected computer, can be computed in logarithmic time using a $(N \times N)$ Mesh of Trees. Indeed for several problems on digitized images, Mesh of Trees organization seems to be a natural candidate as opposed to the pyramid organization. This is illustrated by simple and elegant parallel algorithms for these problems.

## V. REFERENCES

[DYER 81] C. R. Dyer, "A VLSI pyramid machine for hierarchical parallel image processing", proc. IEEE conference on Pattern Recognition and Image Processing, 1981

[FREE 75] H. Freeman and R. Shapira, "Determining the Minimum-Area Encasing Rectangle for an Arbitrary Closed Curve", Communications of ACM, Volume 18, Number 7, PP409-413, July 1975.

[LGHT 81] F. T. Leighton, "New lower bound techniques for VLSI", IEEE FOCS, 1981.

[MILL 84] R. Miller and Q. F. Stout, "Convexity algorithms for pyramid computers", proc. 1984 International Parallel Processing Conference.

[MILL 85] R. Miller and Q. F. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh Connected Computer", IEEE Transactions on Pattern Analysis and Machine Intelligence, March 1985.

[NATH 83] D. Nath, S. N. Maheshwari, and P. C. P. Bhat, "Efficient VLSI networks for parallel processing based on orthogonal trees", IEEE transactions on Computers, 1983.

[OVER 80] M. H. Overmars and J. V. Leeuwen, "Notes on Maintenance of configurations in the plane", Technical report, Department of Computer Science, University of Utrecht, Netherlands, September 1980.

[PRAS 85] V. K. Prasanna Kumar and C. S. Raghavendra, "Array Processor with Multiple Broadcasting", Proceedings of the 1985 Annual Symposium on Computer Architecture, June 1985.

[PRAS 86a] V. K. Prasanna Kumar and D. Reisis, "Pyramids versus Enhanced Arrays for Parallel Image Processing", Technical report, Department of Electrical Engineering-Systems, U.S.C.

[PRAS 86b] V. K. Prasanna Kumar and M. M. Eshaghian, "Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees", Technical report, Department of Electrical Engineering-Systems, U.S.C.

[ROSE 83] A. Rosenfeld, "Parallel Processors for Image Processing: 2-D arrays and extensions", IEEE Computer, January 1983.

[STOU 85] Q. F. Stout, "Pyramid Computer Solutions of the Closest Pair Problem", Journal of Algorithms, PP200-212, 1985

[TANI 83] S. L. Tanimoto, "A Pyramidal approach to Parallel Processing", proc. 1983 International Symposium on Computer Architecture.

[VOSS 82] K. Voss and R. Klette, "On the maximum number of edges of convex digital polygons included into a square", Friedrich-Schiller-Universitat Jena, Nr. N/82/6.

# PARALLEL $A^*$ AND $AO^*$ ALGORITHMS: AN OPTIMALITY CRITERION AND PERFORMANCE EVALUATION

Keki B. Irani and Yi-fong Shih

Computing Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

Abstract-- A criterion is presented for validating the optimality of solutions acquired by parallel $A^*$ and $AO^*$ algorithms using multiple processors and a global *OPEN* queue or local best solution graph selection. It is shown analytically that the parallel $A^*$ algorithm with an *OPEN* queue can achieve linear speedups for up to a large number of processors when processor contention to access the queue is small. For the parallel $AO^*$ algorithm, utilizing Nilsson's solution graph cost revision in every processor gives a much higher performance than ordering solution graphs in a global queue.

## Introduction

The introduction of Very Large Scale Integration has enabled the construction of massively parallel systems[1,2]. In the field of Artificial Intelligence, parallel processing has been recognized belately as a powerful tool for accelerating the execution of search algorithms. In this paper we concentrate on both the state-space search and the AND-OR search. An optimality termination criterion for the parallel versions of the $A^*$ and $AO^*$ algorithms is presented and its correctness demonstrated. The solution that has the best merit at the time the criterion is first satisfied is the optimal solution.

In the past, research on performance evaluation of search algorithms concentrated on Branch-and-Bound search. Wah and Ma [3] have proposed an architecture, called MANIP, that uses a global register to hold the current best heuristic value. Wah has also given the performance speedups for Branch-and-Bound search[4,5]. Lai and Sahni [6] find that when using multiple processors it is possible to be trapped in an anamoly where the speedup dips below 1.0. Our emphasis, however, is on the state-space search and the AND-OR search algorithms that utilize admissable heuristic evaluation functions.

## Algorithm $E^*$

Throughout this paper we will use the same notations for heuristic evaluation functions (HEFs) as given in Nilsson[7]. We will refer to a heuristic evaluation function (HEF) as being *admissable* if the cost estimate function $h(n)$ of the HEF $f(n)$ satisfies the relation $h(n) \leq h^*(n)$ for all nodes $n$. A HEF $f(n)$ is also said to be *monotonic* if $f(n_1) \leq f(n_2)$ for any pair of nodes $(n_1,n_2)$ such that $n_1$ is a predecessor of $n_2$. We will call our parallel version of the $A^*$ algorithm $E^*$ algorithm. The algorithm $E^*$ runs on every PE in a system consisting of $p$ processors(PEs), a global OPEN queue for ordering unexpanded nodes, and a shared variable $\alpha$ which is initially set to $\infty$. The algorithm $E^*$ is identical to $A^*$ except that everytime a solution is found by the system, $\alpha$ is updated to contain the cost of the least costly solution found thus far. The purpose of $\alpha$ is explained in the next section.

## Proofs for the Alpha Criterion

Since our multiprocessor system is not synchronized, it is possible to obtain a suboptimal solution before an optimal one is found. It is therefore essential to derive a sufficient condition, called the $\alpha$-criterion in this paper, for determining the optimality of a solution that has been acquired. We will omit the proofs of the following sequence of propositions that shows the condition does indeed exist which optimally terminates the $E^*$ algorithm. For the derivations of all the expressions in this paper the reader shall refer to [8].

*Lemma 1:* If an optimal solution exists, $E^*$, without the $\alpha$-criterion, will acquire it.

*Lemma 2:* Before algorithm $E^*$ acquires the optimal solution there exists at least one node $n$ on the OPEN queue such that its heuristic merit $f(n) \leq \alpha$.

*Theorem 1:* If for every node $n_j$ on the OPEN queue the condition $f(n_j) > \alpha$ holds for all $j$, then the least costly solution that has been found by $E^*$ is the optimal solution.

The condition $f(n_j) > \alpha$ for all $j$ is referred to as the $\alpha$-*criterion*.

## Performance Evaluation

### Assumptions for A* and E*

Our purpose in analyzing the $E^*$ algorithm is primarily to derive an upper bound for the number of nodes that $E^*$ needs to expand in order to acquire the optimal solution. If the contention by the PEs to access the global queue is minimal, then this upper bound becomes indicative of the potential speedup possible for $E^*$. Therefore, in the analysis of the $E^*$ algorithm we will assume the amount of contention is negligible for one of two reasons: (1) $p$ is small so there is little contention or (2) hardware support for queue management is available for reducing the processor idle time due to contention caused by queue management overhead. Moreover, we make the following assumptions to simplify the evaluation of algorithms $A^*$ and $E^*$.

(1) There exists a unique optimal solution of cost $N$ $(=N_\gamma)$. There are $\gamma-1$ other suboptimal solutions with costs $N_1, N_2, \ldots, N_{\gamma-1}$. The heuristic merit $f(n)$, where node $n$ lies on the $i^{th}$ solution path, lies in the range between 0 and $N_i$. If node $n$ is on a path that does not lead to any goal, then $f(n)$ lies between 0 and $\infty$. The cost of each arc in the search graph is assumed to be 1.

(2) The search graph is acyclic and every node except the root $s$ has exactly one parent.

(3) The heuristic evaluation function $f$ is admissable and monotonic.

(4) $N_i \leq 2N$ for all $i$. The probability density of $f(n)$, where $n$ is on the $i^{th}$ solution path, at any point before reaching its goal is $\dfrac{1}{N_i - g(n)}$, where the end points of the density function are $g(n)$ and $N_i$.

(5) For evaluation, $E^*$ is assumed to execute on a cycle-synchronized multiprocessor.

The assumption of admissability implies that $A^*$ will expand a path from $s$ up to a node $n$ such that $n$ is the last node on this path to have $f(n) \leq f^*(s)$. Probabilistically, $f(n)$ of the final node $n$ expanded along a suboptimal path is equally likely to be less than $N$ as to be greater than $N$. Therefore,

$$N_i - N = N - g(n)$$

### Expansion Cost for A*

From Nilsson [7] if a HEF is admissable and monotonic then a node on OPEN, excluding the optimal goal node, will be expanded by $A^*$ if and only if $f(n) < N$. Henceforth, a node expanded by $A^*$ will be referred to as a *critical node*, and $E_I(z)$ will denote the expected number of nodes expanded by algorithm $I$ before the optimal solution is obtained.

$$E_{A^*}(z) = N + \sum_{i=1}^{\gamma-1} E(L_i)$$

where $E(L_i)$ is the expected length of a suboptimal solution path before the search stops for that path. To compute $E(L_i)$ we note that the expected length is at the point on the density function where the probability of $f(n) \leq N$ equals the probability that $f(n) > N$. Therefore,

$$E_{A^*}(z) = N + \sum_{i=1}^{\gamma-1} (2N-N_i)$$

### Expansion Cost for E*

Assume that all $p$ processors are used for every expansion cycle. Then there is a distinct possibility that some of the nodes expanded during a cycle may not be critical. We call these cycles *mixed cycles*. A cycle during which only critical nodes are expanded is called a *perfect cycle*. It can be shown that every node expanded by $A^*$ will also be expanded by $E^*$. When the monotonicity is considered as an additional constraint on the HEF, any non-critical nodes expanded by $E^*$ in mixed cycles cannot produce critical successors that may rank ahead of the nodes on the optimal path and slow down search. This leads to the following theorem.

*Theorem 2:* If a HEF is admissable and monotonic, then the critical nodes expanded by $A^*$ are exactly those critical nodes expanded by $E^*$.

Li and Wah [4] has shown that the number of mixed cycles $y$ in an $E^*$ search is less than or equal to $N$. Then $E_{E^*}(z)$ is

$$E_{E^*}(z) \leq E_{A^*}(z) + \left( yp - \sum_{i=1}^{y} m_i \right)$$

where $m_i$ is the number of critical nodes expanded during the $i^{th}$ mixed cycle. Knowing $y \leq N$ and $m_i \geq 1$ for all $i$ and using the uniform density assumption for all the nodes and letting all suboptimal solutions have the same depth $N_n$, we have

$$E_{E^*}(z) \leq pN + (\gamma-1)(2N-N_n)$$

Now we can estimate the speedup $U_p$ for a p-processor $E^*$ algorithm:

$$U_p = \frac{E_{A^*}(z)p}{E_{E^*}(z)}$$

$$\approx \frac{p\left(N + \left(\gamma-1\right)\left(2N-N_n\right)\right)}{pN + \left(\gamma-1\right)\left(2N-N_n\right)}$$

From the equations above we can see that the search speedup is at its best when we have a bushy search graph where the values of $N$, $N_n$ and $\gamma$ are large. To define the range of processors in which the speedup is almost $p$, we rearrange the speedup equation into

$$U_p \approx p - \frac{p^2 N - pN}{pN + \left(\gamma-1\right)\left(2N-N_n\right)}$$

For the speedup in the above equation to be approximately $p-1$, $p$ must be greater than 1 and less than $1 + \sqrt{1+\frac{1}{N}\left(\gamma-1\right)\left(2N-N_n\right)}$. As we can see this range increases for a larger and/or deeper search graph.

### Algorithm AO*

The $AO^*$ algorithm is the equivalent version of $A^*$ algorithm for the AND_OR problem reduction search. The purpose of $AO^*$ and its parallel counterparts is to find a solution graph from $s$ to the terminal set $T$ that has the minimum cost. The algorithm consists of two major operations (see Nilsson [7]): (1) a top-down graph-growing operation to find the best partial solution graph (PSG) and (2) a bottom-up, cost revising, connector-marking operation called *upward cost revision* (UCR). Starting with the node $n$ just expanded, this step revises its cost using newly computed costs of its successors, and marks the outgoing connector from $n$ on the estimated best path to the terminals. This revision process is repeated for every node in the PSG from the parent of $n$ to the start node $s$.

### Assumptions for AND-OR Algorithms

All the assumptions pertaining to the search graph for state-space search are assumed valid for the PSG in AND-OR search. In addition, every node in a PSG has $l$ connectors and every connector is a $k$-connector. The cost of every connector is exactly $k$. The probability density for the cost estimation function $f_i(s)$ ( the estimated cost of the $i^{th}$ PSG $G_i$ ) is uniformly distributed between its known arc cost and the actual total cost $N_i$.

### Total Cycle Time for AO*

From our assumptions it is obvious the algebraic expressions for $E_{AO^*}(z)$ and $E_{EO^*}(z)$ are identical to those for $E_{A^*}(z)$ and $E_{E^*}(z)$ respectively if the parameters are properly interpreted. Unlike the cases for $A^*$ and $E^*$, however, the speedup performance for $AO^*$ is different because the comparison cost in UCR must be taken into account. Similarily, the speedup expression for $EO^*$ will also be different because of the comparison costs in queue management and processor idle time waiting for PSG insertion into the global OPEN queue.

For the calculations that follow, let $\omega$ be the unit cost for a node expansion, and let

$$\epsilon = \frac{\textit{unit comparison cost}}{\textit{unit expansion cost}}$$

where the unit comparison cost is the time taken to compare two numerical values in a given computer system. The unit comparison cost is used in computing the UCR comparison cost, and in the search for a place in the OPEN queue when ordering a PSG. The unit expansion cost is the cost of expanding a node in a PSG.

The $AO^*$ cycle time in the $i^{th}$ cycle varies with the cycle number, the height of the PSG being expanded in the $i^{th}$ cycle, the unit expansion cost and the unit comparison cost. In our calculation, the height of the expanded PSG will be taken to be the height of an uniform search graph that has been expanded breadth-first.

The average height is

$$height_{avg}(i) = \lceil \log_{lk}(ilk) \rceil$$

where $i$(cycle number ) = 1,2,3,.... The total execution time over all cycles is

$$TCT_{AO^*} = \omega \sum_{i=1}^{E_{AO^*}(z)} \left(1+\epsilon(l-1)\lceil height_{avg}(i)\rceil\right)$$

$$\approx \omega\left(E_{AO^*}(z)\left(1+\frac{\epsilon(l-1)\ln(lk\ E_{AO^*}(z))}{\ln lk} - \frac{\epsilon(l-1)}{\ln lk}\right)\right.$$

$$\left. + \frac{\epsilon(l-1)}{\ln lk}(\ln lk + 1) - 1\right)$$

### Algorithm EO* and Its Total Cycle Time

To execute AND-OR search in parallel, Algorithm $EO^*$ employs a global queue to order PSGs. Since its implementation is virtually identical to that of $E^*$, $EO^*$ will also terminate when $\alpha$-criterion is satisfied. However, in computing the performance of $EO^*$ we will include the cost of comparison intrinsic to the insertion operations in queue management, and show that the comparison cost causes an unacceptable bottleneck at the OPEN queue. The average number of comparisons needed to locate a place to insert in OPEN will be taken to be $\frac{|OPEN_i|}{2}$, where $|OPEN_i|$ is the size of the OPEN queue in the $i^{th}$ cycle. The expansion process is partitioned into two phases: the initial phase in which $|OPEN_i| < p$, and the second phase in which $|OPEN_i| \geq p$. Then

$$TCT_{EO^*} \approx \omega\left(\frac{E_{EO^*}(z)}{p} + \frac{\epsilon}{2}\left(E^2_{EO^*}(z)(l^2-l)\right.\right.$$

$$\left.\left. + E_{EO^*}(z)\left(2\lceil \log_l p\rceil pl + 2l^{\lceil \log_l \rceil+1} - pl - 2pl^2\lceil \log_l p\rceil - l\right)\right)\right)$$

The $E^2_{EO^*}(z)$ term is the dominant term in the whole equation. The speedup for $EO^*$ is tabulated in Table 1 where the increasing size of the OPEN queue apparently increases the ordering cost, which in turn increases the $EO^*$ single cycle cost dramatically over that of $AO^*$ as the cycle number increases.

Table 1

$EO^*$ Speedup $(=\frac{TCT_{AO^*}}{TCT_{EO^*}})$

$l=3, k=2, \epsilon=0.1,$

$\gamma=100, N=50, N_n=75$

| $p$ | $U_p$ |
|---|---|
| 1 | 1.0 |
| 3 | 0.00116 |
| 9 | 0.00349 |
| 27 | 0.0105 |
| 81 | 0.0314 |
| 243 | 0.0943 |

## Distributed Queueing AND-OR Search: The FO* Algorithm

An algorithm that is more "distributed" than $EO^*$ is needed to eliminate the global bottleneck. Such an algorithm, called $FO^*$, assumes that an interconnection network connects all PEs to enable PSGs to be passed during the execution of $FO^*$. However, for the performance evaluation that follows, we assume that every PE has one or more PSGs and that no communciation between PEs takes place. Every PE does UCR to determine the next best PSG to expand. To terminate optimally, the system periodically checks if $f^j(s) > \alpha$ is satisfied for all $j$, where $f^j(s)$ is the cost of the best PSG in the $j^{th}$ processor.

### Cycle Types

Unlike $E^*$ and $EO^*$, $FO^*$ contains five distinct cycle types classified according to the patterns with which critical PSGs ( those having $f_i(s) \leq N$ ) and non-critical PSGs $(f_i(s) > N)$ appear in the PEs. Let $Q$ be the total number of critical PSGs in all the PEs at an instant during execution. Then the five cycle types can be described as:

(1) $PB$ $(p \leq Q)$

The $p$-best type $a$ cycle. Cycles belonging to this type has a critical PSG as the local best PSG in every PE, and every local best PSG is among the global $p$-best in the system. $PB_a$ is identical to the perfect cycle type in $EO^*$.

(2) $PB$ $(p > Q)$

The $p$-best type $b$ cycle. This type corresponds to the mixed cycle in $EO^*$. This cycle expands not only all the critical PSGs in the system but also some of the non-critical ones as well.

(3) $NPB_a$ ( $p \leq Q$)

The Not-$p$-best type $a$ cycle. Every OPEN has a critical PSG as its local best PSG, although not every one is global $p$-best.

(4) $NPB_b$ $(p \leq Q)$

The Not-$p$-best type $b$ cycle with $p \leq Q$. At least one PE in this cycle has a non-critical PSG, and therefore a non-global best PSG, as its local best PSG.

(5) $NPB$ $(p > Q)$.

The Not-$p$-best type $b$ cycle with $p > Q$. At least one PE in this cycle contains at least two globally best PSGs.

We may now define two probability parameters and a parameter for measuring the relative expansion power of an algorithm. First, the *existential probability* $P^i_e(I)$ is defined to be the probability of occurrence of cycle type $i$ in algorithm $I$ during execution. The *expansion probability* $P^i_{\exp}(I)$ is the probability of expanding the optimal solution in a cycle type $i$ of algorithm $I$. The product $P^i_e(I) \cdot P^i_{\exp}(I)$ gives us the *expansion power* of cycle type $i$ in algorithm $I$, where expansion power of a cycle type is the probability that it is that cycle type that will expand the optimal solution when given a random cycle. Using these two parameters we may define a new parameter, the power ratio $PR$, which measures the ratio of expansion powers of certain cycle types $M$ and $N$ in algorithms $C$ and $D$, respectively.

$$PR(C_M, D_N) = \frac{\sum\limits_{i \in M} P^i_e(C)\, P^i_{\exp}(C)}{\sum\limits_{j \in N} P^j_e(D)\, P^j_{\exp}(D)} = \frac{E_{DN}(z)}{E_{CM}(z)}$$

where $E_{DN}(z)$ and $E_{CM}(z)$ are the expected numbers of nodes expanded by cycle set $N$ in algorithm $D$ and cycle set $M$ in algorithm $C$ respectively.

### Expected number of nodes for FO*

Let cycle type 1 and type 2 be the perfect cycle and the mixed cycle in $EO^*$ respectively. Then

$$E_{FO^*}(z) = \frac{E_{EO^*}(z)}{PR(FO^*_{1,2,3,4,5}, EO^*_{1,2})}$$

$$\leq E_{EO^*}(z) \cdot \left( \frac{P^1_{\exp}(EO^*)P^1_e(EO^*)}{\sum\limits_{i=1,3,4} P^i_{\exp}(FO^*)P^i_e(FO^*)} + \frac{P^2_{\exp}(EO^*)P^2_e(EO^*)}{\sum\limits_{i=2,4} P^i_{\exp}(FO^*)P^i_e(FO^*)} \right)$$

We will denote the first term inside the parentheses power ratio $PR_1$, and the second term $PR_2$.

### Assumptions for FO*

If the number of critical PSGs in a $EO^*$ cycle is taken to be $Q$, then we can assume that the probability of expanding the optimal PSG in this cycle to be $\frac{p}{Q}$ if all $p \leq Q$. Thus $P^1_{\exp}(EO^*) = \frac{p}{Q}$ and $P^2_{\exp}(EO^*) = 1.0$ for the perfect and the mixed cycles of $EO^*$ respectively. For the cycle types of algorithm $FO^*$ we have:

(1) Type 1.

Same as the perfect cycle of $EO^*$. That is, $P^1_{\exp}(FO^*) = \frac{p}{Q}$.

(2) Type 2.

Same as the mixed cycle of $EO^*$. That is, $P^i_{\exp}(FO^*) = 1.0$.

(3) Type 3.

In this cycle type every local best PSG is a critical PSG, but not every local best PSG is also global best. Again w estimate the expansion probability to be $P^3_{\exp}(FO^*) = \frac{p}{Q}$.

(4) Type 4.

In this cycle there is at least one PE that has a non-critical, non-global best as its local best PSG. Letting the number of local best PSGs that are also global best be $r$, then $P^4_{\exp}(FO^*) = \frac{r}{Q}$, where $r \leq p$.

(5) Type 5.

In this cycle there exists at least one PE that has at least two critical PSGs as its top two PSGs. We use a lower bound $\frac{1}{G_1}$ for $P^5_{\exp}(FO^*)$, where $G_1$ is the number of global best PSGs in the PE that has the largest number of global best PSGs in the system. To understand this lower bound see [8].

Since we are only interested in an upper bound of $E_{FO^*}(z)$, we will assume that $P^1_e(EO^*) = P^2_e(EO^*) = 1.0$. The existential probabilities of all five cycle types of $FO^*$ is computed in the following section by using combinatorial analysis.

## The Power Ratio and Combinatorial Analysis

The existential probabilities can be determined by examining how $p$-best PSGs rank among other PSGs in heuristic merit in every processor. The number of combinations of $p$-best PSG rankings in different PEs, given the number of local queues that do not possess any $p$-best PSGs (known as the *bad queues*), is known as an *i-combination* (denoted by $BQ_i$ ), where $i$ is the number of local queues lacking $p$-best PSGs. For example, if $p=5$ and $i=2$, a possible configuration in the 2-combination is

queue 1: 1,4
queue 2: none
queue 3: 3
queue 4: 2,5
queue 5: none

where the numbers 1,4 in queue 1 indicates that queue 1 has the first and the fourth best PSGs in the system. We also define a *pattern* to be a set of numbers where each number corresponds to the number of $p$-best PSGs in an unspecified queue. For instance, if $i=2$, then there are two possible patterns: 3-1-1 and 2-2-1, where each number separated by a dash denotes the number of $p$-best PSGs queued up at a local OPEN queue without considering which processor this queue belongs. Every such number in the pattern is referred to as a *digit*. The number of combinations in $BQ_i$ ( combinations with $i$ number of bad queues ) can be written as:

$$BQ_i = \binom{p}{i}\sum_{all\ j}\left(\frac{\left(p-i\right)!\prod_{n=0}^{m}\binom{p-\sum_{l=0}^{n}c_l^j k_l^j}{c_{n+1}^j}}{\prod_{l=1}^{m}k_l^j!}\right)$$

where $c_0^j=0$ for all $j$, $1 \le i \le p-1$, and $j$ is the total number of patterns possible for $i$ bad queues, $p$ is the number of PE's, $c_l^j$ is the $l^{th}$ digit of the $j^{th}$ pattern, $k_l^j$ is the number of times the $l^{th}$ digit appears in the $j^{th}$ pattern and $m$ is the total number of distinct digits in the $j^{th}$ pattern. The existential probability of a cycle type $i$ can be expressed as

$$P_e^i(FO^*) = \frac{\sum_{m\in M} BQ_m}{\sum_{n=0}^{p-1} BQ_n}$$

Since every OPEN queue in $FO^*$ determines its best PSG by using local UCR, the equation for total cycle cost for $FO^*$ is identical to that of $AO^*$ except for a substitution of $\frac{E_{FO^*}(z)}{p}$ for $E_{AO^*}(z)$. Therefore,

$$TCT_{FO^*} \le \omega\left(\frac{E_{FO^*}(z)}{p}\left(1 + \frac{\epsilon(l-1)\ln\frac{(lkE_{FO^*}(z)}{p})}{\ln lk} - \frac{\epsilon(l-1)}{\ln lk}\right)\right.$$
$$\left. + \frac{\epsilon(l-1)}{\ln lk}\left(1+\ln lk\right)-1\right)$$

The Table 2 shows a decreasing trend for processor efficiency. However, the rate of decrease is relatively small, and the speedup is large enough to justify the use of local queueing over global queueing for small $p$.

Table 2
$FO^*$ Measurements
$l=3,k=2,\epsilon=0.1,\gamma=100,N=50,N_n=75$

| $p$ | $PR_1$ | $PR_2$ | speedup | efficiency | $E_{FO^*}(z)$ |
|---|---|---|---|---|---|
| 2 | 0.75 | 0.75 | 0.75 | 0.38 | 6867 |
| 3 | 0.70 | 0.59 | 0.96 | 0.32 | 8199 |
| 4 | 0.68 | 0.51 | 1.17 | 0.29 | 9179 |
| 5 | 0.67 | 0.51 | 1.38 | 0.28 | 9865 |
| 6 | 0.67 | 0.44 | 1.59 | 0.27 | 10449 |
| 7 | 0.66 | 0.42 | 1.80 | 0.26 | 11006 |
| 8 | 0.66 | 0.41 | 2.02 | 0.25 | 11368 |
| 9 | 0.66 | 0.40 | 2.24 | 0.25 | 11744 |
| 10 | 0.65 | 0.40 | 2.48 | 0.25 | 12014 |

## Conclusion

In this paper we have examined the performances of two parallel search algorithms. When only the number of nodes expanded is taken into account, we show that the speedup is linear for a large number of processors. However, when one also considers ths cost of managing a global queue, especially in the context of AND-OR search, then the availability of more processors in $EO^*$ actually slows down execution. Finally, it has been shown that algorithm $FO^*$, which uses local UCR, performs much better than $EO^*$.

## References

[1] K.E. Batcher, " Design of a Massively Parallel Processor ", IEEE Transactions on Computers , September, 1980, pp. 836-840.

[2] S.J. Stolfo, " DADO: a Tree Structured Machine Architecture for Production Systems ", 1982 Annual Conference of American Association for Artificial Intelligence , pp. 242-246.

[3] B.W. Wah and Y.W. Ma, " The Architecture of MANIP: a Multicomputer Architecture for Solving Combinatorial Extremem Search Problems ", IEEE Transaction on Computers , May, 1984, pp.377-390.

[4] G.J. Li and B.W. Wah, " Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms ", 1984 International Conference on Parallel Processing , pp. 473-480.

[5] G.J. Li and B.W. Wah, " Parallel Processing of Serial Dynamic Programming Problems ", 1985 COMPSAC , pp. 81-89.

[6] T.H. Lai and S. Sahni, " Anomalies in Parallel Branch-and-Bound Algorithms ", 1983 International Conference on Parallel Processing , pp. 183-190.

[7] N.J. Nilsson, Principles of Artificial Intelligence , Tioga Publishing Co., Palo Alto, CA, 1980.

[8] Y.F. Shih, Parallel Execution of Production Based Systems , Ph.D Thesis, 1986, University of Michigan, Ann Arbor, MI 48109.

# PARALLEL PREFIX ON FULLY CONNECTED DIRECT CONNECTION MACHINES

**Clyde P. Kruskal**
Department of Computer Science
University of Maryland
College Park, Maryland 20742

**Tom Madej**
Computer Science Department
University of Illinois
Urbana, Illinois 61801

**Larry Rudolph**
Institute for Mathematics and Computer Science
The Hebrew University of Jerusalem
Jeusalem, Israel

**ABSTRACT** This paper presents an algorithm to perform the parallel prefix operation on a linked list of items. Let $N$ be the length of the list, and let $P$ be the number of processors. The algorithm described here runs on a fully connected direct connection machine (DCM) in time $O(N/P + P^2)$. It attains linear speedup for $N=\Omega(P^3)$. Previously, linear speedup has only been attained on shared memory models.

## 1. Introduction

Graph algorithms for sequential machines have received intensive study, and by now the basics, at least, are well understood. In contrast, the study of parallel algorithms began fairly recently; even for some seemingly simple problems, efficient parallel algorithms are hard to find. Two fundamental problems are product computation and initial prefix computation. The *product computation* problem is to compute the product $a_0 o\, a_1 o \cdots o\, a_{N-1}$, given $N$ elements $a_0, a_1, ..., a_{N-1}$, and a binary, associative operation, denoted $o$. The *initial prefix* problem is to compute all $N$ initial prefixes $a_0$, $a_0 o\, a_1$, The initial prefix problem when solved in parallel is known as *parallel prefix*. In this paper, we present efficient parallel algorithms for product computation and initial prefix computation, when the elements are stored in a linked list. As shown in [8], these results can be used to obtain efficient parallel algorithms for other graph theoretic problems.

Since parallel processors will be built with a fixed number of processors, and since users tend to push the limits of their machine, it is important to consider the situation in which there are fewer processors than data elements. In such cases, the user wishes to use all the processors efficiently. We assume that there are $p \leq N$ processors cooperating (to solve a problem of size $N$). We desire parallel algorithms that solve a problem in $\Theta(t(N)/p)$ time, where $t(N)$ is the sequential time to solve the problem; this is called *linear speedup* and is always optimal.

Various models of parallel computations have been studied. In the *PRAM* model, a set of processors can all concurrently access shared memory. Three variants are distinguished depending on the type of simultaneous action permitted to each memory cell: The most powerful model, the *CRCW*, allows concurrent reading and writing to any cell by any number of processors. The *CREW* model requires that no two processors simultaneously write to the same cell; however, simultaneous reading is allowed. The *EREW* model restricts reading and writing so that no two processors can simultaneously access (read or write) the same cell. In contrast to the PRAM model, a somewhat more realistic model is the *Direct Connection Machine (DCM)*: Memory is divided into memory modules with an equal number of processors and memory modules. Any processor can access any cell in any module; however, simultaneous access to any module is prohibited (even if the accesses are to different cells).

The product and prefix problems have been extensively studied for the case that the memory layout of the input is *data independent*, i.e. the location of $x_i$ is determined by the index $i$ [3],[5],[6],[9],[11],[14]. In the *data dependent* versions of the product and initial prefix problems, the locations of the elements are given but it is not known which element is which. Only the location of the first element is given along with a map from the $i$th element to the $(i+1)$st element. In practice, the mapping will be represented as a linked list, so that if element $i$ is contained in $A[j]$ then element $i+1$ is contained in $A[Next[j]]$. It is easy to solve the product and initial prefix problems sequentially in $O(N)$ time by starting at the first element and following the links. Examples of such problems are that of computing the rank of each element on a linked list, or that of labeling each element of a linked list with the name of the first element of the list.

The *parallel prefix operation* may be described as follows. We are given a set $S$ of items, together with an associative binary operation $o$ on these items. Let

$$(x_1, x_2, \ldots, x_N)$$

be a list of items, $x_i \in S$, $1 \leq i \leq N$. The parallel prefix operation produces the list of items

$$(x_1, x_1 o x_2, \ldots, x_1 o x_2 o \cdots o x_N),$$

i.e. the $i$th item is the $i$th partial product $x_1 o x_2 o \cdots o x_i$. Parallel prefix is an extremely important operation on parallel machines. For example, it is used in efficient parallel algorithms to solve the following problems (see [1],[2],[4],[10],[12],[15]):

(1) summing, i.e. the operation is ordinary addition;

(2) broadcast a value to all processors;

(3) packing data items;

(4) preorder and postorder traversal of trees;

(5) graph problems, such as finding connected components.

In general, the efficiency of the known algorithms to implement parallel prefix depends upon the data structure used to represent the list of items. When the items are stored in contiguous memory locations, i.e. in an array, we have the following previous results. Let $P$ denote the

Part of this work was done while the first author was at the University of Illinois and the third author was at Carnegie-Mellon University. The first author was supported in part by a grant from IBM.

0190-3918/86/0000/0278 $01.00 © 1986 IEEE

number of processors. There is an $O(N/P + \log P)$ algorithm for parallel prefix on the Shuffle Exchange Machine (see [12]). This readily yields an algorithm of the same time complexity for a fully connected Direct Connection Machine (DCM) and for the (EREW) PRAM (exclusive-read, exclusive-write parallel RAM). Parallel prefix when the data items are stored in a linked list was first discussed in [18]. There is an $O(\frac{N \log N}{P \log(2N/P)})$ algorithm for the EREW [7]. It attains linear speedup for $N = \Omega(P^{1+\epsilon})$, $\epsilon > 0$ any constant. See [17] for randomized shared memory algorithms.

This paper describes an algorithm for parallel prefix on a linked list that runs in time $O(N/P + P^2)$ on a fully connected DCM. It attains linear speedup for $N = \Omega(P^3)$.

Recently, researchers have become interested in finding good models of parallel computation, and studying the differences between the models. Upfal and Wigderson [16] show that any on-line simulation of $T$ steps of a PRAM takes at least $\Omega(T\frac{\log P}{\log \log P})$ steps on a DCM. Furthermore, they show that a fully-connected DCM can simulate any computation of an EREW PRAM at the cost of a factor of $O(\log P(\log \log P)^2)$ in running time (assuming that the size of memory is only polynomial in the number of processors). Together, the two results say that PRAM's are, in some sense, more powerful than DCM's, but not much more powerful.

To really prove that PRAM's are more powerful than DCM's, one should produce a (natural) problem that can be solved faster on a PRAM than on a DCM — not merely an algorithm that cannot be simulated efficiently step by step. While it certainly may be the case that such problems exist and may even be common, it is not easy to think of reasonable candidates — much less actually prove that some problem is hard on a fully connected DCM. Very small problems will not separate the models, because, if the problem is small enough, every data item can be placed in a separate memory module. (Depending on the exact rules for accessing common locations in a memory module, a fully connected DCM with at most one data item in each memory module will be equivalent to some PRAM model.) Intuitively, large graph problems would seem to be plausible candidates because of the difficulty in following pointers in parallel when there is more than one concurrent reference to the same memory module. The techniques in this paper coupled with some recently discovered techniques [8], will produce efficient DCM algorithms for many graph problems; thus, perhaps surprisingly, large graph problems do not separate (the power of) the two models. Our results, however, leave open the possibility that intermediate sized problems separate the models.

## 2. Preliminaries

The model of parallel computation that we will be primarily concerned with in this paper is the *fully connected* DCM. There are $P$ processing elements (PEs), each of which has a (large) local memory. We denote the $i$th processor by $\mathrm{PE}_i$, $1 \le i \le P$, and will refer to its local memory by $\mathrm{M}_i$. Each pair of processors is connected by a data link. During one cycle of the machine clock a processor may access its own or any other processors local memory. However, only one processor may gain access to the same local memory during any single cycle. So if $k$ processors attempt to access the same local memory at the same time, then it will take $k$ cycles before all of their requests are satisfied. In particular, during a single cycle, a set of memory cells may be simultaneously accessed only if they are all in different local memories. This is in contrast with the EREW PRAM, where any set of $k \le P$ distinct memory cells can be accessed simultaneously.

## 3. The Parallel Prefix Algorithm: An Outline

### 3.1. The Algorithm

The main idea of our parallel prefix algorithm is to reduce efficiently the size of the list by a constant factor, call the algorithm recursively, and then obtain the parallel prefix for the whole list from the parallel prefix for the collapsed list. Each of the nodes in the list consists of two parts: a *data item* and a *next link* pointing to the next item in the list. The data items are stored in arrays *Value[ ]*, and the next links are in arrays *Next[ ]*. We reduce the size of the list by *pairing* adjacent nodes, as illustrated in Fig. 1.

We assume that the $N$ data items are distributed evenly among the processors, i.e. about $N/P$ items in each local memory. If $N \le P$, there is at most one item per processor, and we can use the usual parallel prefix algorithm, which is given below. We assume without loss of generality that the data items are located in the first $N$ local memories.

```
for $j \leftarrow 1$ to $\log N$ do
    forall $i$, $1 \le i \le N$, in parallel do
        if $Next[i] \ne NIL$ then
            $Value[Next[i]] \leftarrow Value[i] \mathbin{o} Value[Next[i]]$;
            $Next[i] \leftarrow Next[Next[i]]$;
        end if
    end forall
end for
```

Now we consider the case when $N > P$. The following is a general scheme for solving this problem. We assume the existence of a predecessor map *Pred*, which is easy to construct in $O(N/P)$ time.

```
General Parallel Prefix Algorithm
    if the list contains more than one element then
        pick a set S of nonadjacent elements in the list;
        for each $x \in S$ do
            {replace a pair of adjacent elements by one
              element}
            $Value[Succ[x]] :=$
                $Value[x] \mathbin{o} Value[Succ[x]]$;
            $Succ[Pred[x]] := Succ[x]$;
            $Pred[Succ[x]] := Pred[x]$
        end for;
        execute algorithm recursively;
        for each $\mathrm{A} \in \mathrm{S}$ do
            {expand element back into a pair}
            $Value[x] := Value[Pred[x]] \mathbin{o} Value[x]$;
            $Succ[Pred[x]] := x$;
            $Pred[Succ[x]] := x$
        end for
    end if
```

The first part of the algorithm solves the product problem, successively compacting the list until only one item is left; the second part, where the recursion unfolds, expands the list back, and computes the missing partial products. Any parallel algorithm that solves the product problem by successive compaction can be used to create a parallel prefix algorithm that expands the list back, by matching step by step the compression operations done by the parallel product algorithm. The resulting algorithm will have about twice the running time of the original algorithm. We shall henceforth consider only the product part.

At the outermost level of the recursion, each processor will process its $N/P$ elements one at a time (replacing a pair of adjacent elements by one element). An efficient parallel implementation will process $O(P)$ elements at each time step. This must be done while avoiding two potential conflicts: First, no two adjacent elements can be processed at the same time (which would destroy the linked list). Second, no two processors can access the same memory module at the same time. The first type of conflict must be avoided when one is implementing this algorithm on a shared memory machine; the second type of conflict is unique to DCM machines. We ensure that neither type of conflict occurs by using a memory access routine described below. After pairing elements at the outermost level of the recursion, the elements are packed so that there are approximately an equal number of elements in each processor. The algorithm is then called recursively.

### 4. A Memory Access Technique

In this section we will describe and analyze a method for organizing memory references on a DCM. This is essentially a refinement of an idea due to Gottlieb and Kruskal [4]. It will allow us to obtain efficient parallel algorithms on the DCM (for $N >> P$). This section is organized as follows. One of the components of the main algorithm is a distributed maximal matching algorithm, which is discussed in the first subsection. The next subsection describes the memory access method itself, which is actually an algorithm schema intended to be used as a procedure in other algorithms. In the third subsection we analyze the parallel time required by our algorithm, under natural assumptions that will hold in all of our applications.

#### 4.1. Maximal Matching Algorithm

A *matching* in a graph is a set edges in the graph, no two of which share the same vertex. A matching is *maximal* if no edge from the graph can be added to it and still obtain a matching. There is an obvious sequential algorithm for finding a maximal matching in $O(E)$ time: start with the empty matching, and iterate through the edges one at a time, adding the edge to the matching if neither of its endpoints is already covered. Here we informally describe an $O(V^2)$ sequential algorithm that is easy to parallelize.

Let $N$ be the number of vertices, and number them 0 to $N$-1. For convenience we assume that $N$ is a power of two. We use a Boolean array *Matched* [0..$N$-1]. We initialize *Matched* $[i] \leftarrow$ **false** for all $i$, $0 \leq i < N$. The basic idea of the algorithm is as follows. We try to match the first $N/2$ vertices with the last $N/2$ simply by checking each possible pair of vertices $(i,j)$, with $0 \leq i < N/2$ and $n/2 \leq j < N$. We then call the algorithm recursively, trying to match up the first $N/2$ vertices among themselves, and the last $N/2$ among themselves. Here is a program describing the first level of recursion.

```
for $k \leftarrow 0$ to $N/2 - 1$ do
    for $i \leftarrow 0$ to $N/2 - 1$ do
        $j \leftarrow N/2 + (i+k) \bmod (N/2)$;
        if not Matched[$i$] and not Matched[$j$]
                and $\{i, j\} \in E$ then
            add edge $\{i, j\}$ to $M$;
            Matched[$i$]$\leftarrow$true ; Matched[$j$]$\leftarrow$true ;
        end if
    end for $i$
end for $k$
```

Assume that we have a fully connected DCM with $N$ processors. This algorithm is trivially converted to a parallel algorithm by simply performing the inner loop in parallel with $N/2$ processors on each level of the recursion. It is easy to see that the active processors (e.g. the first $N/2$ at the first level, the first $N/4$ and the third $N/4$ at the second level, etc.) do not have any memory conflicts among themselves. Moreover, since active processors only try to match with inactive ones, we avoid the undesirable situation where $\text{PE}_{i_1}$ is trying to match with $\text{PE}_{i_2}$, which in turn is trying to match with $\text{PE}_{i_3}$. Note that the algorithm is truly distributed in the sense that each processor only needs to know the identities of those other processors that it wants to match up with.

The recurrence relation for the parallel time required by the algorithm is

$$T(N) = cN/2 + T(N/2)$$

where $c$ is some constant. The first term arises because the inner loop (the **for** $i$) can be carried out in constant time using $N/2$ processors, and the second term is the time needed for the recursive calls. These calls are carried out independently in parallel. This is easily solved to obtain $T(N) = O(N)$.

#### 4.2. Memory Access Algorithm

During a computation there will be times when each processor wishes to access several (possibly many) locations and at each access process the data therein. Often it does not matter in what order the accesses occur or in what order the data is processed. Each processor will have a list of locations it wishes to access, and therefore a list of memory modules along with the locations in each module it wishes to access. An efficient parallel program will arrange the accesses so that no two processors conflict at a module.

Formally, a *memory access map* is a directed graph $G = (V, E)$ with $|V| = P$. Associated with each directed edge $e \in E$ there is an integer *weight* $w_e > 0$. There are no multiple edges; however, loops (i.e. edges of the form $(v,v)$) are permitted. Each vertex $v \in V$ represents a processor. There is an edge $e = (i, j) \in E$ if $\text{PE}_i$ needs to access $\text{M}_j$ $w_e$ times. In addition we associate a set $X_{ij}$ of elements, $|X_{ij}| = w_e$, with the edge $e = (i, j)$. These sets are not really a part of the memory access map, but are kept locally in each processor. They represent items that $\mathrm{PE}_i$ needs to process. As part of the processing of these items it is necessary for $\mathrm{PE}_i$ to access $\mathrm{M}_j$. The order in which the items are processed is irrelevant to the overall algorithm. That is, there do not exist processors $\mathrm{PE}_i$ and $\mathrm{PE}_j$ and items $x \in X_{im}$, $y \in X_{jn}$, such that $\mathrm{PE}_i$ must process $x$ before $\mathrm{PE}_j$ can process $y$.

If the set of edges $E'$ represents a set of accesses to memory modules, then the domain($E'$) is the set of processors involved in those accesses. Formally, if $E' \subseteq E$, where $G = (V, E)$ is a directed graph, then the *domain* of $E'$ is defined by

$$\mathrm{domain}(E') = \{i \in V \mid \text{there exists } j \in V \text{ such that } (i,j) \in E'\} .$$

We are now ready to describe the memory access procedure. Each of the steps is executed in parallel by each processor. Throughout the algorithm only one processor is ever active in a given local memory at the same time. The only information that each processor $\mathrm{PE}_i$ initially needs is the sets $X_{ij}$ for $j \neq i$.

Let $\tau$ be a threshold. We will determine its exact value later.

PROCEDURE: ACCESS MEMORY.

[A1] Considering only edges whose weights are larger than the threshold $\tau$, find a maximal matching $M$ in the current memory access map.

[A2] Find the minimum weight $w$ of any edge in the matching. Broadcast $w$ to all of the processors.

[A3] Each $\mathrm{PE}_i$ with $i \in \mathrm{domain}(M)$ executes the following code. (If $i$ is not in domain($M$) then $\mathrm{PE}_i$ is idle.)

```
let $j \in V$ be such that $(i, j) \in M$;
for $k \leftarrow 1$ to $w$ do
    choose an item $x \in X_{ij}$;
    $X_{ij} \leftarrow X_{ij} - \{x\}$;
    process $x$;
end for
```

[A4] Each processor now updates the part of the memory access map contained in its local memory. To do this, set $w_e \leftarrow w_e - w$ for each edge $e \in M$.

[A5] Repeat steps [A2]-[A5] until the weight of every edge is less than the threshold $\tau$.

[A6] Each $\mathrm{PE}_i$ executes the following code.

```
for $j \leftarrow 0$ to $P-1$ do
    for $k \leftarrow 1$ to $s$ do
        choose an item $x \in X_{i,i+j \bmod P}$
            (if there is one);
        $X_{i,i+j \bmod P} \leftarrow X_{i,i+j \bmod P} - x$;
        process $x$
    end for
end for
```

### 4.3. Analysis

We now analyze the ACCESS MEMORY algorithm. In order to calculate how many iterations there are of steps [A1]-[A5], consider any vertex $v$ in the memory access map. Since we always have a maximal matching of the memory access map for edges whose weights are greater than $\tau$, during an iteration either a vertex $v$ is matched, all of $v$'s neighbors are matched, or $v$ has no incident edge with weight greater than $\tau$. In the first case, the weight of $v$ will be reduced by $w > \tau$; in the second case, the weight of all its neighbors is reduced by $w > \tau$. Thus the total number of iterations of steps [A1]-[A5] involving vertex $v$ is at most the sum of its weight and the maximum weight of any of its neighbors, divided by $\tau$. Since each processor has weight at most $\lceil N/P \rceil$, there are are at most $2 \lceil N/P \rceil / \tau$ iterations.

Step [A1] is the maximal matching algorithm discussed above, and each invocation takes time $O(P)$. So all together, step [A1] takes $O(N/\tau)$ time. Step [A2] involves the standard minimizations and broadcast operations, and at each iteration takes time $O(\log P)$; hence, the total time required for [A2] is $O((N \log P)/(P\tau))$.

Reasoning as above, the total number of iterations of the **for** loop in step [A3] involving vertex $v$ is at most the sum of its weight and the maximum weight of any of its neighbors. This is $O(N/P)$ iterations.

Step [A4] takes constant time at each iteration.

Step [A6] iterates through all the memory modules, spending $O(\tau)$ time at each module. Thus step [A5] takes $O(P\tau)$ time.

By summing the costs of all of the steps, we find that the parallel running time of the entire ACCESS MEMORY algorithm is $O(N/P + N/\tau + P\tau)$. This is minimized for $\tau = \Theta(\sqrt{N/P})$, making the total time $O(N/P + \sqrt{NP})$.

## 5. The Parallel Prefix Algorithm: Details

### 5.1. The Algorithm

We now give a detailed description of the fully connected DCM parallel prefix algorithm outlined in Section 3.

ALGORITHM: PARALLEL PREFIX ON A LINKED LIST (DCM)

[P1] First we construct the memory access map as follows. Each processor partitions its $N/P$ data items into $P$ groups. The items in group $j$ are those whose next links reference local memory $\mathrm{M}_j$ of processor $\mathrm{PE}_j$. Group $j$ for $\mathrm{PE}_i$ is denoted by $X_{ij}$. Put $w_{ij} = |X_{ij}|$. Module $i$ will contain $w_{ij}$ for all $0 \leq j < p$.

[P2] Each processor now calls the procedure ACCESS MEMORY defined in the preceding section. To process an item $x \in X_{ij}$, the following code is used to pair the item with the next item. All of the items are initially undeleted (i.e. *Delete* $[x]$ = False).

```
$Value[Succ[x]] := Value[x] \; o \; Value[Succ[x]]$;
$Succ[Pred[x]] := Succ[x]$;
$Pred[Succ[x]] := Pred[x]$;
$Deleted[x] :=$ True;
```

Note that a second invocation of ACCESS MEMORY is required to adjust the *A.Pred* values. The matching automatically ensures that two adjacent elements will not both be updated at the same time.

[P3] For each item in the original linked list, either it has been paired with another item, or its predecessor and successor have been paired. Thus the size of the collapsed list is no more than $2N/3 + O(1)$. Now repack the undeleted items in order to ensure that the items of the collapsed list are distributed evenly among the processors. The repacking procedure is discussed separately below. Then call the algorithm recursively on the collapsed list.

[P4] Upon return from the recursive call, we must unpack the list, "unpair" the items that were paired in step [P2], and correctly compute the parallel prefix for the original list. In order to do this we will need to keep a record of the items that are paired together in step [P2] and where they are moved in step [P3]. An easy modification of the repacking algorithm itself can be used to do the unpacking. Using our additional information, it is straightforward to obtain the parallel prefix for the original list from the parallel prefix for the collapsed list.

Next we describe an algorithm to accomplish the repacking needed in step [P3]. The similarity of the repacking routine to the entire parallel prefix algorithm should be apparent.

### 5.2. Repacking a Linked List

ALGORITHM: REPACK A LINKED LIST (DCM).

[R1] Construct a memory access map as follows. Each processor counts the number of list items in its local memory. For $PE_i$ we denote this quantity by $N_i$. Each processor then sends to every other one a message indicating the number of items that it has. Now compute the sum $N' = \sum_{i=1}^{P} N_i$. If $N_i > N'/P$ then $PE_i$ will be a *sender*. If $N_i < N'/P$ then $PE_i$ will be a *receiver*. Let $S$ denote the set of senders, $R$ the set of receivers. Order the sets $S$ and $R$ arbitrarily but deterministically, e.g. by processor index. A sender $PE_i$ has an *excess* of items, defined by $e_i = N_i - N'/P$; a receiver has a *deficit*, $d_i = N'/P - N_i$. We now define the bipartite digraph that will serve as the memory access map. The set of vertices will be $S \cup R$. A directed edge will always go from a vertex in $S$ to a vertex in $R$. Each edge is labeled with a weight. We determine the edges and their weights as follows. Let $PE_i$ be the first sender, and let $PE_j$ be the first receiver. Add the edge $(i, j)$ to the digraph. Label it with the weight $w = \min\{e_i, d_j\}$. We now readjust the excess $e_i \leftarrow e_i - w$ and also readjust the deficit $d_j \leftarrow d_j - w$. If $e_i = 0$ then $PE_i$ is not considered further. Similarly, if $d_j = 0$ then $PE_j$ is not considered further. We repeat this process with the remaining senders and receivers until there are no more processors to consider. Each $PE_i$ that is a sender will select $w_{ij}$ items arbitrarily and put them in the set $X_{ij}$, for each edge $(i, j)$ in the digraph. (Note: The sets $X_{ij}$ constructed here are distinct from the ones in step [P1].)

[R2] Now call the procedure ACCESS MEMORY to move list items from the senders to the receivers. To process an item $x \in X_{ij}$, $PE_i$ does the following:

```
let $y$ denote a new location in $M_j$;
$A[y] \leftarrow A[x]$;
$Next[y] \leftarrow Next[x]$;
$Next[x] \leftarrow y$;
mark $x$ as "moved";
```

[R3] We must now readjust the next links. To do this we follow the links in parallel. This actually involves another application of the ACCESS MEMORY procedure; however, it is exactly the same as used before where every processor had to examine the successor of each node. When we follow a link and find that $Next[x]$ is marked as moved, then we set $Next[x] \leftarrow Next[Next[x]]$. This is done for all of the data items, including the newly created copies, but excluding those items marked as moved. Fig. 2 illustrates how a list item is moved, and how the next links are then readjusted.

### 5.3. Analysis

In this section we will obtain the time bound for our parallel prefix algorithm. First note that step [P1] of the algorithm can easily be accomplished in time $O(N/P + P)$. Each next link may be viewed as an ordered pair $(i, \alpha)$, where $i$ designates a local memory $M_i$ and $\alpha$ specifies an address within $M_i$. Each processor sorts on the $i$ coordinate of the next links. This takes time $O(N/P + P)$ using a bucket sort with $P$ buckets. Construction of the first maximal matching requires $O(P)$ time as describe previously.

Step [P2], the ACCESS MEMORY routine, takes time $O(N/P + \sqrt{NP})$.

Let $N'$ denote the number of items in the new (collapsed) list. The purpose of the repacking in step [P3] is to ensure that the items of the new list are distributed evenly across the processors, i.e. $N'/P$ per processor. The repacking routine takes time $O(N/P + \sqrt{NP})$. To prove this, first note that it is easy to see that the memory access map defined in step [R1] has at most $P$ vertices and $P$ edges, and the method of construction clearly takes time $O(P)$. For step [R2], a maximal matching can be found in time $O(P)$ and at least one edge is (effectively) eliminated at each iteration of steps [A1]-[A5] of ACCESS MEMORY. So the total time spent finding maximal matchings is $O(P^2)$. Now we can apply our theorem again. In this case the weight of each node is its excess or deficit, which must be at most $N/(2P)$. Thus the total time spent moving items is $O(N/P)$, and so steps [R1] and [R2] of the repacking take time $O(N/P + \sqrt{NP})$. Finally, by a similar analysis it can be shown that step [R3] of the repacking takes time $O(N/P + \sqrt{NP})$. The memory access map needed for that step is similar to the one constructed in step [P1] of the parallel prefix algorithm.

It is readily seen that after step [P2], all of the links in the list will have been considered. It follows that for every node in the list, either it has been paired with its predecessor or its successor, or else both its predecessor and its successor (with the obvious exception for the head and tail of the list) have been paired. Thus the length of the collapsed list is no more than $2N/3 + O(1)$. Let $T_P(N)$ denote the time required by the entire parallel prefix algorithm on a list of size $N$. Then we have the recurrence relation

$$T_P(N) \leq c\,(\frac{N}{P} + \sqrt{NP}\,) + T_P(\frac{2N}{3})$$

for some constant $c$. We stop the recurrence when the number of items is less than $P$. At this point, the algorithm for one item per processor can be used to solve the problem in $O(\log P)$ time. The solution to the recurrence thus becomes:

$$T_P(N) = O(\frac{N}{P} + \sqrt{NP}\,)).$$

For $N >> P$ we have that $T_P(N) = O(N/P)$, so that the algorithm is totally efficient for $N$ large relative to $P$. In fact it is easy to see that the algorithm is totally efficient for $N = \Omega(P^3)$.

**6. Remarks**

For large $P$ a fully connected DCM is not feasible. It is well known, however, that some bounded degree connection machines, e.g. the Shuffle-Exchange machine, can simulate a fully-connected DCM at the cost of only an $O(\log P)$ factor in the running time. Nevertheless, due to the $P^2$ term in the running time, our algorithm as it stands is not practical for large $P$. Thus the obvious open problem is to find an efficient parallel algorithm for parallel prefix on a linked list that will run on a DCM and that is totally efficient for $N$ smaller than $\Theta(P^3)$.

## REFERENCES

[1] B. Awerbuch and Y. Shiloach, New Connectivity and MSF Algorithms for Ultracomputer and PRAM, *Proc. of 1983 ICPP,* Aug. 1983, 175-179.

[2] F.Y. Chin, J. Lam, and I-Ngo Chen, Efficient Parallel Algorithms for Some Graph Problems, *CACM* 25, 9 (1982), pp. 659-665.

[3] F. Fich, New Bounds for Parallel Prefix Circuits, Proc. of the 15th Annual ACM Symposium on Theory of Computing, May 1983, 100-109.

[4] A. Gottlieb, C. P. Kruskal, Complexity Results for Permuting Data and Other Computations on Parallel Processors, *JACM,* v. 31, n. 2, Apr. 1984, pp. 193-209.

[5] P. M. Kogge, Parallel Solution of Recurrence Problems, *IBM Journal of Research and Development,* Mar. 1974, 138-148.

[6] P. M. Kogge and H. S. Stone, A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, *IEEE Trans. on Computers,* Vol. C-22, Aug. 1973, 786-792.

[7] C.P. Kruskal, L. Rudolph, M. Snir, The Power of Parallel Prefix, *IEEE Transactions on Computers,* Oct. 1985, pp. 965-968.

[8] C. P. Kruskal, L. Rudolph, and M. Snir, Efficient Parallel Algorithms for Graph Problems, 1986 Intl. Conf. on Parallel Processing, Aug. 1986.

[9] R. E. Ladner and M. J. Fischer, Parallel Prefix Computation, *JACM,* Oct. 1980, pp. 831-838.

[10] G. F. Lev, N. Pippenger, and L. G. Valiant, A Fast Parallel Algorithm for Routing in Permutation Networks, *IEEE Trans. on Computers,* Vol. C-30, Feb. 1981, 93-100.

[11] J. Reif, Probabilistic Parallel Prefix Computation, *Proc. of 1984 ICPP,* Aug. 1983, 291-298.

[12] J. T. Schwartz, Ultracomputers, *ACM Transactions on Programming Languages and Systems,* Dec. 80, 484-521.

[13] M. Snir, On Parallel Searching, *ACM Symposium on Distributed Computing,* Aug. 1982, 242-253.

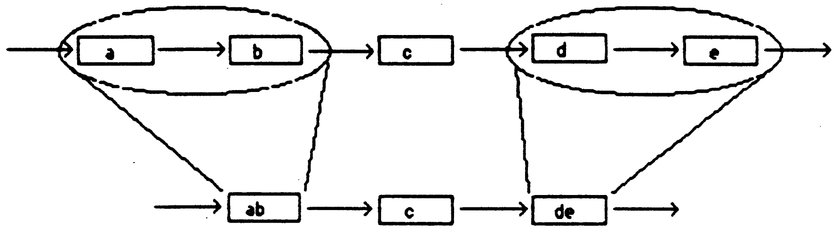
[14] M. Snir, Depth-Size Tradeoffs for Parallel Prefix Computation, manuscript.

[15] Robert E. Tarjan and Uzi Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Time, 25th Annual Symposium on Foundations of Computer Science, Oct. 1984, 12-20.

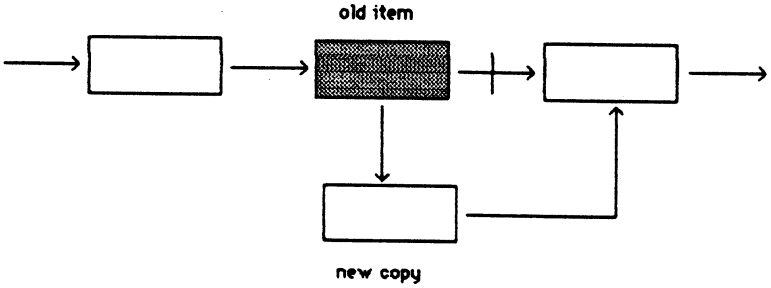
[16] E. Upfal and A. Wigderson, How to Share Memory in a Distributed System, *FOCS 25* (1984), pp. 171-180, preliminary version.

[17] Uzi Vishkin, Randomized Speed-Ups in Parallel Computation, Proc. of the 16th Annual ACM Symposium on Theory of Computing, Apr. 1984, 230-239.

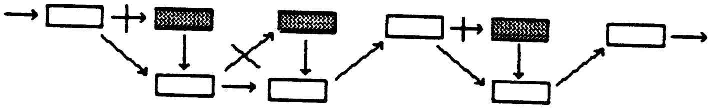
[18] J. C. Wyllie, The Complexity of Parallel Computation, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.



Combining list nodes.
Figure 1.



Moving an item from one processor to another.
Figure 2a.



Readjusting next links.
Figure 2b.

# REPLICATION AND PIPELINING IN MULTIPLE-INSTANCE ALGORITHMS

Jaime H. Moreno, Tomás Lang

Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024

**Abstract.** An algorithmic model and a methodology to evaluate the effectiveness of replication, pipelining, and local parallelism in the implementation of multiple-instance algorithms are presented. In the class of algorithms considered, instances are divided into groups with dependences between corresponding instances of consecutive groups. The implementations use a variable number of identical operation units, and the methodology permits the selection of the most efficient combination of concurrency techniques for a given execution time. The methodology is illustrated with an example, and has been used to obtain implementations for the Singular Value Decomposition computation.

## 1.- Introduction

Replication, pipelining, and local parallelism are well-recognized methods used, separately or in combination, to improve the speed of execution of digital systems. For a given computation, there are many alternatives in the use of these concurrency techniques to satisfy performance and cost constraints. In this paper, we study the effectiveness of those techniques for a class of computations encountered in a variety of important applications. This class has the following characteristics:

- The overall computation corresponds to the **execution of many instances** of a certain algorithm. This characteristic is the basis for the use of replication and pipelining. The **instances are not totally independent** but can be divided into groups of independent instances. This property limits the degree of replication and pipelining that can be used in an implementation, and complicates the tradeoffs between these techniques.

- An instance of the algorithm is described by a **directed graph** (where nodes represent subcomputations and arcs correspond to precedences among the subfunctions), without loops nor conditionals excepting those required to detect the end of the computation.

- The class of implementations we consider uses only **one type of operation unit.** This unit can be single function or multiple function.

As an example of computations that belong to the class outlined above, we can mention those that are described by expressions of the form

$$A_x^y = A_x^{y-1} \, (+,*) \, B^{y-1}$$

where $A$ and $B$ are terms of any complexity, usually involving matrices. This includes, for instance, matrix multiplication, LU-decomposition, transitive closure, and singular value decomposition[a] (SVD), which are frequently used in signal processing applications [1,2,3]. The main recurrences in these algorithms are procedures to update the values of a matrix using a modifier term that may depend on the current value of the matrix. These algorithms have been considered suitable for systolic array implementations [4,5,6].

Our goal for the class of computations described is to reduce the execution time for a given number of operation units. We are interested in identifying those implementations that offer better efficiency in terms of speed/cost. Since the algorithms of interest are compute bound and have implementations with negligible communication delay [4,5,6], only the computation time and the cost of the operation units are the relevant parameters for the design.

Our design methodology consists of top-down decomposition and bottom-up grouping of the nodes in the graph of an algorithm. Our results show that:

- for low throughput (or equivalently, up to a speed-up equal to the number of independent instances), replicated or pipelined implementations with efficiency equal to 1 can be achieved.

- for larger throughput, it is necessary to use combinations of the concurrency techniques, including the local parallelism in the graph of the algorithm. If portions of the graph have varying degrees of parallelism, pipelined implementations can be more effective because they require the addition of operation units only to those stages that exhibit greater local parallelism.

We have used this methodology to evaluate implementation alternatives for the singular value decomposition (SVD). It is shown in [7] that significant improvement in efficiency is possible by using a combination of pipelining and local parallelism, with respect to what is achieved in the replicated implementation (i.e. a linear systolic array) proposed in [5].

Previous research in the evaluation and selection of concurrency techniques includes [8] and [9]. However, these works deal with particular implementations (two-dimensional arrays, general purpose architectures, respectively), which are different from the ones sought here. Other related researches present formal models for the analysis of algorithms and ar-

[a] In the SVD, data dependencies are with two instances of the previous group instead of one. However, the model can be easily adapted to account for such situation.

chitectures [10, 11]. These approaches provide formal descriptions of hardware structures and algorithms, but assume the existence of an array of processors and attempt to map algorithms onto it, without evaluating other possible concurrent implementations. A combination of concurrency techniques is described in [12], which presents a two-level pipelined systolic array to perform convolutions. Such scheme deals with one specific algorithm and does not provide a methodology to select, among the alternatives for concurrency, an implementation for any given algorithm.

This paper is organized as follows. In Section 2, we present an algorithmic model for the class of computations of interest and a methodology for the design and evaluation of implementation alternatives. In Section 3 and 4 we discuss the characteristics of the concurrency techniques when applied to this class of algorithms, and we evaluate different implementation alternatives using only one or a combination of the techniques.

## 2.- Algorithmic Model and Methodology

In this section, we formalize a model for the class of algorithms considered, and define a set a performance and cost measures for implementations using concurrency. We also describe the methodology for the design and evaluation of such alternatives.

To design the system for a given computation it is necessary to describe the algorithm in a suitable form. Several models have been used to represent the dependences between subfunctions, including conditionals and loops [13, 14]. We use a directed graph in which nodes correspond to subcomputations and arcs indicate the precedences between these subcomputations. For the class of algorithms considered here, it is sufficient to use AND conditions as shown in Figure 1.

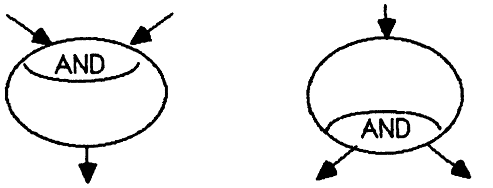


Figure 1 - Dependence Graph Elements

The primitive subcomputations used in the algorithm can have any level of complexity. Depending on this level (also called the **granularity** of the algorithm), implementations with different degrees of concurrency can be obtained. It is clear that larger concurrency can be obtained for finer granularity, which would indicate that it is always convenient to consider the representation with the finest granularity. However, this can lead to descriptions with large number of nodes, making the design of the system complex and unstructured. Therefore, it is convenient to resort to a structured design, in which one begins with the algorithm consisting of a relatively small number of nodes and then refines each of the nodes into subalgorithms. This top-down approach has the limitations that it is possible to loose some potential concurrency. The bottom-up approach is not totally satisfactory either, because the use of concurrency in the implementation of the node depends on how critical its execution time is in the overall algorithm. Consequently, the design process consists of several iterations until a satisfactory solution is found.

### Model of the Algorithm and the Implementation

From the considerations in Section 1, we can formalize the following model:

- The overall computation requires that an algorithm be executed for $M$ instances. These instances are divided into groups of $r$ each, where instance $x$ of group $y$ depends on instance $x$ of group $y-1$, as shown in Figure 2.

- The algorithm is described by a directed graph, in which the nodes are indivisible for a level of the implementation (i.e. the implementation of a node cannot be split across different processors or stages within a processor). Each node corresponds to a subfunction of the algorithm, and may exploit its internal concurrency using more than one operation unit. Therefore, for each node there might be more than one alternative implementation. To reflect this, the node is specified by a set of values $t_i(j)$ corresponding to its time of execution in an implementation with $j$ operation units.

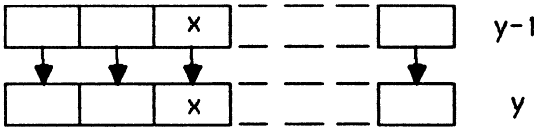


Figure 2 - Dependences between Groups of Instances

- Only one type of operation unit is used to execute all nodes. These operation units may be single function or multiple function and are considered indivisible. The use of a single type has implications in the design, since in such a case pipelining a sequential implementation of an algorithm requires additional units. (In contrast, if each node of the algorithm used a different type of unit, pipelining would not increase the number of those units).

- An increase in the number of operation units used to compute a node reduces its computation time at most proportionally to the number of units. Therefore,

$$t_i(j) \geq t_i(k) \geq \frac{j}{k} t_i(j) \quad , \quad k > j$$

For example, if a node corresponds to five independent operations as shown in Figure 3, then three or four operation units require two time units to compute the node, and the implementation with four units is not advantageous; however, the use of five units reduces the computation time to one.

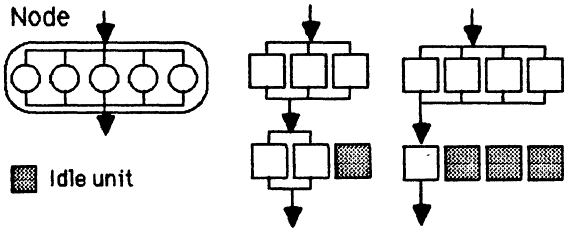


Figure 3 - Tradeoffs in Computation Time and Number of Operation Units

## Performance, Cost Measures, and Design Objective

Alternative implementations are compared using the following performance and cost measures:

- $t$: Time of execution of the computation (all $M$ instances of the algorithm).
- $N$: Total number of operation units.
- $SU = t_{cs} / t$ : Speedup of the concurrent implementation with respect to a **completely-sequential implementation** (with time $t_{cs}$), which uses only one operation unit to execute all nodes.
- $E = (t_{cs}\ N_{cs})/(t\ N)$ : Efficiency in a concurrent implementation with respect to the reference system. For our case, $N_{cs} = 1$.

Since we assume that there is only one type of operation unit, and since the computation time decreases at most proportionally to the number of units, **the speedup of any alternative is less or equal to $N$ and its efficiency is less or equal to 1.**

In terms of these measures, the main objective of the design can be described as selecting the implementation that for a given speedup (or computation time) has the largest efficiency (or uses the minimum number of operation units).

## 3.- Characteristics of Replicated, Parallel, and Pipelined Systems

We apply now the concepts of replication, pipelining and local parallelism to the class of algorithms described earlier. We evaluate the characteristics, performance, and cost measures for different implementations, using only one of the concurrency approaches in a given system, and provide a comparison of the resulting measures. In the next section, we will look at systems which use a combination of approaches.

To illustrate the choices that can be made we use the following example. We consider an algorithm that is executed for $M = 104$ instances and with dependences in groups of $r = 8$. At the topmost level in the design methodology, this algorithm appears as one node which requires only one operation unit and its computation time is $20M$ (completely - sequential implementation), as shown in Figure 4a. If higher throughput in the computation is desired, it is possible to decompose the node into subcomputations and then study the effectiveness of the concurrency techniques in the implementations of the decomposed algorithm. This process is repeated until a desired throughput is achieved.

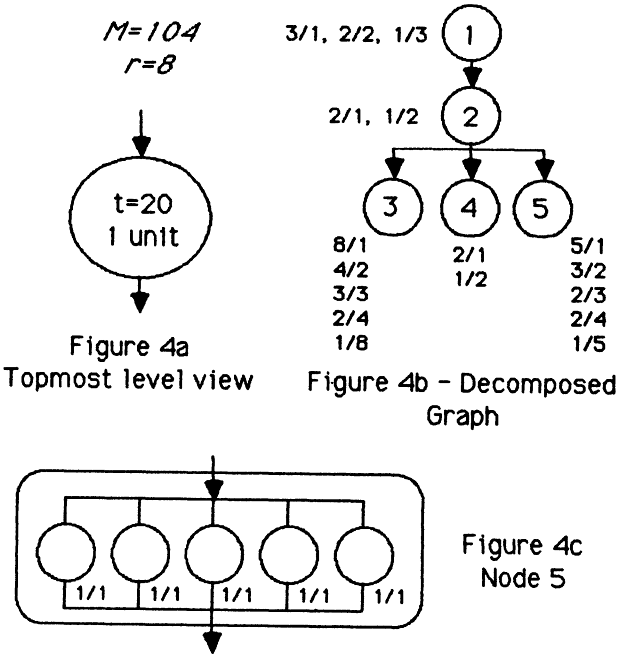


Figure 4a
Topmost level view

Figure 4b - Decomposed Graph

Figure 4c
Node 5

Figure 4 - Example Algorithm with Concurrent Capabilites

Consider now that the algorithm is described by Figure 4b. In this decomposed graph, each node has several alternative implementations with different number of operation units. This data was obtained from an analysis of each node, searching for internal parallelism and devising possible implementations for them. Figure 4c shows the structure of node 5 resulting from such analysis. The alternative implementations are indicated by the descriptors $t/n$ next to each node ($t$: computation time, $n$: number of units).

## Sequential Implementations

The sequential implementations are considered first, since they form the basis for some of the others. **An implementation of an algorithm is sequential if only one node (subfunction) is executed at a time.** To obtain this type of implementation the graph of the algorithm must have a total ordering of the nodes, which can be obtained by adding precedences. Since nodes may use more than one operation unit, we call $t_{seq}(j)$ the computation time of the sequential implementation that uses $j$ operation units. The following expressions describe the performance and cost measures for the sequential implementations:

$$\text{Comput. Time} \quad t_{seq}(j) = M \sum_i t_i(j)$$

$$\text{Speedup} \quad SU_{seq}(j) = \frac{t_{cs}}{t_{seq}(j)}$$

$$\text{Efficiency} \quad E_{seq}(j) = \frac{t_{cs}}{t_{seq}(j)\, j}$$

Table 1 includes the performance achievable with this scheme for the algorithm used as example.

### Replicated Systems

For our purposes, **a replicated implementation of an algorithm performs several instances of the computation simultaneously, using identical and separate hardware resources (processors) for each of the simultaneous instances.** Figure 5 shows an example of a replicated implementation of an algorithm. Due to the dependencies between instances in the class of algorithms considered, the number of instances executing simultaneously should be less or equal to $r$, the number of instances in a group. However, instances $x+1, x+2, ..., r$ in a group may be computed simultaneously with instances $1, 2, ..., x-1$ in the following group.

Consequently, if the hardware required to process an instance (a processor) is replicated $P$ times, the execution time of the $M$ instances, for $P \leq r$, is

$$t_{rep}(j,P) = \lceil M/P \rceil \frac{t_{seq}(j)}{M}$$

since the instances are performed in sets of $P$, excepting the last set which might be smaller than $P$. In particular, if $P = r$ then all instances in a group are processed at once and

$$t_{rep}(j,r) = \frac{t_{seq}(j)}{r}$$

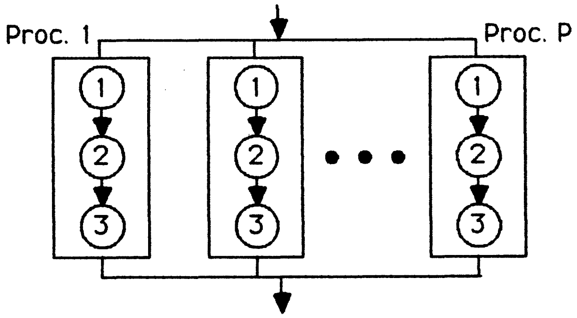


Figure 5 - Replicated Implementation of an Algorithm

The speedup and efficiency with respect to the completely-sequential implementation are

$$SU_{rep}(j,P) = \frac{t_{cs}}{t_{rep}(j,P)} = \frac{M}{\lceil M/P \rceil} SU_{seq}(j)$$

$$E_{rep}(j,P) = \frac{t_{cs}}{t_{rep}(j,P)\, jP} = \frac{M/P}{\lceil M/P \rceil} E_{seq}(j)$$

For the important case in which $M \gg r \geq P$, these expressions become

$$SU_{rep}(j,P) = P \; SU_{seq}(j)$$

$$E_{rep}(j,P) = E_{seq}(j)$$

Consequently, the maximum speedup is $r \; SU_{seq}(j)$ and the efficiency is the same as that of the sequential implementation that is being replicated. Therefore, **replication of the completely-sequential implementation is an optimal solution for speedup less or equal to $r$,** since efficiency $E = 1$ is preserved. For larger speedup, it is necessary to replicate a sequential implementation with $j$ units, which has lower efficiency.

| $j$ | $SU_{seq}$ $(j)$ | $SU_{rep}$ $(j,2)$ | | $SU_{rep}$ $(j,8)$ | $E_{rep}$ $(j,P)$ |
|---|---|---|---|---|---|
| 1 | 1.00 | 2.00 | -- | 8.00 | 1.00 |
| 2 | 1.82 | 3.64 | -- | 14.56 | 0.91 |
| 3 | 2.50 | 5.00 | -- | 20.00 | 0.83 |
| 4 | 2.86 | 5.72 | -- | 22.88 | 0.71 |
| 5 | 3.33 | 6.66 | -- | 26.64 | 0.67 |
| 6 | 3.33 | 6.66 | -- | 26.64 | 0.55 |
| 7 | 3.33 | 6.66 | -- | 26.64 | 0.48 |
| 8 | 4.00 | 8.00 | -- | 32.00 | 0.50 |

Table 1 - Replication of Sequential Implementations

Table 1 depicts this alternative for the algorithm in Figure 4, for different values of $j$ and $P$, up to the maximum replication $P = r = 8$.

### Pipelined Systems

In a pipelined implementation, **an algorithm is divided into stages and different instances are executed simultaneously at different stages** [15]. To perform the partitioning into stages, the algorithm should be sequential (i.e. precedences among nodes are such that a total ordering of nodes exists). Since each of the nodes in the graph is viewed as indivisible, two restrictions arise:

- A stage is composed of one node or a set of consecutive nodes. This partitioning is done (adding delays if necessary) so that the resulting stages have an (approximately) uniform delay.

- The maximum possible number of stages in the pipeline is equal to the number of nodes in the graph.

The number of stages $S$ is also limited by the number of independent instances, such that $S \leq r$. In terms of the number of stages $S$ and the stage delay $t_S$, the pipelined implementations are described by

$$\text{Comput. Time:} \quad t_{pipe} = [S + (M-1)]\, t_S$$

Speedup: $SU_{pipe} = \dfrac{t_{cs}}{[S + (M-1)]\, t_S}$

Assuming that the implementation that uses $j$ operation units is pipelined, then the stage delay with $j$ units is

$$t_S(j) \geq \frac{t_{seq}(j)}{M\, S}$$

Therefore,

$$SU_{pipe}\,(j,S) \leq \frac{M\, S}{S + (M-1)}\, \frac{t_{cs}}{t_{seq}(j)} = \frac{M\, S}{S + (M-1)}\, SU_{seq}(j)$$

In these expressions, the equality is achieved when the partitioning of the sequential implementation can be done perfectly, that is, into stages of uniform delay.

The total number of operation units is the sum of the units used in each stage. Since we are pipelining the sequential implementation with $j$ units, $j$ units per stage are used, which results in an efficiency with respect to the completely - sequential implementation of

$$E_{pipe}(j,S) \leq \frac{t_{cs}}{t_{pipe}(j,S)\, j\, S} = \frac{M}{S + (M-1)}\, E_{seq}(j)$$

In this case, the efficiency of the sequential implementation is reduced as a result of the startup time of the pipeline.

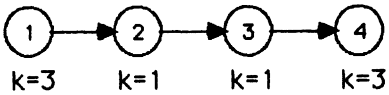
Of special interest is the situation in which $M \gg S$. In such case, the speedup and the efficiency tend to

$$SU_{pipe}(j,S) \leq S\ SU_{seq}(j)$$

$$E_{pipe}(j,S) \leq E_{seq}(j)$$

For perfect pipelining (no delay added to the stages) these expressions are identical to those obtained for replicated systems. Consequently, this alternative would be preferred to replication when the latter has implementation problems (such has interconnection complexity), since pipelining only requires communications between stages. However, pipelining is limited by the number of nodes in the graph of the algorithm.

Larger efficiency using pipelining can be obtained if all stages do not need the same number of units. Consider for example the algorithm shown in Figure 6, in a sequential implementation which uses three units. If such algorithm is pipelined, the stages in the middle do not need three units but only one (since fewer operations are involved), without affecting the computation time. As a consequence, the resulting efficiency is higher than what was obtained when all stages had the same number of units.



k: number of independent operations in a node

Figure 6 - Saving Operation Units in a Pipelined Implementation

| $j$ | $S$ | $SU_{pipe}$ $(j,S)$ | $E_{pipe}$ $(j,S)$ | $N_{saved}$ |
|---|---|---|---|---|
| 1 | 2 | 1.98 | 0.99 | 0 |
|  | 3 | 2.45 | 0.82 | 0 |
| 2 | 2 | 3.30 | 0.83 | 0 |
|  | 3 | 4.90 | 0.82 | 0 |
| 3 | 2 | 4.95 | 0.82 | 0 |
|  | 3 | 6.54 | 0.82 | 1 |
| 4 | 2 | 4.95 | 0.83 | 2 |
|  | 3 | 6.54 | 0.82 | 4 |
|  | 4 | 9.71 | 0.88 | 5 |
| 5 | 2 | 6.60 | 0.83 | 2 |
|  | 3 | 9.80 | 0.82 | 3 |
| 8 | 2 | 6.60 | 0.83 | 8 |
|  | 3 | 9.80 | 0.82 | 12 |
|  | 5 | 19.20 | 0.96 | 20 |

Table 2 - Pipelining of Sequential Implementations

Table 2 shows what is achievable with this approach in the algorithm described in Figure 4. The last column in this table indicates the number of units which are not needed for each one of the implementations.

The pipelined implementations require registers between stages. The addition to the cost and to the execution time that these registers produce is assumed to be negligible. This is true if the cost and delay of the stages are much larger than those of the registers. Also, the control of the system becomes somewhat more complex since each stage has to be controlled independently and the delays of the stages have to be made equal.

## Systems with Local Parallelism

We consider that **an implementation of an algorithm has local parallelism when it exploits the parallelism present in the graph to perform independent nodes concurrently.** Consequently, the time of execution of the sequential implementation might be reduced, but this may require additional operation units.

The speedup depends on the characteristics of the graph and on the scheduling of the nodes. An optimal schedule has to be devised to obtain the maximum speedup for a given number of operation units. In general, the determination of this schedule requires an exhaustive search, so several suboptimal heuristics and upper and lower bounds have been developed [16]. However, for algorithms with few nodes the exhaustive search is possible and convenient.

Since the improvements in computation time are, in the best case, proportional to the number of units, the speedup and the efficiency are bounded by

$$SU_{par}(j) \leq j \quad , \quad E_{par}(j) \leq 1$$

Without performing the actual scheduling it is not possible to know whether the parallel implementation (with $j$ operation units) is more efficient than the corresponding sequential one. That might be the case in computations with few instances or algorithms with dependences between groups of few instances.

This scheme is described in Table 3 for the example. A scheduling was performed for each value of $j$, which resulted in some implementations where nodes are executed in parallel (marked with *). The table also indicates the implementation chosen for each node.

In this section we have evaluated implementation alternatives for a given algorithm, using only one concurrency technique. The results obtained for the example algorithm are presented in Figure 7, where they are also compared with implementations using combinations of concurrency approaches, which are discussed in the next section.

| $j$ | $SU_{par}$ $(j)$ | $E_{par}$ $(j)$ | Implementation of Nodes 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| 2 | 1.82 | 0.91 | 2/2 | 1/2 | 4/2 | 1/2 | 3/2 |
| 3 | 2.50 | 0.83 | 1/3 | 1/2 | 3/3 | 1/2 | 2/3 |
| 4 | 3.33 | 0.83 | 1/3 | 1/2 | 2/4 | 2/1* | 2/3* |
| 5 | 4.00 | 0.80 | 1/3 | 1/2 | 2/4* | 2/1* | 1/5 |
| 8 | 5.00 | 0.63 | 1/3 | 1/2 | 1/8 | 1/2* | 1/5* |

** : nodes in parallel*

Table 3 - Implementations Using Local Parallelism

## 4.- Combinations of Replication, Local Parallelism and Pipelining.

In an implementation it is possible to combine two or all three of the approaches discussed previously. The characteristics of these combinations are described now and the corresponding performance and cost measures are evaluated.

### Replication and pipelining

In this case **the pipelined processor is replicated.** Because of the necessity of having enough independent instances, for the class of algorithms considered here the total number of processors and stages combined is limited so that $PS \leq r$. The following two possibilities exist:

- Replication of the pipelined completely-sequential implementation. This alternative produces a speedup of $PS$ and has efficiency 1 for large $M$ and values of $S$ that result in perfect pipelining. In such a case, this is equivalent in speedup and efficiency to the implementation that uses only replication. Therefore, its only advantage is that it requires fewer processors than the replicated implementation (which may replicate processors up to $r$, eventually creating realization problems such as a complex interconnection among the processors).

- Replication of the pipelined implementations that use more than one operation unit in some stages. These implementations increase the speedup of the single pipelined processor and maintain its efficiency. They are therefore suitable for higher speedups than that available with replication of the completely-sequential implementation.

For the example discussed, this scheme corresponds to implementations that are replications of the alternatives in Table 2, with $PS \leq 8$. Table 4 shows, for each possible number of stages, the configuration that offers the highest efficiency with that many stages.

| $S$ | Units per PE | $SU_{rep/pipe}(S,P)$ $P=1$ | $P=2$ | $P=3$ | $P=4$ | $E_{rep/pipe}$ $(S,P)$ |
|---|---|---|---|---|---|---|
| 2 | 4 | 3.30 | 6.60 | 9.90 | 13.20 | 0.83 |
| 3 | 8 | 6.54 | 13.08 | - | - | 0.82 |
| 4 | 11 | 9.71 | 19.42 | - | - | 0.88 |
| 5 | 20 | 19.20 | - | - | - | 0.96 |

Table 4 - Replication of Pipelined Processor with more than one Unit per Stage

### Replication and graph parallelism

**A processor which uses graph parallelism is replicated** in this implementation. It provides an increase of speedup with an efficiency equal to that of the implementation using only graph parallelism. Usually this efficiency will be significantly smaller than 1. Consequently, this scheme is only effective if the speedup cannot be achieved using a more efficient technique.

For the example discussed, this approach corresponds to implementations which are replications of the alternatives in Table 3. The maximum speedup achievable in the example with this scheme and the corresponding efficiency are

$$SU_{rep/par}(P=8, j=8) = 40.0$$

$$E_{rep/par}(P = 8, j = 8) = 0.63$$

Pipelining and graph parallelism

In this case, **one or more of the stages of the pipeline uses graph parallelism** (i.e. it executes more than one node concurrently). The partitioning into stages has to be modified (with respect to the implementation using pipelining only) to obtain stages of equal delay.

This scheme might be effective in increasing the efficiency of the pipeline, since it can help to get stages of equal delay and to tailor the number of operation units to the exact requirements of the stage. The number of stages is restricted by the number and the characteristics of the nodes. Nodes may be rearranged (preserving the dependences, of course) to achieve stages of (approximately) equal delay. The smallest stage delay possible is determined by the node with the longest computation time. The maximum number of stages is determined by the number of nodes in the critical path of the graph.

For the example discussed, Table 5 shows alternative implementation for different stage delays. Since the critical path in the graph traverses three nodes, up to three stages are possible.

| $t_S$ | $S$ | Total Units | $SU_{pipe/graph}$ $(j,S)$ | $E_{pipe/graph}$ $(j,S)$ |
|---|---|---|---|---|
| 7 | 2 | 3 | 2.83 | 0.94 |
| 5 | 3 | 4 | 3.92 | 0.98 |
| 4 | 2 | 6 | 4.95 | 0.83 |
| 3 | 2 | 8 | 6.60 | 0.83 |
| 2 | 2 | 11 | 9.90 | 0.90 |
| 1 | 3 | 20 | 19.61 | 0.98 |

Table 5 - Pipelining with Parallelism from the Graph

All three approaches

A possible application of this alternative is to **replicate the processor obtained by the use of a combination of pipelining and graph parallelism.** For the analysis of its effectiveness the same considerations apply as those discussed in the section on pipelining and graph parallelism, and on replication and pipelining.

For the example under discussion, the pipelined implementations in Table 5 are replicated as long as $PS \leq 8$. Table 6 shows the possible implementations. This approach produces implementations with high speedup and high efficiency. The system with highest speedup has the following parameters:

$$SU(S = 3, P = 2, j_{tot} = 40) = 39.22$$

| $t_S$ | $S$ | Units per PE | $SU_{all}$ | | | | $E_{all}$ |
|---|---|---|---|---|---|---|---|
| | | | $P = 1$ | $P = 2$ | $P = 3$ | $P = 4$ | |
| 7 | 2 | 3 | 2.83 | 5.66 | 8.49 | 11.32 | 0.94 |
| 5 | 3 | 4 | 3.92 | 7.84 | - | - | 0.98 |
| 4 | 2 | 6 | 4.95 | 9.90 | 14.85 | 19.80 | 0.83 |
| 3 | 2 | 8 | 6.60 | 13.20 | 19.80 | 26.40 | 0.83 |
| 2 | 2 | 11 | 9.90 | 19.80 | 29.70 | 39.60 | 0.90 |
| 1 | 3 | 20 | 19.61 | 39.22 | - | - | 0.98 |

Table 6 - Implementations with All Techniques

$$E(S = 3, P = 2, j_{tot} = 40) = 0.98$$

where $j_{tot}$ is the total number of units in all processors and stages.

Figure 7 illustrates the largest speedup achievable for different number of operation units, using either one concurrency technique or a combination of them. From the figure we infer that the selection of a particular implementation depends heavily on the characteristics of the algorithm and the number of operation units available. Depending on the characteristics of the algorithm, combinations of the different concurrency techniques might result more convenient than replication or pipelining of a completely-sequential processor. However, such conclusion is only possible after an evaluation of the alternative implementations of the algorithm.

Conclusions

We have identified replicated and pipelined implementations for computations that consist of multiple instances of a basic algorithm. We have concentrated on the case in which the multiple instances can be divided into groups, and there is a dependency between the corresponding instances of consecutive groups. Moreover, the implementations considered use only one type of operation unit.

We have devised a methodology to analyze such algorithms, using a dependence graph as the description tool. Such methodology leads to the evaluation of different implementations for an algorithm, at any level of decomposition. In this methodology, at a particular level one looks at all possible alternatives using either one or a combination of the concurrency techniques, searching for those cases which offer the highest efficiency. The analytical results obtained in this paper should be useful to facilitate the identification of those alternatives with higher efficiency. It might seem that this approach implies an exhaustive search for every possible implementation and is too costly in terms of the effort involved. However, that is not the case since the structured methodology allows to keep the number of nodes under evaluation at a given time restricted to a reasonable quantity. Therefore, it is possible to perform an intelligent search in the selection of the alternatives, which reduces the number of cases to evalu-

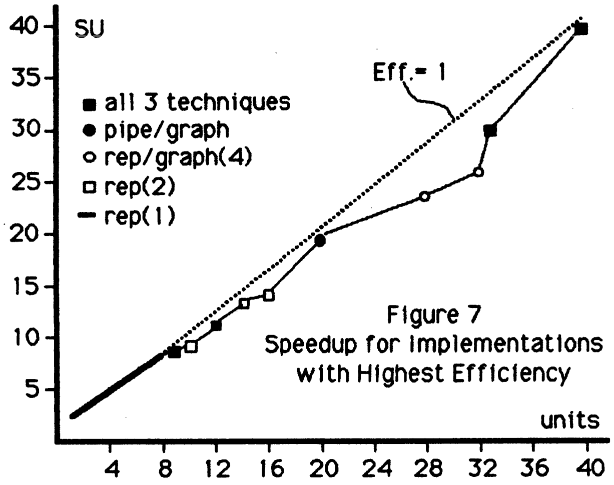


Figure 7
Speedup for Implementations with Highest Efficiency

ate.

Our results demonstrate that the selection of the most efficient implementation is strongly dependent on the algorithm that is implemented. This is particularly true when the dependence graph of the algorithm shows widely varying degrees of concurrency at different steps in the computation. Furthermore, we have been able to establish conditions for the concurrency techniques to be more effective in producing the highest speedup with a given number of operation units. Our results show that, for the class of algorithms considered, replication (or pipelining in some cases) alone is convenient up to a certain speedup value, with the limit imposed by the data dependences in the computation. For higher throughput, it is necessary to use combinations which include the local parallelism in the graph of the algorithm.

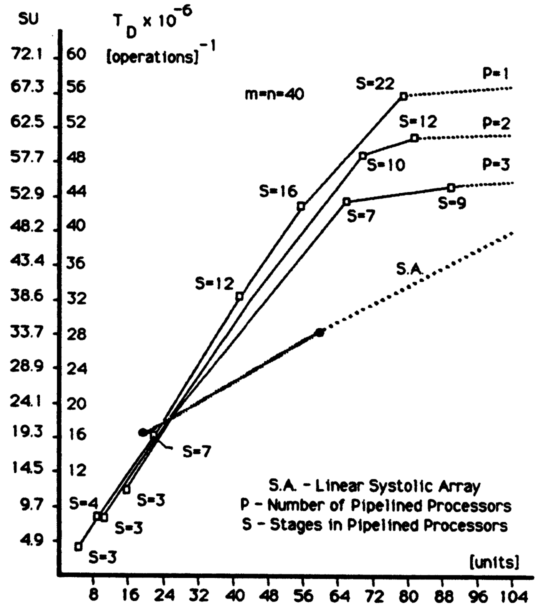


Figure 8 - Speedup and Throughput in Singular Value Decomposition for a 40 by 40 Matrix

We have applied this methodology to the analysis of alternatives for a Singular Value Decomposition (SVD) processor [7]. In particular, we have used it to compare pipelined implementations with graph-parallelism with respect to a linear systolic array (i.e. replication of a sequential implementation using graph parallelism) proposed for such computation [5]. Figure 8 depicts the difference in throughput obtained with those implementations. The plots show that the linear array is convenient only for lower throughput, but higher speedup is achieved with better efficiency in the pipelined architecture.

## References

[1] J. Speiser and H. Whitehouse, "Parallel Processing Algorithms and Architectures for Real-Time Signal Processing," *SPIE Real Time Signal Processing IV* **298**, pp. 2-9 (1981).

[2] J. Speiser, H. Whitehouse, and K. Bromley, "Signal Processing Applications for Systolic Arrays ," *14th Asilomar Conference on Circuits, Systems and Computers*, pp. 100-104 (1980).

[3] H. Ahmed, J. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *IEEE Computer* **15**, pp. 65-82 (January 1982).

[4] H.T. Kung, "Let's Design Algorithms for VLSI Systems," *CalTech Conference on VLSI*, pp. 65-90 (1979).

[5] R.P. Brent and F.T. Luk, "A Systolic Architecture for the Singular Value Decomposition," Tech. Rep. TR-82-522, Computer Science Department, Cornell University, (1982).

[6] K. Hwang and Y. Cheng, "VLSI Computing Structures for Solving Large-Scale Linear Systems of Equations," *1980 International Conference on Parallel Processing*, pp. 217-227 (1980).

[7] J.H. Moreno, "Analysis of Alternatives for a Singular Value Decomposition Processor," Tech. Rep. No. CSD-850035, Computer Science Department, UCLA, (1985).

[8] J.R. Jump and S. Ahuja , "Effective Pipelining of Digital Systems," *IEEE Transactions on Computers*, pp. 855-865 (September 1978).

[9] T. Chen, "Parallelism, Pipelining and Computer Efficiency," *Computer Design*, pp. 69-74 (January 1971).

[10] Y.P. Chiang and K. Fu, "Matching Parallel Algorithm and Architecture," *1983 International Conference on Parallel Computing*, pp. 374-380 (1983).

[11] I. Koren and G. Silberman, "A Direct Mapping of Algorithms onto VLSI Processing Arrays based on the Data Flow Approach," *1983 International Conference on Parallel Computing*, pp. 335-337 (1983).

[12] H.T. Kung, L. Ruane, and D. Yen, "A Two-Level Pipelined Systolic Array for Convolutions," pp. 255-264 in *VLSI Systems and Computation*, ed. Guy Steele (1981).

[13] G. Estrin, "A Methodology for the Design of Digital Systems - Supported by SARA at the Age of One," *AFIPS Conference Proceedings, NCC* **47**, pp. 313-324 (1978).

[14] J. Peterson, in *Petri Net Theory and Modelling of Systems*, Prentice Hall (1981).

[15] P. Kogge, in *The Architecture of Pipelined Computers*, McGraw-Hill (1981).

[16] M. Gonzalez, "Deterministic Processor Scheduling," *Computer Surveys* 9(3), pp. 173-204 (September 1977).

# HIERARCHICAL ARRAY PROCESSOR (HAP)
# FEATURING HIGH RELIABILITY AND HIGH SYSTEM PERFORMANCE

Tsutomu Ishikawa, Shigeharu Momoi, Shigeo Shimada, Yoshio Ogawa

Communications and Information Processing Laboratories
NTT Electrical Communications Laboratories
Musashino, Tokyo, Japan

Abstract--The HAP is a MIMD type highly parallel processor with 4096 PEs that basically uses NNM connection. It is capable of exceedingly high data transfer capability and reliability. Data transfer capability is upgraded by multi-layering PE arrays and utilizing its upper layer to transfer data in parallel to and from the lower layer, and also to reduce the inter-PE data transfer delay. For reliability upgrading, automatic high-speed system reconfiguration during failure is realized by a new fault-tolerant configuration and a new parallel diagnosis method that uses neighboring PEs. Thus, the HAP can be expected to gain high system performance in proportion to its number of PEs (max. 16 GIPS, 1.8 GFLOPS) and to realize the high reliability of nearly 1 availability and 1 year MTTF.

## 1. Introduction

Researches into parallel processors are being carried out on a worldwide scale [1]-[4], to meet the increasing needs for high performance computers. Especially approaches that use $10^3$-$10^4$ or more processing elements (PE) are receiving attention [5][6]. This is being stimulated by recent progress in LSI technology. In such a highly parallel processor, upgrading data transfer capability and reliability appear to hold the most promise for developing a practical machine, though, development of a parallel processing algorithm for each application is presupposed.

The system performance of a parallel processor is evaluated on the total time from initial data supply to processed data output. It is progressively limited by the data transfer time to and from all the PEs, rather than their processing time, as the number of PEs increases and therewith the overall processing power increases. Accordingly, the most important problem concerns data transfer to and from all PEs. The next important problem is the data transfer delay between PEs. This is because the maximum internode distance (1 + the number of PEs used as a relay) increases with the number of PEs in any network-type parallel processor [7], and the data transfer delay increases accordingly.

In considering reliability, imagine a system consisting of $10^4$PEs whose failure rate is $10^4$Fit. This means only a 10 hour mean time between failures (MTBF). In this case, one repair is needed on an average of every 10 hours, and if it is time consuming, the availability is lowered. Thus automatic failure recovery and shortening recovery time become important.

Conventional researches rarely touch on such problems. To cope with them, we researched the architecture of a system with the nearest neighbor mesh (NNM) connection selected to be suitable to a highly parallel structure. From those studies, we have developed a highly parallel processor, the hierarchical array processor (HAP). A small scale version is under trial at this time.

The HAP is a MIMD type processor with 4096 PEs designed for scientific calculation and speech, picture and other recognition processing applications. The previously mentioned problems are handled with a hierarchical PE array structure that utilizes its upper layer for data transfer, together with a new fault-tolerant configuration and parallel diagnosis of PEs.

## 2. System Architecture

### 2.1 System Configuration and Data Transfer

In order to cope with the data transfer problems, we adopted a hierarchical PE array structure similar to the EGPA [8], namely a large scale array with a small scale array above it. The number of PEs in the smaller array is approximately the square root of the large one's. By making use of the small array to accomplish data transfer to and from the large PE array and between PEs in the array, realization of a data transfer rate that matches the large array's processing power and reduction of the inter-PE data transfer delay are attempted. Hence, even if there is an increase in the number of PEs, a high system performance that is not limited by data transfer capability can be realized.

### 2.1.1 System Configuration

Figure 1 shows the system configuration of the HAP. In the HAP, multiple users are assumed in order to make full use of the processing power of all the PEs. Namely, it is properly used as the back-end processor for a number of user computers.

1) Configurations and Roles of Each Block

The PE array consists of a maximum of 4096 (64x64) PEs, and executes parallel tasks. The control PE array (cPE array) has a maximum of 64 (8x8) control PEs (cPE). Together with the data I/O mechanism, it performs the input-output of data to the PE array. It also relays the inter-PE data transfers. Besides these, hierarchical parallel tasks can also be executed. The PEs and the control PEs are basically microcomputers. A system management processor (SMP), using a general purpose computer, controls the whole system. The data I/O mechanism performs the input-output of data to user computers and cPEs, and also data buffering.

2) Physical Connection among PEs

PEs inside the PE array are both physically NNM

0190-3918/86/0000/0293 $01.00 © 1986 IEEE

and torus-connected, in consideration of the total data transfer capability / hardware quantities for connection between PEs, and the easy expansion to a physical structure. Therefore, each PE is connected with its four nearest neighbor (north, south, east and west) PEs. Although torus-connection slightly increases the inter-PE wiring length, it has the merit of reducing the maximum internode distance to half, compared to that of only an NNM, and is therefore applied. The cPE array also has the same connection scheme as the PE array. In inter-layer connection, such as the connection between PEs and cPEs, called lower PEs and upper PEs, respectively, a bus is used to reduce the amount of connecting hardware.

3) Logical Coupling between PEs

Logical coupling via memory is used between PEs, cPEs and both, in order to simplify program description and debugging (Fig. 2). Consequently, each PE can, as part of its own memory, directly access the memory contents of its neighboring PEs. Besides these, a cPE can also directly access the memory of the PEs to which it is connected. In this circumstance, when a particular PE is specified, only that memory can be accessed, but when not specified, the memories of all PEs can be accessed.

4) Bypass of the cPE Array

The cPE can be bypassed during the debugging of a PE program or during processing that does not require it. During such circumstances, the SMP and the PE array are directly logically-coupled.

2.1.2 Data Transfer

In the operation of a parallel processor, the program and the required initial data are supplied to the PEs in the first phase. Then, tasks expanded in parallel are executed in the PEs with necessary data transfer between the PEs. Finally, the processed data are collected by the user computer. That is, there are usually four types of data transfer in a parallel processor, namely, program load, initial data supply, inter-PE data transfer and result collecting. Since system performance is evaluated by the total processing time, which includes these data transfers, improvement in its transfer capabilities increases the value of a parallel processor. In the HAP, rapid data transfer is realized through the following approach.

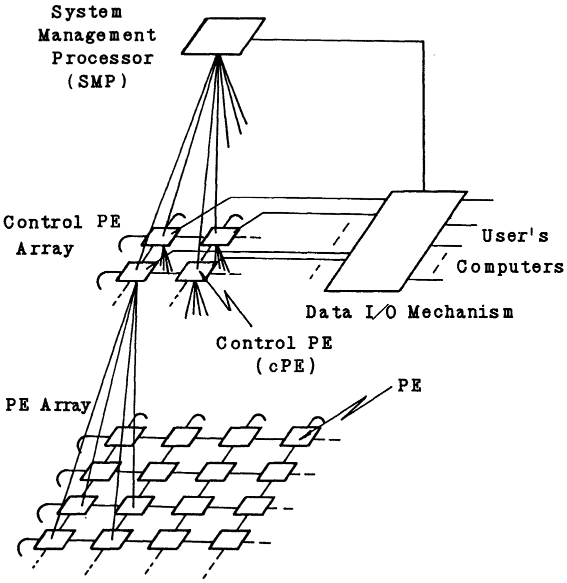


Fig. 1 Configuration of HAP System

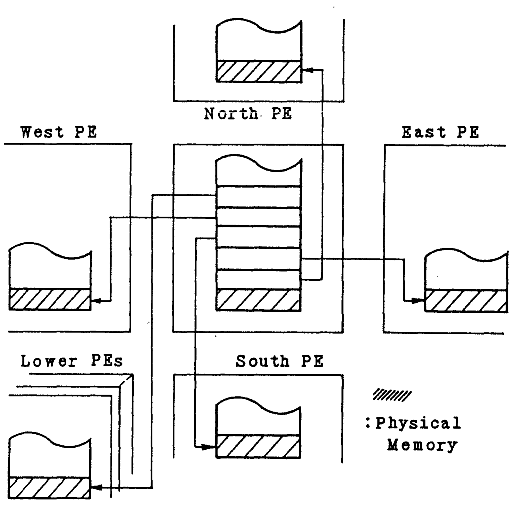


Fig. 2 Memory Sharing between PEs

1) Program Load

Load distribution, instead of function distribution, is used to improve the performance in highly parallel processors. The programs for all PEs are basically identical, so they are broadcast to all cPEs and PEs by the SMP. Moreover, independent program load to each PE is also possible, in consideration of occasions when the programs are different for part of the PEs. These are realized through memory accesses (write operations) to the lower PEs or PE by the upper PE.

2) Initial Data Supply and Result Collecting

Since the data to be processed by each PE is different, it is transferred from the SMP in serial fashion. This is a potential bottleneck in system performance. Therefore, in the HAP, data elements are supplied in parallel through the cPEs. Specifically the initial data buffered in the data I/O mechanism is transferred simultaneously to all the cPEs, then each cPE transfers it to the PEs that are coupled to it. Result collecting is basically similar, except the direction of data flow is reversed. These data transfers are DMA transfers by cPEs.

3) Inter-PE Data Transfer

A parallel processor with NNM connection is very suitable for processing problems where inter-PE data transfer occurs locally or regularly (e.g. solving Poisson's equation by the ODD-EVEN SOR method, or a Fast Fourier Transform (FFT))

Table 1 Inter-PE Data Transfer Method

| Method | Summary | Maximum Internode Distance | Maximum Transfer Parallelization |
|---|---|---|---|
| a) Routing | Same directional transfer in PE Array repeated by all PE's | $2\sqrt{N}-2$ | N |
| b) Array Relaying | Transfer relaying other PE's in PE Array | $\sqrt{N}$ | N |
| c) SMP Relaying | Transfer relaying the SMP (cPEs are bypassed) | 2 | 1 |
| d) Hierarchical Relaying | Transfer from PE to cPE, relaying in cPE array, and then from cPE to PE | $\sqrt{\frac{N}{Nc}}+2$ | Nc |

N : Number of PEs
Nc : Number of cPEs

[4][9]. However, it is not suitable for problems where transfers occur irregularly between PEs with large internode distances, e.g., logic simulation. This is because the maximum internode distance is still as large as $\sqrt{N}$ (N:the number of PEs) in this kind of parallel processor even if torus-connection is employed together with the NNM; therefore, much time is needed for these data transfers in such a problem.

To make it more suitable for the latter type of problem, a new data transfer mode utilizing the cPE (Table 1d) in addition to the usual mode (Tables 1a-1c) is provided to reduce the maximum internode distance and the inter-PE data transfer delay in the HAP. This mode is used for data transfer between PEs where the internode distance is greater than $\sqrt{N/Nc}$ (Nc:the number of cPEs). Hence, for the HAP, wherein N=4096 and Nc=64 (Fig. 1), the maximum internode distance is 10. This is shorter than that (equal to 12) of a hyper cube type [7] parallel processor with an equal number of PEs. Moreover, this mode can be used together with b), thus helping to improve the transfer rate. In the HAP, any of the several data transfer modes shown in Table 1 can be used, depending on the type of problem. These data transfers are realized through the repetition of memory access to the neighboring PE, memory accesses to the lower PE by the upper PE or both of them.

## 2.2 Fault-Tolerant Configuration

### 1) Redundancy Configuration and its Inter-PE Connection Network

There have been a number of proposals [10][11] for establishing redundancy in an NNM configuration. However, we use a new redundancy configuration that features simple inter-PE connection and easy switching control. Figure 3 shows its configuration schematically. One row and one column of spare PEs are provided for an n x n PE array. If the number of faulty PEs in the column does not exceed one, the column is considered good, thus n good PEs, out of n+1 PEs, are available. If there are more than 1 faulty PEs, the column is considered faulty, thus n good columns, out of n+1 columns, are available, forming an n x n good PE array.

With the fabrication of a switching circuit inside each PE, the switching of a faulty PE in this redundancy configuration is realized through the inter-PE connection network (called an IX network due to its shape) shown in Fig. 4 a). In the IX net, each PE is physically connected with 6 neighboring PEs. The switching of a faulty PE or a faulty column is done through bypassing them, as shown in Figs. 4 b) and c), respectively. That is, four connections are used out of a PE's six possible connections, and the IX net becomes functionally an NNM.

### 2) Effect of Application of Redundancy Configuration

The result of the application is shown in Fig. 5. Here [Degree of MTTF improvement G] = [MTTF of the redundantly configured system] / [MTTF of the nonredundantly configured system], where MTTF (Mean Time to Failure) is defined as the elapsed time from when all PEs are placed in good condition to when all spare PEs are used-up. Figure 5 shows that this redundancy configuration has a better effect than simply duplicating each PE, in spite of the smaller amount of redundancy, causing G to exceed $\sqrt{N}$ (N:the number of PEs), in the area where n is relatively small (<10). In the HAP, this configuration is used on each group

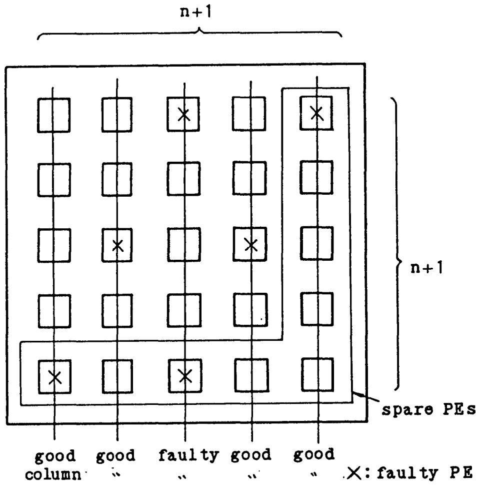
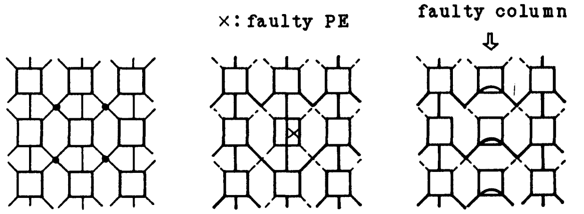


Fig. 3 Two-Level n-out-of-n+1 Fault-Tolerant Array



(a) IX-NET (b) Swithing Method for Faulty PE (c) Switching Method for Faulty Column

Fig. 4 IX-NET and Switching Methods for Faulty PE and Column

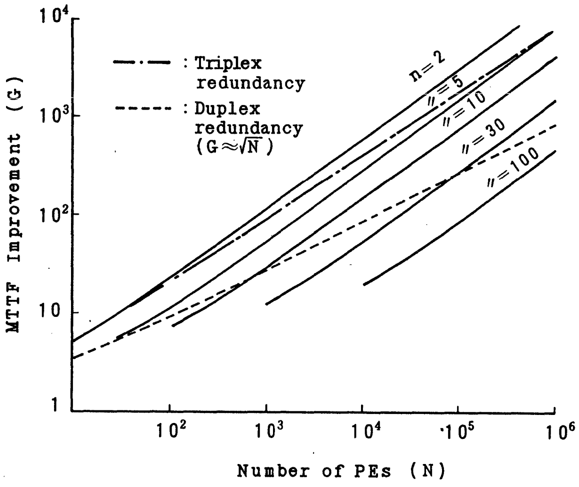


Fig. 5 MTTF improvement by Two-Level n-out-of-n+1 Fault-Tolerant Array

of PEs that are connected to a cPE. Hence, if the PE failure rate is $10^3$Fit, the MTTF of the HAP, excluding the SMP and the data I/O mechanism, would be about 1 year.

## 3. Processing Element (PE)

In the realization of a highly parallel processor with 4096 PEs, there is the necessity to miniaturize the PE. Though it would be ideal to fabricate the whole PE on a single LSI chip, market-available microprocessors, RAMs and gate arrays are used for the following reasons:

i) It is difficult to fabricate the PE for a MIMD machine, in which large memory is essential, on one LSI chip, even with the present advanced LSI technology.

ii) Fabrication of a memory-less PE on one LSI chip does not have much effect on miniaturization.

iii) Existing compilers and other software can be utilized without having to develop them when market-available microprocessors are used.

Moreover, PE and cPE have the same configuration thus reducing the number of LSI types that need to be developed for the HAP.

### 3.1 Configuration and Function

Figure 6 shows the configuration and table 2 shows the specification of the PE.

#### 1) Central Processing Unit (CPU)

Any of Intel's 80186, 80286, or 80386 processors can be used as the CPU. It executes fixed point arithmetic and controls the whole PE.

#### 2) Arithmetic Processing Unit (APU)

This is the co-processor for floating point arithmetic. Any of Intel's 8087, 80287 or 80387 processors can be used as the APU in a combination with the CPU shown in Table 2.

#### 3) Memory (MEM)

Besides the storage of PE programs and data, the MEM is used for various types of data transfer. From the point of view of upgrading PE performance, the MEM should be configured as two independent memory banks to avoid contention between CPU and APU memory accesses and data transfer memory accesses. However, in the PE of a MIMD machine, there are fewer of the latter type than the former (e.g. less than tenths in the ODD-EVEN SOR method). Accordingly, degradation of PE performance due to contention can be ignored and the MEM is configured as one bank. This makes miniaturization of the PE and expansion of the memory area used for data transfer possible.

#### 4) Memory Control Unit (MCU)

The MCU arbitrates the contention between memory accesses of the CPU and APU, and memory accesses due to data transfers, and it transmits data transfer requests and the several control requests to the CCU. In addition, it provides the interface between the microprocessors and the memory, namely the CPU and the APU and the MEM.

The MCU is configured so that it can be used by various types of microprocessors such as the CPU and the APU shown in Table 2. It is realized on a 1-chip gate array packaged in a pin grid array (PGA) case with 176 pins.

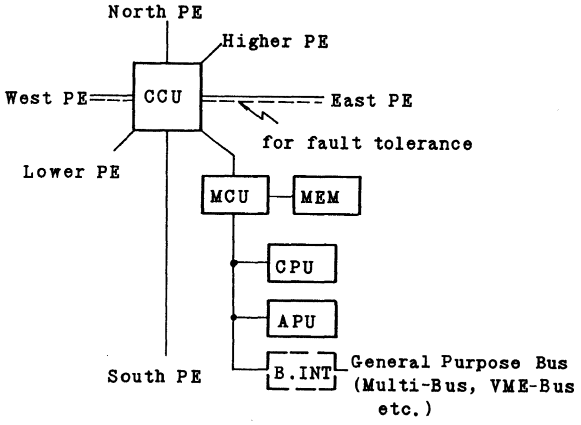


Fig. 6 Configuration of Processing Element (PE)

Table 2 PE Specification

| | | TYPE1 | TYPE2 | TYPE3 |
|---|---|---|---|---|
| CPU | | 80186 | 80286 | 80386 |
| APU | | 8087 | 80287 | 80387 |
| MEM | 256Kb Device | 256KB | 512KB | 1MB |
| | 1Mb Device | — | 2MB | 4MB |
| Interface | in Array | 4bits x (4+2spares) | | |
| Between | Inter-Layer | 4bits x 2 | | |
| PEs | I/O Bus | 16bits x 1 | | 32bits x 1 |
| Performance | General Operation | 0.8 MIPS | 1.6 MIPS | 3.5 ~4 MIPS |
| of PE | Floating Point Op. | 0.04 MFLOPS | 0.08 MFLOPS | 0.45 MFLOPS |
| Data | in Array | 2 MB/S | 2 MB/S | 4 MB/S |
| Transfer | Inter-Layer | 2 MB/S | 2 MB/S | 4 MB/S |
| Rate | I/O Bus | 2 MB/S | 8 MB/S | 32 MB/S |

Data Packet

| header | memory address | data | parity |
|---|---|---|---|

Control Data Packet

| header | control code | PE number (of controled PE) | Control Data | parity |
|---|---|---|---|---|

Fig. 7 Two Types of Packet Format for Data Transfer

5) Communication Control Unit (CCU)

The CCU performs the data transfers between PEs, cPEs and the SMP and provides the control that is needed for them. Packeted imformation (Fig. 7) is asynchronously, time-divided and bidirectionally transfered. Every interface, namely, north, south, east, west, upper and lower, has four data lines with bus structure. Moreover, all the interfaces between the PEs possess a bus-arbiter function so that any PE, cPE or SMP can operate as a master or as a slave. Additionally the CCU also performs the control concerning the synchronization of PEs and the start, stop or other relevant control (when used as a cPE). These controls will be discussed in the next section. The CCU is realized by the same type of gate array as the MCU.

6) Bus Interface (B.INT)

It is used for interfacing general busses such as the multi-bus, the VME bus and others. The cPE is connected to the data I/O mechanism through it.

3.2 Control Method

1) Fundamental Control Scheme

It is necessary to have controls for starting, stoping, data transfer and synchronization of PEs in a parallel processor. These controls are initiated by one instruction to ease program development, and are realized in hardware to provide high speed execution in the HAP. Figure 8 shows the fundamental control scheme where all the control requests of the controlling PE (including the SMP and the cPE) are done in the form of a label access to variables. These become pseudo-memory accesses during processor operation. Based on this access information, the control circuits recognize the control request and generate the necessary control information.

The control information is transfered to the controlled PE (or PEs) through a physical link such as data lines or control lines. The controled PE uses this control information to realize the various control operations through the generation of interrupt vectors, interrupt operations and execution of the required interruption handling routines.

However, controls that demand rapid execution, such as the synchronization of PEs (in this case, the controlling PE is also the controlled PE) are realized by controlling the "Ready" signal of the CPU. Specifically, when synchronization requests are generated by every PE, the CPU is in the wait condition for the "Ready" state. Once the concurrence of synchronization is detected, the "Ready" signal is generated and the CPU is shifted to the execution of the next instruction. With this control, the synchronization of all the PEs is realized in less than one microsecond in the HAP.

2) Flexible Synchronization Mechanism

A flexible synchronization mechanism utilizing the hierarchical structure is realized in the HAP (Fig. 9). It is the sync-mask-register specifying the return of the synchronous signal to its own layer or its propagation to the upper layer. The propagation of synchronous signals is controlled with this register, and the synchronization of all PEs including the cPEs and the SMP, local synchronization of PEs that are connected to the cPE, and other functions are possible. This mechanism is expected to be more useful in some recognition processings (especially pattern matching in their processings), such as character recognition, speech recognition and others where the hierarchical parallel algorithms are often used. Furthermore, the synchronization between neighboring PEs (or cPEs) can easily be realized through the setting and referring of shared varibles such as flags, semaphores and others, since the couplings are via memory, as shown in Fig. 2.

| Function Realized \ Control Level | (Controlling PE)<br>Control Request | PE Start or Stop | (Controlled PE)<br>Synchronization | Data Transfer * Control |
|---|---|---|---|---|
| Software | Label Access | Interruption Handling | Next Process | Interruption Handling |
| Processor Operation | Psuedo Memory Access | Interrupt Operation | Excution of Next Instruction | Interrupt Operation |
| Control Circuit | Decoding of Address or Data | Interrupt Detection | "Ready" Generation | Generation of Interrupt Vector |
| Physical Link | Data Lines<br>Control Lines | Data Lines | Control Lines | |

*Detection of data transfer request from the lower PE

Fig. 8 Fundamental Control Scheme

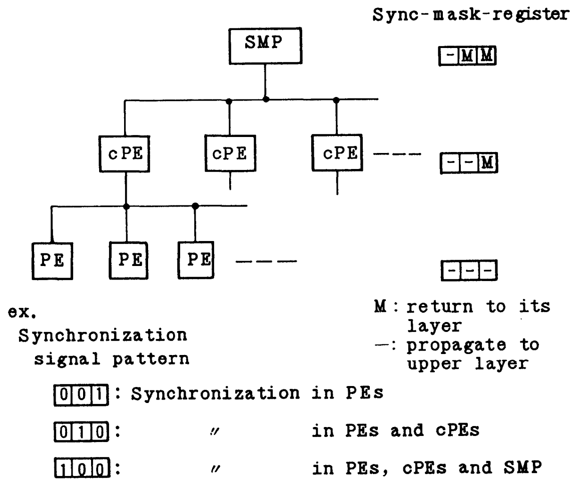

Sync-mask-register
SMP
cPE
cPE
cPE
PE
PE
PE
-MM
--M
---
ex.
Synchronization signal pattern
M : return to its layer
—: propagate to upper layer
001 : Synchronization in PEs
010 : 〃 in PEs and cPEs
100 : 〃 in PEs, cPEs and SMP

Fig. 9 Flexible Synchronization Mechanism

## 4. Performance Estimation

The basic goal in developing the HAP is to realize a system having its performance proportional to the number of its PEs. System performance, $P_H$, is given by the expression,

$$P_H = PoN\alpha,$$

where Po is the processing capability of a single PE, N is the number of PEs and $\alpha$ is the operating efficiency of one PE. $\alpha$ is defined as the ratio of the performance per PE with N PEs operating in parallel to Po. $\alpha$ is dependent on the parallelization algorithm for the problem, with $\alpha = 1$ in the ideal case. The assumptions are that the total processing quantities do not vary with the parallel expansion, the overhead of the parallel processor, i.e., data transfer and synchronization, can be ignored, and the loads are equally divided among all PEs. Under such conditions, the peak system performance of the HAP, using an 80386 and an 80387 as the CPU and the APU respectively, is

$P_H = 4$ MIPS x 4096 x $1 \approx$ 16GIPS
(for fixed point arithemtic or general data processing)

$P_H = 0.45$ MFLOPS x 4096 x $1 \approx 1.8$ GFLOPS
(for floating point arithmetic).

However, $\alpha$ is usually not equal to 1. In the HAP, the factors that lower $\alpha$ from the hardware point of view are sufficiently coped with by the improvement in data transfer capability mentioned in 2.1.2 and by the speed-up in synchronization mentioned in 3.2. An example of this effect is shown in Fig. 10. This is for the case of solving first order simultaneous equations having M unknowns using Gauss' elimination method. The main factor in the lowering of $\alpha$ is the method of supplying initial data (i.e. coefficients of the equations) to the PEs. In the HAP, this is performed in parallel by the cPEs. This improves $\alpha$ up to the solid line shown in the same figure. Hence, the HAP can be expected to have a higher system performance than the usual NNM model on a number of problems, including the aforementioned ones.

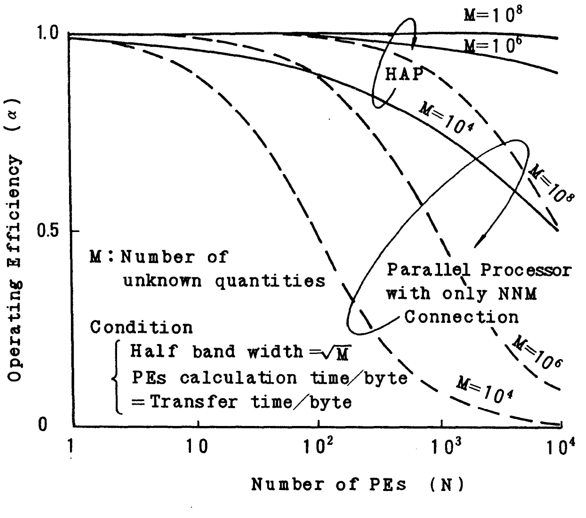


Fig. 10 Operating Efficiency of PE in Solving Simultaneous Equation by Gauss' Elimination Method

## 5. System Auto-Recofiguration

Automatic, speedy system reconfiguration is realized in the HAP to improve its availability and extend its maintenance period. That is, in the HAP, a faulty PE is promptly detected, and an NNM connected array excluding it is automatically reconfigured. This is realized through the use of the newly proposed parallel diagnosis and switching control methods.

### 1) Parallel Diagnosis of PEs

This method employs a decide-by-majority technique using the diagnosis results obtained from neighboring PEs to judge its own condition (called NV diagnosis). Figure 11 shows the concept of NV diagnosis. This diagnosis is carried out on all the PEs simultaneously using the following procedure:

i) The diagnosed PE runs the test program inside itself.

ii) The test result is sent to the 4 neighboring PEs. A comparison between the actual and the expected result is done in both the neighboring PEs and itself.

iii) The neighboring PEs send the compared results back to the diagnosed PE.

iv) Using the results for a decision by majority, the diagnosed PE judges its own condition.

A faulty condition of the diagnosed PE can be known through comparison between the actual and the expected result in the neighboring PEs since the test result will differ from the one expected. Moreover, even if one or two neighboring PEs are faulty, so long as 3 out of 5 (4 neighboring PEs and the diagnosed PE) are in good condition, the faulty condition of the diagnosed PE can be detected through these 3. Consequently, assuming that no more than 2 out of every 5 PEs are faulty, autonomous, parallel diagnosis is possible within the PE array.

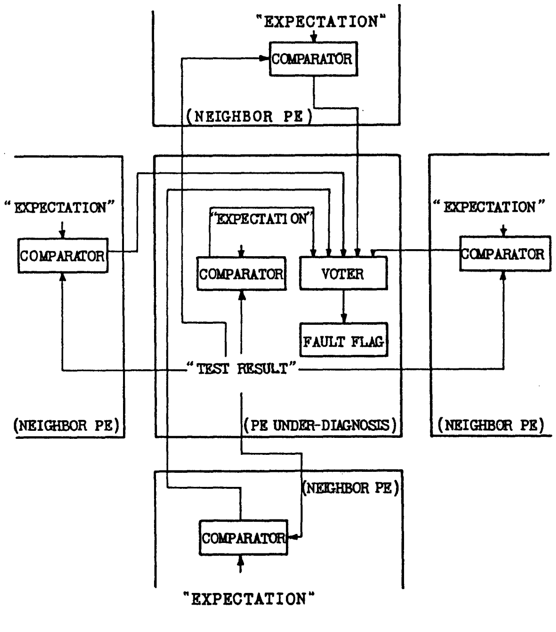


Fig. 11 NV (Neighbor Voting) Diagnosis

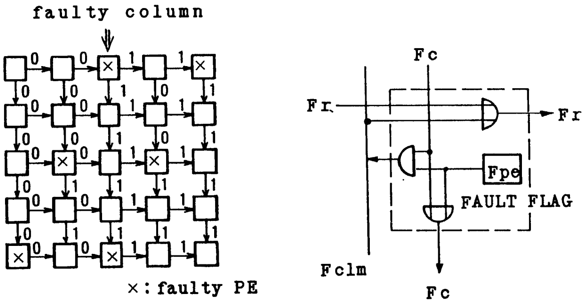
2) Switching Control Method

In the IX net in 2.2, a switching control to exclude the faulty PE and column is necessary (Figs. 4b and c). This control is done as follows:

i) The control signals, Fr and Fc for indicating the existence of a faulty column and a faulty PE, respectively, are transmitted in the manner shown in Fig. 12 a). Figure 12 b) shows the generation circuit for these signals.

ii) The interface direction and bypass of the PE are determined and then the switching is performed in each PE using these signals together with the condition of the PE itself (Fpe) and the condition of its own column (Fclm).

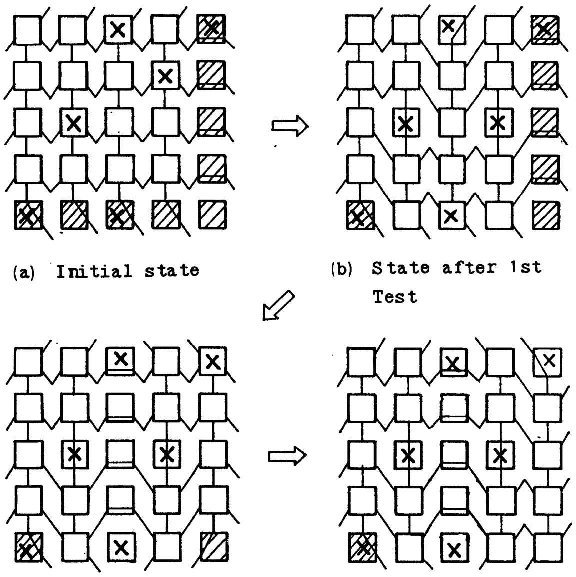
Once a faulty PE is detected, the switching is automatically done within the PE array.



(a) Propagation of Switching Control Signals

(b) Generation Circuit for Switching Control Signals

Fig. 12 Propagation and Generation of Switching Control Signals



(a) Initial state

(b) State after 1st Test

(c) State after 2nd Test

(d) State after 3rd Test

Fig. 13 Auto-Reconfiguration Procedure

3) Auto-Reconfiguration Procedures

The system reconfiguration is done at the time of power on, or during system reset when a PE fails. Initially, all PEs are considered as good and diagnosis of all PEs, except spare ones, is carried out. As a result, if faulty PEs are detected, their flags are set (Fig. 11). Then an NNM connected array, excluding the faulty PEs, is automatically reconfigured by the method in 2). This procedure is executed three times at most before system reconfiguration is completed (Fig. 13). In the HAP, the maintenance period can be extended to the MTTF defined in 2.2 with an availability of nearly 1 by means of this system auto-reconfiguration.

## 6. Programming for HAP

A lot of research [12]-[14] has been done to develop a parallel processing oriented language that makes the abstraction of parallelism in the problem easy. However, the present automatic abstraction of parallelism tends to be at low levels, such as the instruction level or the DO loop level or others. We believe that to achieve an effective improvement in performance, which is the primary goal of a parallel processor, attention must be paid to the higher level of parallelism in the MIMD machine, but that the abstraction of these parallelisms still requires the involvement of man. Consequently, in the HAP, the programmer must be conscious of the parallel structure corresponding to the level of the program written.

1) Software Structure

In order to allow performance improvement' and ease-of-use, and also in consideration of the utilization of the HAP as a back-end processor, the software shown in Fig. 14 is provided. To the HAP user, only writing the program for his computer by utilizing the library program, not being conscious of the HAP hardware, is required. That is, the user's job is expanded to parallel tasks and subrouted at the library level.

Specifically, a parallel processing algorithm suitable to the job is developed at this level, and then the programs for executing the algorithm, divided into a PE program and an SMP program, are written. The main function of the PE program is the parallel alogrithm, and the SMP program is for program load, control of PEs and other functions. When assigning general processing to the cPE, aside from data transfer and synchronization, the writing of a cPE program is necessary too.

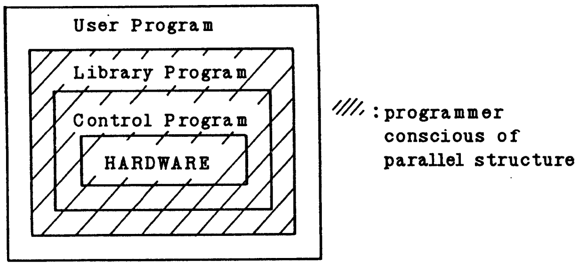
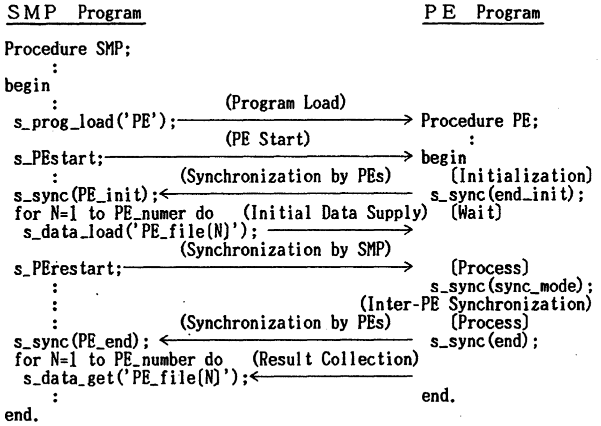


Fig. 14 Software Structure



```
SMP Program                                         PE Program

Procedure SMP;
    :
begin
    :                  (Program Load)
 s_prog_load('PE');─────────────────────────────→ Procedure PE;
                       (PE Start)                      :
 s_PEstart;─────────────────────────────────────→ begin
    :              (Synchronization by PEs)          (Initialization)
 s_sync(PE_init);←───────────────────────────────  s_sync(end_init);
 for N=1 to PE_numer do   (Initial Data Supply)  (Wait)
  s_data_load('PE_file(N)'); ───────────────────→
                   (Synchronization by SMP)
 s_PErestart;───────────────────────────────────→    (Process)
    :                                              s_sync(sync_mode);
    :                                    (Inter-PE Synchronization)
    :              (Synchronization by PEs)          (Process)
 s_sync(PE_end); ←───────────────────────────────  s_sync(end);
 for N=1 to PE_number do   (Result Collection)
  s_data_get('PE_file(N)');←─────────────────────
    :                                             end.
end.
```

Fig. 15 Example of Program Description (PASCAL)

However, these programmings are easy (Fig. 15) since these can use the subrouted control programs, and the fundamental controls are described by single instructions, as mentioned in 3.2. The control programs are written as to the various controls mentioned in 3.2.

2) Program Development Environment

In consideration of library program development using a general computer (not parallel), procedure calls and function calls are used instead of macro instructions in order to reduce dependency on the machine and language used.

For programming, high level languages such as PASCAL, C, Ada, FORTRAN and others can be used without any modification.

## 7. Conclusion

The configuration of a MIMD type highly parallel processor, the HAP, with 4096 PEs is discussed. The HAP appears capable of ensuring practical reliability and data transfer capability that matches the processing capability of its PEs, which are considered the main problems in high parallelization.

The problems of data transfer are solved by adopting a hierarchical PE array structure, namely, a large scale array with a small scale array above it, and utilizing the latter to transfer data. Specifically, the small PE array is used to parallel the data transfer to and from the large PE array and to reduce the inter-PE data transfer delay by relaying it. With these approaches, the HAP is expected to attain a system performance near the peak value of 16 GIPS, 1.8 GFLOPS (in the case of using 80386 and 80387 processors) over a broad range of applications.

Reliability problems are solved with the application of a fault-tolerant configuration where one row and one column of spare PEs are provided to each n x n PE array and automatic switching is done in PE and column units. Also, it applies a newly proposed parallel diagnosis method, called NV diagnosis, that uses neighboring PEs. Fast, automatic system reconfiguration is realized through these features, and thereby the maintenance period (equal to the MTTF) is about 1 year (with a PE failure rate of $10^3$ Fit) with an availability of approximately 1.

A small scale version with 256 PEs and 16 cPEs is under fabrication. Each PE consists of 13 LSIs, namely 80186 and 8087 type processors, 9 DRAMs (256kbits/chip) and 2 gate arrays (PE size 9cm x 6cm x 3cm). Testing of its capability for general scientific calculation and various types of recognition together with an overall evaluation is scheduled.

## Reference

[1] Charles L. Seitz "The Cosmic Cube" Commun. ACM 1 (January 1985), 22-33

[2] H. F. Jordan "Performance Measurements on HEP - A Pipelined MIMD Computer" The 10th Annual Symp. on Computer Architecture, (1983), 207-212

[3] S. J. Stolfo and D. P. Miranker "DADO : A Parallel Processor for Expert Systems" 1984 Int. Conf. Parallel Processing, (August 1984), 74-82

[4] T. Hoshino, T. Kawai, T. Shirakawa, K. Higashino, A. Yamaoka, H. Ito, T. Sato and K. Sawada "PACS: A Parallel Microprocessor Array for Scientific Calculations" ACM Trans. Computer System Vol. 1, No. 3, (August. 1983), 195-221

[5] D. Gajski, D. Kuck., D. Lawrie and A. Sameh "Cedar - A Large Scale Multiprocessor" 1983 Int. Conf. Parallel Processing, (August 1983), 524-529

[6] L. Snyder "Introduction to the Configurable, Highly Parallel Computer" Computer, (January 1982), 47-56

[7] L. D. Wittie, "Communication Structures for Large Networks of Microcomputers" IEEE Trans. Computer, vol. C-30, NO.4, (April 1981), 264-273

[8] M. Vajteršie "Parallel Poisson and Biharmonic Solvers Implemented on the EGPA Multiprocessor" 1982 Int. Conf. Parallel Processing (August 1982), 72-81

[9] T. Hoshino, T. Shirakawa, T. Kamimura, T. Kageyama, K. Takenouchi and H. Abe "Highly Parallel Processor Array "PAX" for Wide Scientific Applications" 1983. Int. Conf. Parallel Processing, (August 1983), 95-105

[10] M. Sami and R. Stefanelli "Reconfigurable architecture for VLSI processing arrays" National Computer Conference, (1983) 565-577

[11] G. Osawa, H. Amano and H. Aiso "HOBONET: An Inter-PU Connection Network with Fault-Tolerancy" 1984 Int. Conf. Parallel Processing, (August 1984), 165-168

[12] C. A. R. Hoare "Communicating Sequential Processes" Commun. ACM 8 (August 1978), 666-677

[13] P. B. Hansen "Distributed Processes: A Concurrent Programming Concept" Commun. ACM 1 (November 1978), 934-941

[14] E. Shapiro "System Programming in Concurrent PROLOG" ACM Symp. on Principles of Programming Languages, (January 1984), 93-105

# MESH-CONNECTED COMPUTER ALGORITHMS FOR RECTANGLE-INTERSECTION PROBLEMS

**Mi Lu**
**Peter Varman**

Department of Electrical and Computer Engineering
Rice University

## Abstract

In this paper, Mesh-Connected Computer (MCC) algorithms for computing several properties of a set of, possibly intersecting rectangles are presented. Given a set of $n$ iso-oriented rectangles, we describe MCC algorithms for determining following properties: (i) The area of the logic "OR" of these rectangles (i.e. the area of the region covered by at least one rectangle). (ii) The area of the logic "AND" of the rectangles (i.e. the area of the region covered by two or more rectangles). (iii) The largest number of rectangles that overlap. This solves the fixed-size rectangle placement problem, i.e. given a set of plan points and a rectangle, find a placement of the rectangle in the plane so that the number of points covered by the rectangle is maximal. (iv) The minimum separation between any pair of a set of non-overlapping rectangles. All these algorithms can be implemented on a $\sqrt{n}\times\sqrt{n}$ MCC in $O(\sqrt{n})$ time. The best known algorithms for the above problems are sequential and have optimal $O(n \log n)$ time complexity.

## I. Introduction

A two-dimensional Mesh-Connected Computer (MCC) consists of a number of identical processors arranged in a two-dimensional array with interconnections between every pair of horizontally and vertically adjacent processors. Each processor has a fixed number of registers and is capable of performing arithmetic and boolean operations. The MCC operates as a single-instruction stream, multiple-data stream (SIMD) computer in which several processors execute the same instruction in parallel on different data items. The simplicity of the inter-processor communication pattern and the economical layout afforded by an MCC have resulted in actual implementations in recent years [1,2,3].

Several MCC algorithms for problems in diverse computational areas have been discovered. Thompson and Kung [4] and Nassimi and Sahni [5] presented fundamental MCC algorithms for sorting, which form the basis of data routing techniques [6]. Algorithms for solving matrix computations [7], solving graph-theoretic problems [8,9,10,11,12,13,14] and more recently for problems in computational geometry [15,16] have also been presented.

In this paper we present MCC algorithms for computing several interesting properties of a set of $n$, possibly overlapping, iso-oriented rectangles. (An iso-oriented rectangle is one whose sides are parallel to the coordinate axes). These problems belong to the class of rectangle intersection problems in computational geometry that has been widely studied for its applications in VLSI design [17], computer graphics and architectural data bases [18]. The specific problems that we cover are as follows.

Given a set of $n$ iso-oriented rectangles determine:
1. The area of the logic "OR" of these rectangles, that is the area of the region that is covered by at least one rectangle.
2. The area of the logic "AND" of these rectangles, that is the area of the region that is covered by two or more rectangles.
3. The maximal number of rectangles that overlap. This also solves the fixed rectangle placement problem, which asks one to determine the placement of a given rectangle in the plane so that it includes the largest number of a set of given planar points.
4. If the rectangles are non-intersecting, the distance between the closest pair of rectangles.

The algorithms presented in this paper employ a divide and conquer technique, based on the "separational principle" for planar objects proposed by Güting [19], and require $O(\sqrt{n})$ time on a $\sqrt{n}\times\sqrt{n}$ MCC. Upto constant factors, our algorithms are optimal on the "standard" MCC model being used, and compare favorably with the known sequential algorithms which have $O(n \log n)$ time complexity.

In sections II through V of the paper we present our algorithms for the problems mentioned earlier. In the discussion we assume familiarity with techniques for sorting a set of records distributed evenly among the processors (see [4]) and for performing parallel random access reads (RAR) or writes (RAW) on an MCC (see [6]).

## II. "OR" Area Reporting

A rectangle with its sides parallel to the coordinate axes is called an iso-oriented rectangle. Given a set of $n$ iso-oriented rectangles, the "OR" area reporting problem is to find the area of the region that is covered by one or more rectangles.

Consider the $2n$ vertical segments that make up the $n$ rectangles. The set of horizontal lines passing through the points at the bottom and the top of those segments partitions the plane into horizontal strips. Each strip is bounded in the vertical dimension by two horizontal rectangle edges (not necessarily belonging to the same rectangle), and each horizontal edge of a rectangle coincides with a strip's boundary. The area of the region covered by the logic "OR" of the set of rectangles is the sum of the covered area in each strip. The y-interval covered by each strip is simply the difference in the y-coordinates of the horizontal lines bounding the strip; so what we are interested in is only the x-interval covered by each strip. We solve the problem efficiently by using a divide-and-conquer approach as described below.

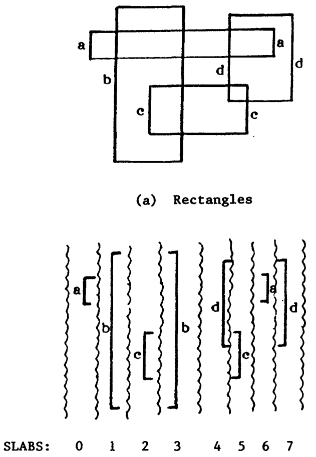
Sort all the vertical segments by their x-coordinates in non-decreasing order. Divide the plane into two slabs by a vertical line, $L$, that partitions the set of vertical segments into two equal-sized subsets so that every segment to the left (respectively right) of the dividing line $L$, has an x-coordinate less than (respectively greater than or equal to) the x-coordinate of $L$.

Iteratively partition each slab obtained into a left and right slab in a similar manner, until each slab finally contains one vertical segment. Each slab (except for the two at the end) is bounded by two dividing lines, which we refer to as the *left boundary* and *right boundary* of the slab. Assume that the *left boundary* of the leftmost slab to be a line passing through the left most segment

This research was partially supported by an IBM Faculty Development Award.

0190-3918/86/0000/0301 $01.00 © 1986 IEEE

and the *right boundary* of the rightmost slab to be a line passing through the rightmost segment. Figure 1 illustrates an example of the partitioning for eight vertical segments.



(a) Rectangles

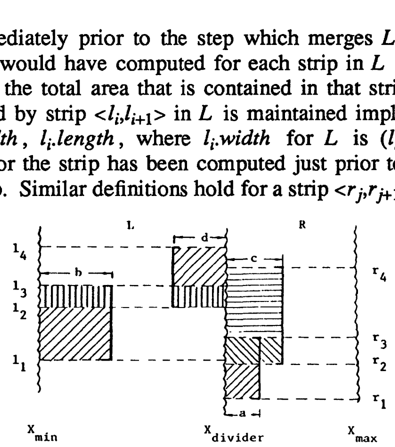
(b) Partitioning of Vertical Segments into Slabs.

Figure 1

The algorithm proceeds to merge together adjacent slabs in a binary tree fashion. All merges at a level of the computation tree are done in parallel, and the computation is completed after $\log_2 n$ iterations. Every merge step merges together slabs that lie to the left and right respectively of a common dividing line. Let us denote the left and right slabs being merged as $L$ and $R$ respectively. Let $l_1, l_2, ..., l_k, l_i \le l_{i+1}$, be the y-coordinates of the segments in $L$ and $r_1, r_2, ..., r_k, r_i \le r_{i+1}$, the y-coordinates of the segments in $R$.

The horizontal lines through $l_i$, $i$=1, ..., $k$, partition the slab $L$ into $k$ horizontal *strips*, where the $i$-th strip of $L$, $1 \le i \le k-1$, denoted by $<l_i, l_{i+1}>$ consists of the region between $l_i$ and $l_{i+1}$ and the $k$-th string (dummy) has width zero. In a similar fashion, the horizontal lines through $r_i$, $i$=1, ..., $k$, partition $R$ into $k$ horizontal strips.

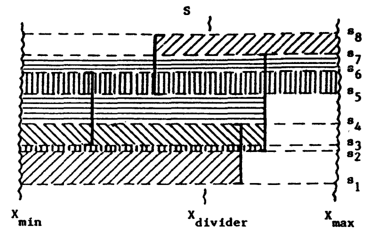
Immediately prior to the step which merges $L$ and $R$, the algorithm would have computed for each strip in $L$ (and $R$), the portion of the total area that is contained in that strip. The area contributed by strip $<l_i, l_{i+1}>$ in $L$ is maintained implicitly by the pair $l_i$*.width*, $l_i$*.length*, where $l_i$*.width* for $L$ is $(l_{i+1} - l_i)$ and $l_i$*.length* for the strip has been computed just prior to the current merge step. Similar definitions hold for a strip $<r_j, r_{j+1}>$ in $R$.



Note: Shaded areas represent the area of the strip that is part of the total area.

| | |
|---|---|
| $l_1$.length = b | $r_1$.length = a |
| $l_2$.length = b+d | $r_2$.length = c |
| $l_3$.length = d | $r_3$.length = c |
| $l_4$.length = 0 | $r_4$.length = 0 |

Fig. 2(a) Prior to merge of L and R



| Left Originating Strips | Right Originating Strips |
|---|---|
| $<s_3, s_4>$ | $<s_1, s_2>$ |
| $<s_5, s_6>$ | $<s_2, s_3>$ |
| $<s_6, s_7>$ | $<s_4, s_5>$ |
| | $<s_7, s_8>$ |

| Left Open Strips | Right Open Strips |
|---|---|
| $<s_1, s_2>$ | $<s_5, s_6>$ |
| $<s_2, s_3>$ | $<s_6, s_7>$ |
| $<s_3, s_4>$ | $<s_7, s_8>$ |
| $<s_4, s_5>$ | |
| $<s_5, s_6>$ | |
| $<s_6, s_7>$ | |

Fig. 2(b) Following the merge of L and R into S.

Figure 2 illustrates a possible situation where $k$=4. The slabs $L$ and $R$ are merged to form a slab $S$, bounded on the left by the left boundary of $L$ with x-coordinate $x_{min}$ and on the right by the right boundary of $R$ with x-coordinate $x_{max}$.

Slab $S$ contains all the line segments in $L$ and $R$. Let $s_i$, $i$=1, ..., $2k$, $s_i \le s_{i+1}$, be the ordered list of y-coordinates of segments in $S$. Obviously, $s_1, ..., s_{2k}$ is obtained by merging together the ordered lists of points $l_1, ..., l_k$ and $r_1, ..., r_k$ corresponding to segments in $L$ and $R$ respectively. That is, the strips are *refined* in the merge step. To simplify the presentation of the merging algorithm, we present some definitions. The strip $<s_i, s_{i+1}>$ of $S$, which lies between the horizontal lines passing through $s_i$ and $s_{i+1}$, is called a *right_originating strip* (respectively *left_originating strip*) if $s_i = r_j$ (respectively $s_i = l_j$), for some $j$, $1 \le j \le k$. If $u$ and $v$ are vertical segments of the same rectangle, then $u$ ($v$) is a *partner* of $v$ ($u$). If the x-coordinate of $u$ is less (respectively greater) than that of its *partner* $v$, then $u$ is a left (respectively right) segment of the rectangle. A segment in $S$ is said to be a *left_open segment* if it is the right segment of a rectangle, lies in $R$ and does not have a left partner in $S$. A segment in $S$ is said to be a *right_open segment* if it is the left segment of a rectangle, lies in $L$ and does not have a right partner in $S$. A strip $<s_i, s_{i+1}>$ is a *left_open* (respectively *right_open*) *strip* if it is crossed by a *left_open* (respectively *right_open*) *segment.*

At the end of the step that merge $L$ and $R$, we should determine $s_i$*.length*, the length of the strip $<s_i, s_{i+1}>$ that is to be included in the overall area. This is computed by determining $(s_i.length)_L$ and $(s_i.length)_R$, which are the portions of $s_i$*.length* that lie to the left and right of the dividing line respectively, and then taking their sum.

The algorithm which computes $s_i$*.length* for each $s_i$, $i=1, ..., 2k$ is presented in Algorithm 1.

*Algorithm 1*:

```
for each strip $<s_i, s_{i+1}>$, $i=1, ..., 2k-1$, do
if $<s_i, s_{i+1}>$ is a left_originating strip, where $s_i = l_j$
  then
    begin
      if $<s_i, s_{i+1}>$ is a left_open strip, then
        /* the entire portion of the strip in L must be included */
          $(s_i.length)_L = x_{divider} - x_{min}$
      else /* no change in the portion covered in L */
```

```
          $(s_i.length)_L = l_j.length$
        if $<s_i,s_{i+1}>$ is a right_open strip, then
          /* the entire portion of the strip in $R$ must be included */
          $(s_i.length)_R = x_{max} - x_{divider}$
        else /* no change in the portion covered in $R$ */
(**)      /* Determine the strip $<r_p,r_{p+1}>$ in $R$ that previously
           covered the region now covered by $<s_i,s_{i+1}>$ */
        /* $r_p$ is the point in $r_1, ..., r_k$ such that
          $r_p < s_i < r_{p+1}$ */
          $(s_i.length)_R = r_p.length$
      end
  else /* $<s_i,s_{i+1}>$ is a right_originating strip, where
      $s_i = r_j$ */
      begin
        if $<s_i,s_{i+1}>$ is a right_open strip then
          $(s_i.length)_R = x_{max} - x_{divider}$
        else
          $(s_i.length)_R = r_j.length$
        if $<s_i,s_{i+1}>$ is a left_open strip then
          $(s_i.length)_L = x_{divider} - x_{min}$
        else
         /* let $l_p$ be the point in $l_1, ..., l_k$
          such that $l_p < s_i < l_{p+1}$ */
          $(s_i.length)_L = l_p.length$
      end
  $(s_i.length) = (s_i.lengh)_L + (s_i.length)_R$
end
```

We now describe the MCC algorithm for the OR-Area reporting problem. The PEs are assumed to be indexed in shuffled row-major order [4].

*MCC Algorithm*
/* Initialization */

1. Sort the vertical segments by their x-coordinates into non-decreasing order on the mesh in shuffled row-major order.

   Each PE contains one segment.

2. /* Determine left and right boundaries of current merge regions */

```
PE(i) executes:
if $i$ is even  $x_{max}$ = x-coordinate of segment in PE(i+1)
                $x_{min} = x$ /* x-coordinate of segment in PE(i) */
if $i$ is odd   $x_{min}$ = x-coordinate of segment in PE(i-1)
                $x_{max} = x$ /* x-coordinate of segment in PE(i) */
```

3. /* initialize length covered by each segment */

```
/* PE(i) maintains the strips corresponding to the top $y_t$ and
  bottom $y_b$ points of the segment it contains */
           $y_t.length = 0$ /* dummy strip */
if left segment then  $y_b.length = x_{max} - x$
                else  $y_b.length = x - x_{min}$
```

We now present details of the MCC implementation of the merge step corresponding to Algorithm 1.

Immediately prior to the merge step, points corresponding to segment in $L$ and $R$ are available in separate adjacent submeshes, two points per PE, sorted by y-coordinates.
Each point has with it the following information:

(i) the length of strip covered by it, in variable *length*;

(ii) a flag *bottom/top* indicating if it is the top or bottom point of a vertical segment;

(iii) a flag *left/right* indicating if it is the left or right segment of a rectangle;

(iv) a flag *full* indicating if the partner of the edge has been found or not;

(v) the co-ordinates of the point;

(vi) the id. of the rectangle to which the point belongs.

Each point also has the following global information:

(i) $x_{divider}$, $x_{min}$ and $x_{max}$.

(ii) the variable *local_rank* which is the position of the point in the sorted (by y-coordinates) list of points in its half.

Steps executed by a PE:

1. Merge the sorted lists of points in the two submeshes into a single sorted list, (using the y-coordinates as the key,) into non-decreasing row-major order. Ties are broken by the id. of the rectangle to which the point belongs. This ensures that points corresponding to segments that are partners will be adjacent in the sorted list. Let *global_rank* denote the position of a point in the merged list.

2. Each PE checks if the point adjacent to a point within it in the merged list has the same rectangle id. If so it sets *full* to 1.

3. Determine whether each strip in the merged list is *left_open* or *right_open*. We explain the steps needed to determine if it is *left_open* (symmetrical steps will determine if the strip is *right_open*). Basically, to determine if $<s_i,s_{i+1}>$ is *left_open* we count the number of right segments in $R$ whose bottom y-coordinate is less than $s_i$ and top y-coordinate exceeds $s_i$.

   Every point that is the top (respectively bottom) of a *left_open* segment (true *iff full* = 0, it belongs to $R$ and is a right segment) sets a local counter, count to 1 (respectively -1). By finding the sum of all "counts" that precede the point, every point can determine whether it lies in a *left_open* segment or not (if the sum is greater than 0, it does, else it does not).

   Since the points are sorted in row-major order, this is easily accomplished by a row-sweep that obtains the sum of counts within a row, followed by a column-sweep, which determines the sum up to the beginning of a row, followed by a final row-sweep, which distributes this partial sum to each PE in that row.

4. Execute the steps described in Algorithm 1.

   The only information that is not locally available to a PE corresponds to the case marked by a "**" in Algorithm 1. However, each PE can in constant time determine the index of the PE which contains the desired information as follows. Since *local_rank* is the position of the point among points only in its sorted list (say $A$) and *global_rank* is the position of the point in the list obtained by merging lists $A$ and $B$, the difference (actually *global_rank - local_rank - 1*) gives the position of the desired point in the sorted list $B$. By performing one RAR from the PE with index (*global_rank - local_rank - 1 + Base* address of the block of PEs containing the sorted list $B$), a PE can obtain the required information. (We presented a detailed description of such a scheme for other MCC algorithms earlier in [16].)

The time needed to find the area of the "OR" of a set of $n$ iso-oriented rectangles on a $\sqrt{n}\times\sqrt{n}$ MCC is $O(\sqrt{n})$. The sorting in the initialization step can be done in $O(\sqrt{n})$ time. Every merge of two sets of $k$ vertical segments takes place in a sub mesh of size no larger than $2\sqrt{k} \times 2\sqrt{k}$. Step 1, the mergings of two sorted lists, can be done in $O(\sqrt{k})$ time. Row and column sweeps required in Step 3 also take no more than $O(\sqrt{k})$ time. Finally, the RAR required in Step 4 also can be done in $O(\sqrt{k})$ time. The total time, $T(n)$ is therefore $T(n) \le c(\sqrt{n} + \sqrt{n}/2 + \sqrt{n}/4 + ... + 1)$

which is $O(\sqrt{n})$.

## III. "AND" Area Reporting

The "AND" of two rectangles $A$ and $B$ is a third rectangle which includes the region that is overlapped by $A$ and $B$. Given a set of $n$ iso-oriented rectangles, the "AND" area reporting problem is to find the area of the region which is covered by at least two rectangles.

We present an algorithm to solve this problem on a $\sqrt{n}\times\sqrt{n}$ MCC in $O(\sqrt{n})$ time. The algorithm follows the same general principle of the algorithm for reporting the OR-area detailed in section II, differing in the merge step which combines vertical segment on opposite sides of a dividing line. In fact, this algorithm may be considered a generalization of the "OR" problem, in that the solution to the latter is also obtained simultaneously.

Consider a strip $<l_i,l_{i+1}>$ in $L$, the region to the left of the dividing line currently being merged as in section II.

Assume that up to now the algorithm has computed the two quantities $l_i.length$ and $l_i.overlap$ for each strip $<l_i,l_{i+1}>$ in $L$. $l_i.length$ is the quantity computed by Algorithm 1 and represents the region in $<l_i,l_{i+1}>$ covered by at least one rectangle. $l_i.overlap$ represents the region in $<l_i,l_{i+1}>$ covered by two or more rectangles (i.e. the overlapped area) and has been computed in the previous iteration.

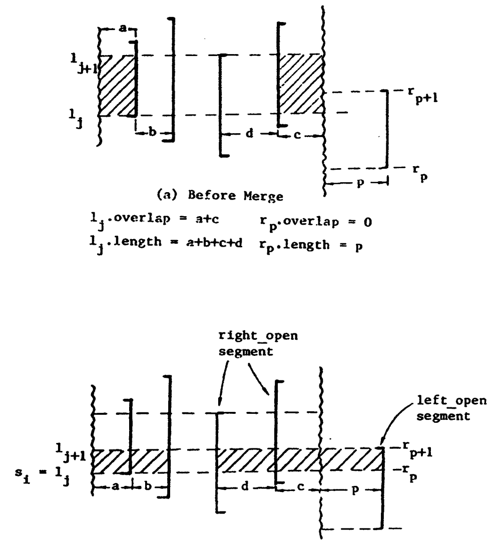
In a similar manner, the corresponding quantities $r_j.length$ and $r_j.overlap$ for all strips $<r_j,r_{j+1}>$ in $R$ have been computed.

The merging of regions $L$ and $R$ partitions the merged region $S$ into strips $<s_i,s_{i+1}>$. Algorithm 2 below presents the computation of $s_l.overlap$ using the values of $l_i.overlap$, $l_i.length$, $r_j.overlap$, $r_j.length$ already computed.

*Algorithm 2*

```
   for each <$s_i$,$s_{i+1}$> in parallel do
     if <$s_i$,$s_{i+1}$> is a left_originating strip where $s_i = l_j$
     then begin
          if <$s_i$,$s_{i+1}$> is a left_open strip then
            /* entire portion of strip <$l_j$,$l_{j+1}$> that was part
              of some rectangle is now in the overlapped area */
(1)            $(s_i.overlap)_L = l_j.length$
          else /* no change in overlapped area L */
(2)            $(s_i.overlap)_L = l_j.overlap$
          if <$s_i$,$s_{i+1}$> is a right_open strip then
            /* Let <$r_p$,$r_{p+1}$> be the segment in R that is now
              crossed by <$s_i$,$s_{i+1}$> */
(3)            $(s_i.overlap)_R = r_p.length$
          else
(4)            $(s_i.overlap)_R = r_p.overlap$
        end
     $s_i.overlap = (s_i.overlap)_L + (s_i.overlap)_R$
     else /* <$s_i$,$s_{i+1}$> is a right_originating strip */
       begin
          /* symmetrical code */
       end
end
```

Note that we also need to update $s_i.length$ during this merge step, using Algorithm 1. Fig. 3 shows the situation corresponding to (1) and (3) of Algorithm 2.



(a) Before Merge

$l_j.overlap = a+c$ $\quad$ $r_p.overlap = 0$

$l_j.length = a+b+c+d$ $\quad$ $r_p.length = p$

(b) After Merge

Case 1: $(s_i.overlap)_L = a+b+c+d$

Case 3: $(s_i.overlap)_R = p$

$(s_i.overlap) = a+b+c+d+p$

Figure 3

The MCC algorithm is similar to that in section II, and computes the AND-area in $O(\sqrt{n})$ time.

## IV. Maximum Overlap Problem

Given a set of $n$ iso-oriented rectangles, the maximum overlap problem is to determine the largest number of rectangles that overlap, i.e. include a common region. We present below an $O(\sqrt{n})$ time algorithm to solve the problem on a $\sqrt{n}\times\sqrt{n}$ MCC. At the expense of some additional book keeping, the algorithm can be modified to report a region of maximal overlap as well.

Just prior to the merge step that we are about to describe, each strip $<l_i,l_{i+1}>$ in $L$ has computed $l_r.max\_overlap$, equal to the maximum number of rectangles that overlap some region in strip $<l_i,l_{i+1}>$. Similarly, each strip $<r_j,r_{j+1}>$ in $R$ has computed $r_j.max\_overlap$. To determine the maximum number of overlapping rectangles that overlap a strip $<s_k,s_{k+1}>$ in the merged region $S$, we determine two quantities viz. $(s_k.max\_overlap)_L$ and $(s_k.max\_overlap)_R$ which are the maximum number of rectangles overlapping a point in strip $<s_k,s_{k+1}>$ that lies in $L$ or in $R$ respectively. $s_k.max\_overlap$ is then the larger of these two computed quantities.

Algorithm 3 below presents the details of the merge for a *left_originating strip* $<s_k,s_{k+1}>$. A symmetrical set of steps would apply if $<s_k,s_{k+1}>$ was a *right_originating strip.*

*Algorithm 3*

```
/* assuming <$s_k$,$s_{k+1}$> is a left_originating strip, $s_k$=$l_j$ */
1.   Find the number of left_open segments crossing <$s_k$,$s_{k+1}$>.
     Denote this by $n_L$
```

Each such *left_open segment* (which by definition belongs to $R$) must overlap all rectangles crossing $<l_j,l_{j+1}>$. Thus the maximum number of overlapping rectangles in the portion of $<s_k,s_{k+1}>$ that lies in $L$ is computed as follows.

2. $(s_i.max\_overlap)_L = (l_j.max\_overlap) + n_L$;
3. Find the number of *right_open* segments crossing $<s_k,s_{k+1}>$. Denote this by $n_R$. This region prior to the merge lay in strip $<r_p,r_{p+1}>$, where $r_p \leq l_j < r_{p+1}$.
4. $(s_i.max\_overlap)_R = (r_p.max\_overlap) + n_R$;
5. $s_i.max\_overlap = Max\,[(s_i.max\_overlap)_L,(s_i.max\_overlap)_R]$;

At the end of the last merge step, the algorithm has computed for each strip, the maximum number of rectangles overlapping within that strip. The algorithm is completed by finding the largest of these quantities over all the strips. As before the time complexity of the algorithm is $O(\sqrt{n})$ on $\sqrt{n}\times\sqrt{n}$ mesh.

An related problem in computational geometry is the fixed size rectangle placement problem. Given $n$ points in the plane, and an iso-oriented rectangle of fixed size, the problem is to find a placement of the rectangle in the plane so that the number of points covered by the rectangle is maximized. This problem is equivalent to the maximum overlap problem. Let the center of the rectangle be the intersection of its two diagonals. Generate $n$ iso-oriented rectangles of the size of the given rectangle and let each be centered at one of the given points. Find the region of maximum overlap for the rectangles, i.e. find the common region which is covered by the largest number of rectangles. Note that if a rectangle $A$ covers rectangle $B'$s center then rectangle $B$ can certainly cover $A'$s center since $A$ and $B$ are of the same size (see Fig. 4). The given rectangle should be placed centered at a region of maximal overlap.

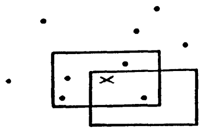


Figure 4

## V. Minimum Separation Between Rectangles

Given a set of $n$ non-overlapping iso-oriented rectangles, the minimum separation problem is to determine the distance between the closest pair of rectangles. The distance between two rectangles is defined to be the minimal (Euclidean) distance between all pairs of points, one belonging to each rectangle. If $p$ and $q$ are points on two non-overlapping rectangles $P$ and $Q$ respectively, then there are only three possibilities by which $p$ and $q$ can determine the distance between $P$ and $Q$, namely :

(i) $p$ lies on the left (right) vertical segment of $P$ and $q$ on the right (left) vertical segment of $Q$, and $p$ and $q$ have the same y-coordinate.

(ii) $p$ lies on the top (bottom) horizontal segment of $P$ and $q$ on the bottom (top) horizontal segment of $Q$, and $p$ and $q$ have the same x-coordinate.

(iii) $p$ is the northeast (northwest, southeast, southwest) corner of $P$ and $q$ is the southwest (southeast, northwest, northeast) corner of $Q$. In this case no points of $P$ and $Q$ have either a common x- or a common y-coordinate.

We refer to the distance between $P$ and $Q$ defined by the each of the three cases above as the horizontal, vertical and diagonal separation respectively. The minimum separation between the rectangles is the smallest of the horizontal, vertical and diagonal separations between all pairs of rectangles for which the corresponding separation is defined. Our algorithm to compute the minimum separation uses a divide-and-conquer strategy along the basic lines of the previous algorithms.

Assume inductively, that we have computed $\delta_L$ and $\delta_R$, the minimum separation between all pairs of rectangles (possibly incomplete) in the left and right regions, $L$ and $R$, being merged. In the merge step, we compute $\delta_S$, the minimum separation between rectangles in region $S$ as follows: Let $\delta = Min[\delta_L,\delta_R]$. We determine whether (a) the horizontal separation, (b) the vertical separation or (c) the diagonal separation between any pair of rectangles in $S$ is less than $\delta$. If so we update $\delta_S$ to the smallest of the three values computed above, else $\delta_S = \delta$. We outline the main steps in the algorithm.

### Horizontal Separation

We need to identify pairs of segments, one in $L$ and the other in $R$, which might have a horizontal separation less than $\delta$. Since the rectangles are non-overlapping, the only possibility for this occurs if a right vertical segment in $L$ and a left vertical segment in $R$ intersect a common horizontal strip, and the segment in $L$ (respectively $R$) is the rightmost (respectively leftmost) segment crossing that common strip. (See Fig. 5(a).)

To implement the merge step, we therefore need to maintain for each strip in $L$ (respectively $R$) information about the rightmost (respectively leftmost) edge in the strip. If in a refined strip we find that the rightmost segment in $L$ and the leftmost segment in $R$ crossing that strip are left-facing and right-facing respectively, we computes the horizontal separation between the two rectangles equal to the difference in the x-coordinates of the segment in $R$ and the segment in $L$. To complete the merge, the rightmost (respectively leftmost) segment in the refined strip is updated to be the rightmost (respectively leftmost) segment that crossed the strip in $R$ (respectively $L$). Finally, determine $\delta_h$, the minimum of the horizontal separations computed for each strip.

Set $\delta_S = Min[\delta,\delta_h]$. It is easy to see that these steps may be implemented on an MCC in a manner similar to the previous three algorithms.

### Vertical Separation

We need to identify pairs of segments one in $L$ and the other in $R$, which define two rectangles whose vertical separation is less than $\delta$.

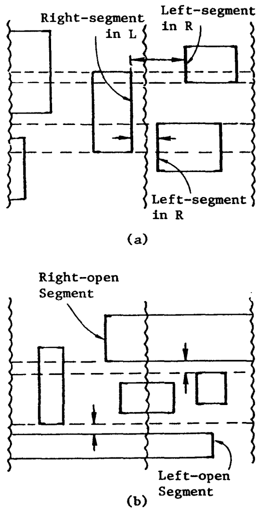


Figure 5

Observe first that, since the rectangles are non-overlapping any strip must be bounded by either the top and bottom horizontal edges of the same rectangle or by a pair consisting of the top and the bottom edges respectively of two different rectangles. A strip of the second type can influence the minimum vertical separation if and only if the horizontal edges of the two rectangles that bound the strip share a common x-coordinate. Consider a strip $<s_i,s_{i+1}>$ in the merged region $S$, where $s_i = l_j$ and $s_{i+1} = r_k$. The strip defines a vertical separation of $(s_{i+1}-s_i)$ between two rectangles if and only if either $l_j$ is a *right_open segment*, or $r_k$ is a *left_open segment* (see Fig. 5(b)). A symmetrical set of conditions holds if $s_i = r_j$ and $s_{i+1} = l_k$.

In the merge step of the algorithm, for each refined strip $<s_i,s_{i+1}>$ we determine whether it satisfies the conditions stated above. If so, we determine the vertical separation in the strip to be equal to $s_{i+1} - s_i$. Finally, update $\delta_S = Min[\delta_S,\delta_v]$. The implementation of this step on the MCC is simpler than all other cases considered so far, and does not require any random-access read. As before $\delta_v$ is the minimum of these computed values for all the strips.

Finally, we present the algorithm for examining the diagonal separation between the rectangles.

Diagonal Separation

We need to identify pairs of segments one in $L$ and the other in $R$, which define two rectangles whose diagonal separation is less than $\delta$.

In the merge step we check whether any northeast (respectively southeast) point in $L$ is within a distance $\delta$ of a southwest (respectively northwest) point of $R$. A crucial observation that enables efficient implementation of the step is that all the points of the same type (e.g. all the southwest points in $R$ or all the northeast points in $L$) obey the *sparsity* restriction, in that the distance between any two such points will be greater than $\delta$. This follows because the distance between two such points must always exceed the minimum separation between the rectangles computed so far which in turn is at least $\delta$ (see Fig. 6). Let us focus only on the northeast points in $L$ and the southwest points in $R$. (Similar considerations hold for southeast points in $L$ and northwest points in $R$.) We need to consider only those northeast points in $L$ and those southwest points in $R$ that lie within $\delta$ of the dividing line and determine if any such pairs of points one from $L$ and one from $R$, are closer than $\delta$ to each other. Since all the points under consideration in $R$ and $L$ are at least $\delta$ apart, for the density of planar point packing [20], it follows that for any northeast point in $L$, there can be at most a constant number (four in this case) of southwest points in $R$ which can be closer to the point in $L$ than the minimum separation between rectangles computed so far, i.e. $\delta$. Thus, for a northeast point in $L$ we only need to examine the distances between it and at most four southwest points in $R$ to determine if the pair from $L$ and $R$ are closer than the minimum distance between rectangles found so far. This step of the problem thus reduces to a variant of the closest pair problem for a set of planar points. We can use the merging step of the MCC algorithm presented in [16] to determine the closest pair of points consisting of a northeast point from $L$ and a southwest point from $R$. We very briefly outline the steps in the implementation. Sort all the northeast points in the left slab and all the southwest points in the right slab which are within a horizontal distance of $\delta$ from the boundary together. Record for each point, its *global_rank*, which is its position in the sorted list. Then sort the northeast and southwest points separately, and record for each point its *local_rank*, which is its position in its own sorted list. The index of a processor which contains one of the desired points on the other side is then obtained as described in Section II. The other three points are in processors immediately adjacent to this.

Find the minimum over all the diagonal distances computed above and set it equal to $\delta_d$. Finally set $\delta_S = Min[\delta_S,\delta_d]$.

Each of the three merging steps can be implemented in $O(\sqrt{k})$ time where $k$ is the number of segments being merged. The overall time complexity of the algorithm is therefore $O(\sqrt{n})$.

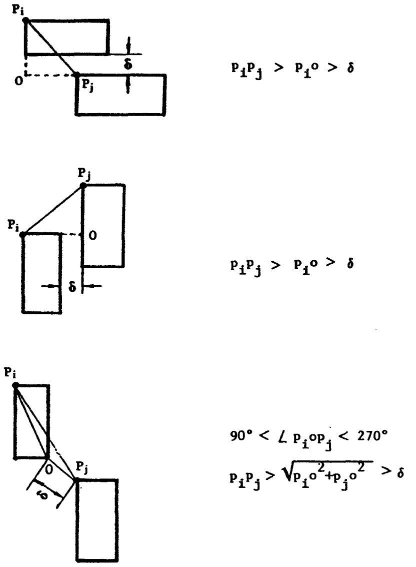


Figure 6

## VI. Summary

We have presented MCC algorithms for several rectangle intersection problems. Given a set of $n$ iso-oriented rectangles, we presented algorithms to determine the area of the logic "OR" and the logic "AND" of these rectangles. We then presented an algorithm for determining the maximum overlap for the set of rectangles, which also provides a solution for the fixed-size rectangle placement problem. Finally, we described an algorithm for computing the minimum separation of a set of non-overlapping iso-oriented rectangles. All the algorithms require an optimal $O(\sqrt{n})$ time on a $\sqrt{n}\times\sqrt{n}$ MCC with constant storage per processor.

In conclusion, we note that there are several other rectangle intersection problems similar to these considered in this paper (e.g. determining for each rectangle the number of rectangles that intersect it), that can be efficiently solved on an MCC using the "strip refinement" strategy. One interesting problem needing research is to determine an efficient means of *reporting* all pairs of intersecting rectangles using a Mesh-Connected Computer.

## Reference

[1] Gregory, J. and McReynolds, R., "The SOLOMON computer," IEEE Trans. Comput., c-13, 1963, pp. 774-781.

[2] Kruse, B., "A parallel processing machine," IEEE Trans. Comput., c-23, 1973, pp. 1057-1087.

[3] Hwang, K. and Fu, K. S., "Integrated computer architectures for image processing and database management," IEEE Computer, 15, 1983, pp. 51-60.

[4] Thompson, C. D. and Kung, H. T., "Sorting on a mesh-connected parallel computer," Communications of the ACM, 20, no. 4, Apr. 1977, pp. 263-271.

[5] Nassimi, D. and Sahni, S., "Bitonic sort on a mesh-connected computer," IEEE Trans. Comput., C-27, no. 2, 1979, pp. 2-7.

[6] Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," IEEE Trans. Comput., c-30, Feb. 1981, pp. 101-106.

[7] Kung, H. T. and Leiserson, C. E., "Systolic arrays (for VLSI)," Sparse Matrix Proceedings 1978, ed. G. W. Stewart, 1979, pp. 256-282.

[8] Atallah, M. J. and Hambrusch, S. E., "Solving tree problems on a mesh-connected processor array," manuscript, Purdue Univ .

[9] Atallah, M. J. and Kosaraju, S. R., "Graph problems on a mesh-connected processor array," J. of ACM, 31, no. 3, July 1984, pp. 649-667.

[10] Stout, Q. F., "Tree-based graph algorithms for some parallel computers," Proc. of the 1985 Int. Conference on Parallel Processing, 1985, pp. 727-730.

[11] Gopalakrishnan, P. S., Ramakrishnan, I. V., and Kanal, L. N., "An efficient connected components algorithm on a mesh-connected computer," Proc. of 1985 Int. Conf. on Parallel Processing, 1985, pp. 711-714.

[12] Gopalakrishnan, P. S., Ramakrishnan, I. V., and Kanal, L. V., "Computing tree functions on mesh-connected computers," Proc. of the 1985 Int. Conference on Parallel Processing, 1985, pp. 703-710.

[13] Guibas, L. J., Kung, H.T., and Thompson, C. D., "Direct VLSI implementation of combinatorial algorithms," Proc. Caltech Conference on VLSI, 1979, pp. 509-525.

[14] Nassimi, D. and Sahni, S., "Finding connected components and connected ones on a mesh-connected Parallel Computer," SIAM Journal on Computing, 9, no. 4, 1980, pp. 744-757.

[15] Miller, R. and Stout, Q. F., "Computational geometry on a mesh-connected computer," Proc. of 1984 Int. Conf. on Parallel Processing, 1984, pp. 66-73.

[16] Lu, M. and Varman, P., "Solving geometric proximity problems on mesh-connected computers," Proc. of 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Nov. 1985, Miami Beach, Florida, pp. 248-255.

[17] Mead, C. and Conway, L., in Introduction to VLSI-systems, ed. Addison-Wesley, MA: Reading, 1980.

[18] Eastman, C. M. and Lividini, J., "Spatial search," Report 55, Institute of Physical Planning, 1975.

[19] Guting, R. H., "Optimal divide-and-conquer to compute measure and contour for a set of iso-rectangles," Acta Informatica, 21, 1984, pp. 271-291.

[20] Bentley, J. L. and Shamos, M. I., "Divide-and-conquer in multidimensional space," Proc. of 8th Ann. ACM Symp. on Theory of Computing, May 1976.

# M$^2$-Mesh: An augmented mesh architecture

Tsair-chin Lin and D. I. Moldovan

Dept. of Electrical Engineering/Systems
University of Southern California
Los Angeles, CA 90089-0781

**Abstract** An architecture called m$^2$-mesh is proposed here. This architecture augments mesh structure with shared memories and multiplexed multiple buses. The approach to the design of this architecture is presented. The routing capability of this architecture is discussed and its performance is analyzed. Routing algorithms for performing permutation, two point communication and broadcasting are proposed. It is shown that any permutation can be performed on m$^2$-mesh in $2N^{1/2}$ steps, as compared to $3(N^{1/2}-1)$ steps for mesh structure. Communication between any two processors, row broadcasting, column broadcasting can all be done in 2 steps on this architecture. We also compare the performance of m$^2$-mesh with mesh, mesh with single broadcasting, mesh with multiple broadcasting and tree structures. The applications of m$^2$-mesh to problems such as semi-group computations, algorithms with nested loops, linear algebra problems are discussed.

## 1. Introduction

A parallel computer achieves speedup over a uniprocessor system by executing several tasks on different processors simultaneously. The degree of speedup depends primarily on the parallelism of the program and the performance of the parallel computer. The communications between different problem tasks are performed by the interconnection network of the system. If the communication requirements of a problem "match" the system's interconnection structure, then the problem can be solved quickly. Otherwise, extra time has to be spent on data routing. Thus, often the performance of a parallel computer depends primarily on it's data routing capability. Several approaches can be taken to increase the performance of parallel computers. Most common are to design a system with a powerful interconnection network [19, 2], to design efficient parallel memories [9], and to design effective parallel algorithms for a particular architecture [21, 15]. Recently researchers proposed the use of small size systems to balance computation and communication [10, 12], or finding an efficient way to map algorithms to computers [1, 6, 11].

One of the most frequently mentioned parallel computers is the mesh-connected computer (MCC). An MCC has the advantages of simple topology and good scalability. Because of this, many results on the application of MCC have been reported, and among these are: sorting [22, 15], image processing [18], and graph algorithms [16]. However, previous researches showed that the communication time in MCC tends to dominate the total execution time. For example, the multiplication of two $N^{1/2}$ by $N^{1/2}$ matrices takes $O(N^{1/2})$ routing steps when executed on a $N^{1/2} \times N^{1/2}$ MCC [3]. Regarding the routing capability of MCC, it's shown that in general $3(N^{1/2}-1)$ steps are required to perform any permutation [17]. Therefore, for general application, mesh connected computers suffer from the limitation of their routing capability.

In this paper, we propose an architecture based on mesh structure, called multiplexed-multiple-bus mesh, or simply m$^2$-mesh. The system consists of both local and shared memory modules. We will show that by using this structure, many algorithms can be executed efficiently. The performance of this architecture is compared with mesh, mesh with broadcasting, tree structures. It can be seen that this structure performs more efficiently for a broader range of applications than MCC, with reasonable additional cost.

## 2. M$^2$-mesh Architecture

### 2.1. Comparision of two models of SIMD architectures

This research was supported by NSF grant ECS-8307258 and JSEP contract No. F49620-85-C0071

A mesh connected computer with N processors (N is a perfect square) is shown in figure 1.

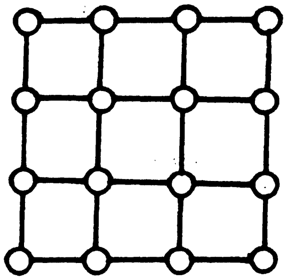


**Figure 1:** A mesh connected computer.

The processors are numbered in row major ordering. PE(i) is connected directly to PE(i±1), PE($i \pm N^{1/2}$), except the boundary PEs. The near neighbor interconnection is a simple but useful topology. Algorithms requiring frequent interactions between problem tasks can be mapped to this architecture naturally. The other feature of MCC is its good scalability, i.e., it's easy to expand from small size MCC to large size. However, for problems requiring global communication, mesh interconnection does not perform well. Considering the data routing time only, a permutation takes $3(N^{1/2}-1)$ steps on mesh. Broadcasting from one PE to any other PE takes $2(N^{1/2}-1)$ steps. In general, since the diameter of mesh is $2N^{1/2}$, normally $\Omega(N^{1/2})$ of routing steps are required for one routing function. So, if an algorithm takes O(f(N)) computation steps, and one routing function has to be performed between two computation steps, the worst case execution time could be $O(N^{1/2}f(N))$, instead of O(f(N)) (because $O(N^{1/2})$ routing steps are required for each of the $O(N^{1/2})$ data routing functions). Thus, the performance of an MCC could be greatly impaired by its communication capability. On way to overcome this problem without changing the system architecture is using mapping techniques. In [11] it was shown that any permutation can be performed on a MCC in at most four steps if the data are loaded properly. Thus, any algorithm which requires only a single type of permutation can always be achieved in constant data routing time. However, if the permutation requirement of the algorithm changes from time to time during execution, this mapping technique may not perform well.

The other way for overcoming the commnunication limitation of an MCC is to reconsider the original problem and to devise a better algorithm for the MCC. For example, the bitonic sorting algorithm on an MCC designed in [15] is achieved by this approach. This approach involves the design of parallel algorithms, which apparently is not an easy task for many users. Also, it suffers from lack of universality, because every algorithm has to be considered separately.

Our approach in this paper is to enhance the performance of MCC with reasonable additional cost. The idea of this newly proposed $m^2$-mesh architecture comes from the following observations. A mesh connected computer is really a multiprocessor system with local memory scheme. That is, every processor has its own private memory. Any two processors wishing to exchange information with each other have to use the mesh interconnection. This type of architecture can be modelled by the architecture shown in figure 2.

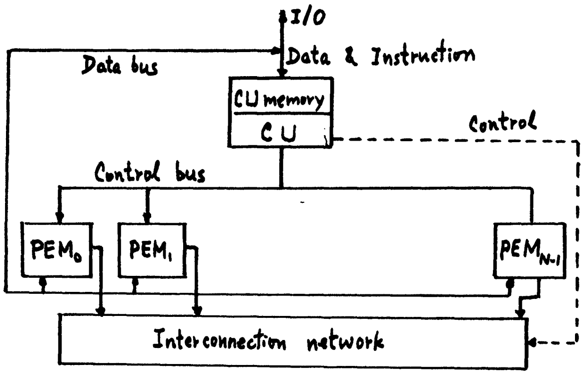


**Figure 2:** An SIMD machine with local memory scheme

However, since the longest path between any two processors is $2N^{1/2}$, long communication delay is inevitable. This problem is basically caused by the fact that local mamory scheme is used. Should two far-separated processors share the same memory module, then the communication time will not depend on their physical location. Since mesh structure has its merits, our strategy here is to include both local memory and shared memories. First consider the array processor model in figure 3 as used in [9]. In this architecture, both local memory and shared memory schemes are used here. Moreover, the interconnection between processors is separated from the inteconnection between memories.

**There are four interconnection networks in this architecture, each one may have different topology.**

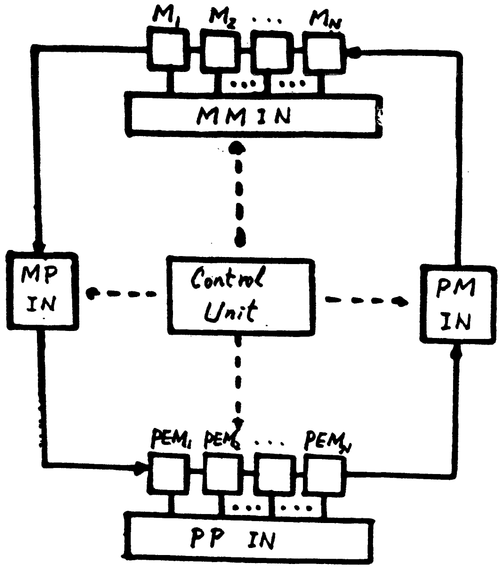


**Figure 3:** An SIMD with both local memory and shared memory system.

Every PEM contains a processor with its own local memory. The global memory modules M are shared by all processors. Thus interprocessor communication can be done either through processor-processor interconnection (PP), or through the shared memory. Compared to model shown in figure 2, this scheme is more powerful.

### 2.2. The $m^2$-mesh architecture

The idea in last section is adopted in our proposed architecture, which is shown in figure 4.

The system consists of N processors and N shared memory modules. Processors are interconnected by an $N^{1/2}$ by $N^{1/2}$ mesh, with local memory attached to each processor. Processors are identified by their two dimensional coordinates. Shared memory modules are numbered similarly. $M(i_1,j_1)$ and $M(i_2,j_2)$ share a row bus if $i_1 = i_2$, and similarly, $M(i_1,j_1)$, $M(i_2,j_2)$, $PE(i_3,j_3)$ share a column bus if $j_1=j_2=j_3$. In terms of the model from figure 3, our architecture uses mesh for processor-processor communication and processor-memory and memory-memory communications are done through buses.

The memories used here have special organization. Each memory module is divided into $N^{1/2}$ banks. The kth bank of M(i,j) is denoted as $M_k(i,j)$. Memory accesses work in the following fashion:

1. PE(i,j) can read every bank of every memory module in the jth column, i.e., PE(i,j) can read $M_k(l,j)$ $\forall k,l, 0 \leq k,l \leq N^{1/2}-1$.

2. PE(i,j) can only write to $M_i(j,j)$, the ith bank of M(j,j).

3. When a data item is written to $M_i(j,j)$ by PE(i,j), it's written through all the ith bank of the memories in the same row, i.e., $M_i(j,k)$ $\forall k, 0 \leq k \leq N^{1/2}-1$.

Since PE(i,j) can only write data to $M_i(j,j)$, only one of the processors in the same column can write at a particular time. From this point of view, the processors in the same column are time *multiplexed* on the column bus. However, processors in different columns can write to shared memory simultaneously. This means that the *multiple* column buses can be used concurrently. So, this architecture is called $m^2$-mesh. The write through scheme allows data to be transferred between processors efficiently, as will be explained in the next section.

## 3. Routing strategy and routing scheme

Since $m^2$-mesh contains mesh structure as a subset, the advantage of mesh is kept. Our routing stategy is that if communication is between near-neighbor processors, local communication through mesh is exploited. For long distance routing, buses and shared memory are used. This can be done by the implementation of two routing instructions: G(i,j) and L(i,j), where G(i,j) is global communication, and L(i,j) is local communication. In the following, we will show that many data routing functions which mesh cannot perform well can be achieved by $m^2$-mesh efficiently.

1. Two point communication: One way to evaluate the routing capability of an interconnection network is the communication time between any two processors. The communication from $PE(i_1,j_1)$ to $PE(i_2,j_2)$ requires $|i_1-i_2| + |j_1-j_2|$ steps on mesh, which is $2(N^{1/2}-1)$ in the worst case. By using $m^2$-mesh, this can always be achieved in two steps. First, $PE(i_1,j_1)$ writes to $M_{i_1}(j_1,j_1)$, then $PE(i_2,j_2)$ reads back from $M_{i_1}(j_1,j_2)$.

2.Column broadcasting: PE(i,j) sends data to PE(*,j)'s. This can be done by first writing data by PE(i,j) to $M_i(j,j)$, then by activating all PE's in the column, all the PE's in the same column read it back simultaneously. Two steps are sufficient.

3.Row broadcasting: PE(i,j) writes data to PE(i,*)s. This can be done in two steps: First, PE(i,j) writes to $M_i(j,j)$, then, every PE(i,k), $0 \leq k \leq N^{1/2}-1$, read data from $M_i(j,k)$.

4.Permutation: Any permutation can be done in $2N^{1/2}$ steps. Assume permutation P maps PE(i,j) to PE(p,q). We use the notation p = r(P(i,j)), q = c(P(i,j), where r and c indicate the row and the column number of a PE respectively.

Similarly, $i = r(P^{-1}(p,q))$, and $j = c(P^{-1}(p,q))$. The routing algorithm for permutation P is as following:

**Routing algorithm for permutations**

```
/* Permutation algorithm for P : PE(i,j) --> PE(p,q) *
    for n = 1 to $N^{1/2}$
      cobegin        /* row major loop */
        PE(n,i) --> $M_n(i,i)$    $0 \leq i \leq N^{1/2} - 1$
      coend
    for n = 1 to $N^{1/2}$
      cobegin        /* column major loop */
                     /* $j = r(P^{-1}(n,i))$, $k = c(P^{-1}(n,i))$*
        PE(n,i) <-- $M_j(k,i)$    $0 \leq i \leq N^{1/2} - 1$
      coend
```

Notice that if more than one column broadcastings occur concurrently, they can be done simultaneously. The situation for concurrent row broadcasting is different. A column bus contension will happen if the sources (or destinations) of some row broadcastings are in the same column. The situation is called row broadcasting conflict. For example, in figure 5 (a) row broadcasting conflict occurs, because to send data from PE(0,0) to PE(0,2) through global communication path, PE(0,0) has to write to M(0,0) which will conflict with PE(2,0)'s writing to M(0,0). In general, these broadcastings should be done sequentially. However, there are two ways to get around it. The first way is to skew the source (in case of source conflict) one step to the right (or left) through local link if possible, then the row broadcastings can be done concurrently (see figure 5 (b)). However, if the conflict involves too many sources, then this method is not efficient. The second way is to map the problem tasks in a transposed manner, then every row broadcasting changes to column broadcasting, and no conflicts will occur (see figure 5 (c)). This method is particularly suitable to algorithms in which only row broadcasting conflict exists. We will show the usefulness of this technique later.

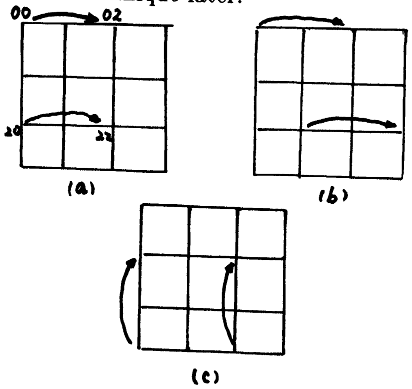


**Figure 5:** Row broadcasting conflict.

## 4. Performance analysis

The performance of $m^2$-mesh is studied by comparing it to other mesh related networks.

### 4.1. $m^2$-mesh vs. mesh

For problems requiring extensive information transfers, $m^2$-mesh is not necessary better than mesh. For example, $\Omega(N^{1/2})$ time is required to solve sorting problems on MCC, no matter with or without broadcasting [20, 8]. However, in general for ordinary algorithms $m^2$-mesh performs much better than mesh, as can be seen below.

1.Communication between any two processors: This requires $2(N^{1/2}-1)$ steps on mesh, but 2 steps on $m^2$-mesh.

2.Broadcasting: On mesh, this requires $O(N^{1/2})$, but on $m^2$-mesh, 2 steps are sufficient.

3.Permutation: $m^2$-mesh can perform permutation easier and faster. The lower bound for any permutation on mesh is $3(N^{1/2}-1)$ steps. On $m^2$-mesh, any permutation can be done in $2N^{1/2}$ steps. Moreover, the system overhead is different. For mesh, performing different permutations may require different routing algorithms. Besides, in each routing step, a masking function has to be computed, because some processors may need transfer data, some may not. Therefore the overhead is very large. In the case of $m^2$-mesh, our routing algorithm proposed in the last section is universal, i.e., independent of permutations. Also, system overhead is reduced significantly; because during routing, processors are activated row by row regularly.

### 4.2. $m^2$-mesh vs. mesh with single broadcasting

The mesh with single broadcasting interconnection is proposed in [20]. It's easy to see that $m^2$-mesh can simulate mesh with single broadcasting. So we have the following lemma.

**Lemma 1**: Any algorithm taking $\theta(n)$ time on mesh with single broadcasting can be executed on $m^2$-mesh in O(n) time.

For some algorithms, $m^2$-mesh performs better. This can be seen in the next section.

### 4.3. $m^2$-mesh vs. mesh with multiple broadcasting

The comparison between the two architectures is done in two aspects.

1.Permutation: Mesh with broadcasting structure doesn't perform better than mesh in terms of permutations. So, in general $3(N^{1/2}-1)$ steps are required to perform permutation on mesh with multiple broadcasting. Also, the control overhead is large and the routing algorithms are complex. For $m^2$-mesh, $2N^{1/2}$ steps are required. Also, the overhead is low.

2.Broadcasting: For row broadcasting, column broadcasting, or one to all broadcasting, both networks perform equally well. When row broadcasting conflict happens, $m^2$-mesh is worse. In the worst case, if row broadcasting conflict involves $N^{1/2}$ rows, then $m^2$-mesh could be $N^{1/2}$ steps slower. However, by using the technique mentioned in section 3, this can be prevented in some cases.

### 4.4. $m^2$-mesh vs. tree

We can use row broadcasting or column broadcasting to simulate tree operations. Therefore, using the same argument as in [8], we have the following lemma:

**Lemma 2**: If n data items are distributed in n different rows with one in each row, then any non-trivial semi-group computation of n data items can be performed in O(logn) steps.

## 5. Applications

The $m^2$-mesh is an architecture with a varity of data routing capabilities. So its potential is promising. For algorithms requiring lots of information flow (such as sorting), $m^2$-mesh can perform at least as well as mesh. So any parallel algorithms designed for mesh can be adopted by $m^2$-mesh. However, if global communications are required, $m^2$-mesh is much better architecture. In the following, we will discuss some application examples. Other applications can be probed by using the technique described here.

### 5.1. Semi-group computations

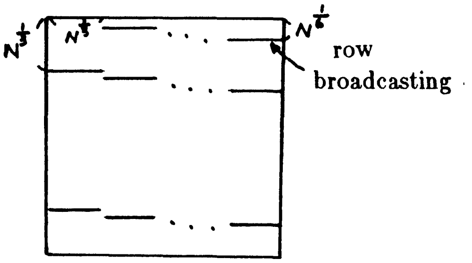
This type of computations include finding maximum, minimum, sum of N data items, etc.. We have the following theorem for these problems.

**Theorem 1**: A semi-group operation of N data items can be performed in $O(N^{1/6})$ time in a $m^2$-mesh.

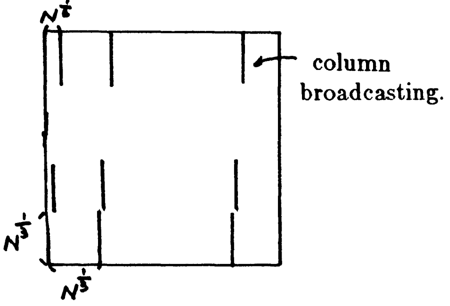
*Proof*: An algorithm for semi-group operations taking $O(N^{1/6})$ in 2-MCC with multiple broadcasting was porposed in [8]. Our proof exploits the same algorithm, except that we have to prove that every step of that algorithm can be simulated here. We will not list the original algorithm here. Based on that algorithm, the following two arguments are sufficient to prove this theorem.

(1) The operation of finding the Max of a block of data within $N^{1/6}$ x $N^{1/6}$ PE's using local communication can be executed on $m^2$-mesh in exactly the same way as in a mesh with multiple broadcasting architecture.

(2) The algorithm exploits concurrent row broadcasting operations, but no concurrent column broadcasting. Therefore we can transpose the original problem tasks so that no row broadcasting conflict will occur. For example, the following communications are required in that algorithm:



By transposing the assignment of problem tasks, the communications are as follows:



This way the row broadcasting conflict can be prevented. So from that algorithm, $O(N^{1/6})$ time is sufficient.

### 5.2. Algorithms with nested loops

Algorithms with nested loops cover a broad range of algorithms in signal processing, linear algebra, graph theory, and others. Examples of this type of algorithms include Finite Impulse Response (FIR) filtering algorithm, matrix multiplication, LU decomposition, transitive closure algorithm, etc.. The mapping of this type of algorithms to systolic array and array processor has drawn a lot of attention recently [7, 14, 13, 5]. One of the most effective way to map this type of algorithms is via algorithm transformation [14, 13]. In [10], the algorithm transformation technique is applied to mapping algorithms to array processors with mesh interconnection. One important conclusion there is that if a given array processor has broadcasting capability, then sometimes algorithms can be mapped to architecture with shorter execution time. For example, consider an algorithm with dependence matrix D. The way to map algorithm to mesh architecture is by finding a transformation T such that the transformed algorithm has a shorter execution time. The transformation T transforms D to $D'$ (=TD). One of the constraints in solving T is that every element in the first row of $D'$ should be greater than zero. However, if the system has broadcasting capability, this constraint can be released. (For details of reason for this argument, please see [10].) Besides, the criteria for considering tradeoff among computation time, communication time and local memory size of each processor will be different if broadcasting capability is provided by the architecture. Therefore, $m^2$-mesh can be used to solved algorithms with nested loops more efficiently than by using mesh architecture.

*Example 1*: Convolution computation. Given two sequences h(k) and x(k), k = 1, 2, ...., n, the convolution of h and x is defined as

$$y(j) = \sum_{k=1}^{n} h(k)x(j\text{-}k) \quad j = 1, ..., n.$$

This algorithm can be implemented by the following Fortran-like program:

```
  DO 10 j = 1,n
  DO 10 k = 1,n
    y(j) = h(k) * x(j-k) + y(j)
10 CONTINUE
```

The dependence matrix for this algorithm is

$$D = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Without using broadcasting, we found that the transformation

$$T_1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

is optimal, and the transformed matrix $D_1'$ is

$$D_1' = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 1 \end{bmatrix}$$

The number of computation steps $M_1$ and the number of data routing steps $R_1$ using this transformation are

$$M_1 = 2n$$

$$R_1 = 2n\text{-}1$$

respectively.

Now, consider another transformation

$$T_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Using this transformation, the transformed dependence matrix is

$$D_2' = \begin{array}{c} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \\ \begin{array}{ccc} y & h & x \end{array} \end{array}$$

The number of computation steps and data routing steps are

$$M_2 = n$$

$$R_2 = n\text{-}1$$

respectively.

Therefore, this transformation results in a faster algorithm. However, from $D_2'$, it can be seen that the value of h vector must be broadcasted during computation. Therefore, this is not a valid transformation for mesh without broadcasting capability, but is a good one for the $m^2$-mesh.

### 5.3. Parallel linear algebra algorithms

From the analysis in previous sections, there are mainly two things that $m^2$-mesh performs better than mesh architecture:

1. The communication between any two processors on $m^2$-mesh requires only two steps. However, it could be as worse as $2(N^{1/2}\text{-}1)$ steps on mesh.

2. $m^2$-mesh has broadcasting capability.

In the following, we will show that these two characteristics match directly with the requirements of many parallel linear algebra algorithms.

*Example 2*: Gauss-Jordan algorithm for computing a solution X of AX = B. Assume A is N by N, and B is N by M, this algorithm can be described as follows:

```
for j = 1 step 1 until N do
    row i <-- row i - $(a_{ij}/a_{jj})$row j, $(1 \le i \le N, i \ne j)$;
  $x_{ij}$ <-- $a_{ij}/a_{ii}$, $(1 \le i \le N, N+1 \le j \le N+M)$.
```

In the jth step of the algorithm, row j is broadcasted to all other rows. This can be done in $M^2$-mesh in two routing steps only. Then all rows can perform computations simultaneously. Therefore this algorithm can be solved on $m^2$-mesh efficiently.

*Example 3*: C-K algorithm for triangular system problem [4]. This algorithm, proposed by Chen and Kuck, can be described as follows:

```
row i = (row i)/$a_{ii}$   $1 \le i \le N$
for j = 1 step 1 until N-1 do

    row i = row i - $\sum_{k=j}^{2j-1} a_{i,i-k} rowi-k$
                    $j+1 \le i \le N$

$x_i = a_{i,N+1}$        $1 \le i \le N$
```

Assume the elements in one row of the matrix are mapped to one row of processors. When $j = j_1$, the processors at the ith row need information from processors at row $i\text{-}j_1$, row $i\text{-}j_1\text{-}1$, up to row $i\text{-}2j_1\text{-}1$. Hence, the communication cost is $2j_1\text{-}1$. However, on $m^2$-mesh, this can be achieved in 2 steps by using broadcasting. Overall, the communication cost of this algorithm on mesh is

$$J_1 = 1+2+ \ldots.+ N\text{-}1 = O(N^2)$$

While on $m^2$-mesh, only O(N) steps are required.

## 6. Conclusion

In this paper we proposed a multiplexed-multiple-bus mesh parallel computer system. The system not only retains most of the useful characteristics of mesh connected computers, but also augment them with efficient broadcasting and permutation capabilities. This architecture doesn't increase too much the hardware of each processor. The only significant cost is that N shared memory modules are required. However, with today's VLSI technology memories are rather inexpensive. We have shown that $m^2$-mesh has a much more powerful routing capability than mesh. We also showed that a wide range of applications are possible. More applications can be explored by using the architecture characteristics of $m^2$-mesh and the idea of mapping technique discussed in this paper.

## References

[1] S. H. Bokhari.
On the mapping problem.
In *Proc. of 1979 International Conference on Parallel Processing.* 1979.

[2] T. Y. Feng.
A survey of interconnection networks.
*Computer* 14(12):12-27, December, 1981.

[3] W. M. Gentleman.
Some complexity results for matrix computations on parallel processors.
*J. ACM.* 25(1):112-115, Jan., 1978.

[4] D. Heller.
A survey of parallel algorithms in numerical linear algebra.
*SIAM Review* 20(4):740-777, Oct., 1978.

[5] A. Khurshid and D. Fisher.
Algorithm implementation on reconfigurable mixed systolic arrays.
In *Proc. of 1986 Int'l conf. on parallel processing,* pages 748-755. Aug., 1985.

[6] R. Kuhn.
Efficient mapping of algorithms to single stage interconnections.
In *7th International Symposium on Computer Architecture.* 1979.

[7] H. T. Kung.
Let's design algorithms for VLSI systems.
In *Proc. Conf. on VLSI: Architecture, Design, Fabrication*, pages 65-90. Jan. , 1979.

[8] V. K. P. Kumar and C. S. Raghavendra.
Array processor with multiple broadccasting.
In *Annual Symposium on Computer Architecture.* 1985.

[9] D. H. Lawrie.
Access and alignment of data in an array processor.
*IEEE Transactions on Computers* 24(12):1145-1155, Dec., 1975.

[10] T. C. Lin and D. Moldovan.
Tradeoffs in mapping algorithms to array processors.
In *Proc. of 1985 international conf. on parallel processing,* pages 719-726. 1985.

[11] T. C. Lin, D. I. Moldovan.
*Mapping of algorithm permutations into mesh-connected SIMD computers.*
Technical Report CRI-86-1, USC, Jan., 1986.

[12] R. Miller and Q. F. Stout.
Varying diameter and problem size in mesh-connected computers.
In *Proc. of 1985 International conf. on parallel processing,* pages 697-699. 1985.

[13] D. I. Moldovan and J. A. B. Fortes.
Partitioning and mapping algorithms into fixed size systolic arrays.
*IEEE Trans. Comp.* C-35(1):1-12, Jan., 1986.

[14] D. Moldovan.
On the design of algorithms for VLSI systolic arrays.
*Prodeedings of the IEEE* 71(1):113-120, Jan., 1983.

[15] D. Nassimi and S. Sahni.
Bitonic sort on a mesh-connected parallel computer.
*IEEE Trans. Computer* C-28(1):2-7, Jan., 1979.

[16] D. Nassimi and S. Sahni.
Finding connected components and connected ones on a mesh connected parallel computer.
*SIAM J. Computer* 9(9):744-757, 1980.

[17] C. S. Raghavendra, P. Kumar.
Permutation on Illiac IV type network.
In *Proc. of 1984 International Conference on Parallel Processing.* 1984.

[18] A. Rosenfeld.
Parallel image processing using cellar arrays.
In *IEEE Computer*, pages 14-20. 1983.

[19] H. J. Siegel.
A model of SIMD machines and a comparison of various interconnection networks.
*IEEE Transactions on Computers* C-28(12):907-917, Dec., 1979.

[20] Q. F. Stout.
Mesh connected computers with broadcasting.
*IEEE Trans. on computers* C-32(9):826-830, Sept., 1983.

[21] H. S. Stone.
Parallel processing with the perfect shuffle.
*IEEE Trans. on Computers* c-20(2):153-161, Feb., 1971.

[22] C. D. Thompson and H. T. Kung.
Sorting on a mesh-connected parallel computer.
*IEEE Tran. on Comp.* 20:263-271, Apr., 1977.

# ON THE UNIVERSALITY OF MULTISTAGE INTERCONNECTION NETWORKS *

*T.H. Szymanski*

Department of Electrical Engineering
University of Toronto

Abstract – We present a simple analytic model for the blocking probability of circuit switched multistage interconnection networks (MINs) when the set of requests submitted during each cycle is a randomly selected permutation. This model provides a quantitative measure of a network's permutation capability. We present an analytic model for the number of conflict free permutations realizable in a multipath network, and we present an analytic model for the expected number of cycles required to realize an arbitrary permutation in a network (i.e., the universality of a network). These models and techniques solve many interesting theoretical problems in the realm of interconnection network modelling.

However, these models also have practical aspects; they give a quantitative measure of the connectivity of a network, and they give an approximate figure of merit of a network's ability to supply data vectors to array processors in SIMD array processor architectures.

## 1. Introduction

The *permutation capability* of an multistage interconnection network (MIN) can be loosely defined as the blocking probability of the network when the requests submitted during each cycle form a randomly selected permutation. The *rearrangeability* of a MIN can be defined as the ability to realize an arbitrary permutation in one cycle [4]. The *universality* of a MIN can be loosely defined as its ability to realize an arbitrary permutation in multiple network passes (where each pass requires one cycle) [21].

Non-blocking (rearrageable) switching networks have been the subject of extensive theoretical research. Of the known rearrageable networks, large crossbars are far too expensive, and the various MINs that meet this criteria ([4][14][29]) require precomputed global routing for each individual permutation. The necessity for precomputed global routing makes these networks impractical, especially if the permutations are not known at compile time.

Many suboptimal (nonrearrangeable) MINs with simple distributed routing algorithms, such as banyan networks [8] and the Augmented Data Manipulator (ADM) [1], have been analyzed for aspects of their permutation capabilities or universalities. However, to date "there does not exist any satisfactory technique which compares and quantifies permutability of ADM, Banyan and other MINs" [2].

An analytic model that yields the blocking probability of a banyan network when the request pattern submitted during each cycle is a randomly selected permutation (and when the usual decentralized routing algorithm for banyan networks is used) has been presented in [27]. We use the techniques presented in [27] to develop an alternate analytic model, which will be used throughout this paper. The development of this alternate analytic model illustrates the generality of the technique described in [27].

Using this analytic model, we derive an accurate approximation for the number of conflict-free permutations realizable by a multipath MIN (i.e., the number of permutations realizable in one pass). Previously, no general techniques have been proposed for estimating the number of conflict free permutations in multipath MINs. Lower and upper bounds on the number of conflict free permutations in the AMD have been presented in [1], and a complex analysis for the exact number of conflict free permutations in the ADM has been presented in [15][16]. Our technique is simple and general, and can be used to obtain models for arbitrary networks such as the ADM and the many variations of banyan networks, using various decentralized routing algorithms.

We present an accurate analytic model for the expected number of network passes required to realize an arbitrary permutation, i.e., the universality of a MIN, when simple distributed routing algorithms are used. Upper bounds for the expected number of network cycles required to realize an arbitrary permutation in certain banyan networks (when precomputed global routing is used) have been presented [2][28]. It has been shown that by using precomputed global routing, the omega network [12] can realize an arbitrary permutation in 3 passes [21][25]. However, the fastest known algorithm for the routing requires $O(\log^4{}_2 N)$ time on an array processor [25]. A simpler routing algorithm that results in a universality of $6\log_2 N-1$ passes for a variation of the omega network has been presented in [25]. Our analysis is fundamentally different from these in that we assume a simple distributed routing algorithm is used; in the first pass, all requests in the permutation are submitted to the network. Requests that are not established in one pass must be resubmitted in the next pass, etc. The modelling technique we use is very general, and can be used to create models for the universality of other MINs using various decentralized routing algorithms.

While these models and techniques solve many interesting theoretical problems, they also have practical aspects. The number of simultaneous, conflict free connections possible in a network (when the requests form a permutation) is a quantitative measure of the *connectivity* of a network. "The connectivity of a MIN is critical with respect to the overall performance of a large parallel system" [2].

The universality of a network has often been used as an approximate figure of merit for the suitability of the network in *Single Instruction Multiple Data* (SIMD) array processor architectures. These machines operate in synchronization on numeric applications that can be "vectorized". When the processor array accesses a vector in unison, the resulting request distribution is (usually) a permutation. "One of the most significant factors in determining the performance of an array processor is the system's capability for providing data vectors at a rate matched to the processor rate" [12]. The system's capability for providing data vectors is closely related to its universality (or indirectly to its permutation capability), and hence "the permutation capability of a network is extremely important for the efficient operation of a supersystem" [2].

Section 2 includes the necessary definitions. Section 3 presents a simple analytic model for the blocking probability of MINs when the set of requests submitted during each cycle are a randomly selected permutation. Section 4 presents an analytic approximation for the expected number of conflict-free permutations realizable by a multipath MIN (in one cycle). Section 5 presents an analytic approximation for the expected number of network cycles required to realize an arbitrary permutation, i.e., the universality of a MIN. Section 6 summarizes the paper and

* This research was partially supported by NSERC (Canada) Grant A-5192.

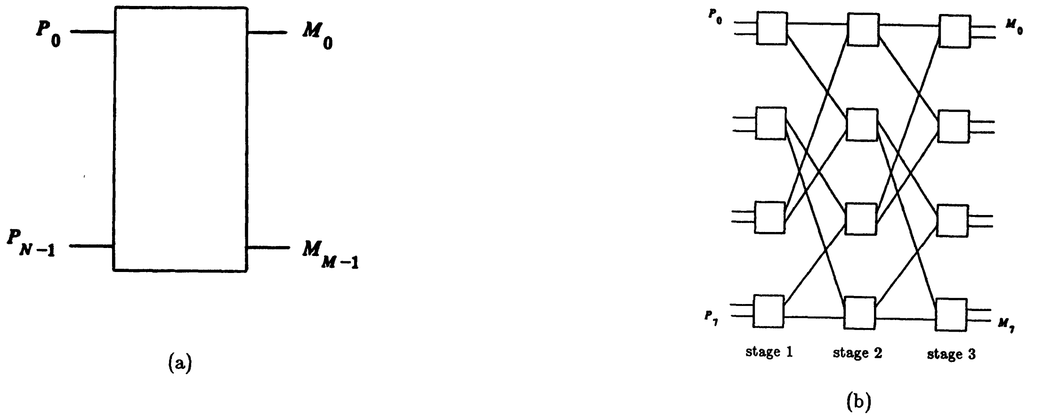


**Fig. 1:** a) an interconnection network connecting $N$ processors and $M$ memories. b) a $2^3 \times 2^3$ banyan network connecting 8 processors and 8 memories.

contains some concluding remarks.

## 2. Definitions

In a multiprocessor system, an ***interconnection network*** is used to connect $N$ processors to $M$ memories, as shown in fig. 1a. The crossbar network is non-blocking for all permutations, and hence is a candidate for array processor systems, but it is far too expensive for large $N$. Unique path multistage interconnection networks (***unipath MINs*** or ***banyan*** networks [8]) provide a reasonable alternative to large crossbars. A square $k^n \times k^n$ banyan network of size $N$ consists of $\log_k N$ stages, where each stage consists of $N/k$ crossbar switches of size $k \times k$, as shown in fig. 1b. ***Banyan*** networks have a number of desirable features; their cost grows only moderately as the network size increases and they use very simple, distributed routing algorithms [8].

Two fundamental criteria in the selection of an interconnection network are fault tolerance and performance. Banyan networks can be extended in many ways to significantly improve their fault tolerance, and such improvements may result in performance improvements. Since any practical multiprocessor system will almost certainly require a fault tolerant interconnection network, it is important to have analytic models that are general, and that can be used on these extended networks, as well as on the unipath networks.

### 2.1. Fault Tolerance

Banyan networks can be extended in many ways to significantly improve their fault tolerance. Assuming link failures are the predominant failure mechanism [19], each link in a network can be replaced by 2 links, as shown in fig. 2a. When each link is replaced by $d$ links, we call the resulting network a $d$-dilated network [10]. If total switch failures are a predominant failure mechanism, then the $d$ links in a dilated network that replace one link in the unipath network can actually be distributed among a number of the switches in the next stage [23]. In a 2-augmented network, one link leads to the same switch as it would in a unipath network, and the other link leads to a functionally equivalent switch. These networks are called ***augmented*** networks [23]. A 2-augmented delta network is shown in fig. 2b. The solid lines represent links leading to the same successor as they normally would, and the dotted lines represent links leading to equivalent successors. (The regular distributed routing algorithm is slightly modified; see [23]). Throughout this paper, we do not distinguish between dilated and augmented networks (their performances are similar [23,26]); denote either of these multipath networks as $p$-path MINs, where $p$ is the path multiplicity.

Banyan networks can also be replicated to improve their fault tolerance. We consider only dilated (or augmented) networks in this paper; see [26] for an analysis of replicated networks.

### 2.2. Performance Under Random Distributions

We now define the operating environment of MINs considered in this paper. We assume that the network is synchronously circuit switched; a system clock defines a ***cycle*** in the network; during the beginning of any cycle, a processor issues a request with probability $u$ (0). Assume (for now) that requests are randomly and evenly distributed over the memories, and are independent from cycle to cycle. For each request, the network tries to establish a circuit switched connection from the processor to the desired memory. The routing algorithm is very simple; in every stage, a switch that receives a request examines a particular bit in the destination, selects an appropriate output link, and forwards the request out on that link [8]. If the network has multiple paths (i.e., $p > 1$), then assume the switch randomly selects a free link from the $p$ appropriate links, and forwards the request on that link. One reason for the popularity of these networks is their simple distributed routing algorithms.

A connection may not be established because $p+1$ or more requests compete for $p$ output links at a particular switch; we say that ***link contention*** has occurred; $p$ requests are selected randomly and forwarded, and the others are ***blocked***, and must be resubmitted during the next cycle. Hence, during one cycle only a fraction of the submitted requests will have their connections established. Let $pb$ be the ***blocking probability*** of a network; if $N$ requests are submitted in one cycle, then $N \cdot pb$ requests are expected to be blocked during that cycle. Under these operating assumptions, (i.e., circuit switched MINs with random request distributions) numerous models exist for the blocking probability of a network [10] [22].

### 2.3. The Universality of an Interconnection Network

A very important practical operating assumption is to suppose $N$ requests are submitted during each cycle, where these $N$ requests form a permutation of the memory module indices (i.e., the requests are ***uniquely*** distributed). This situation occurs in ***Single Instruction Multiple Data*** (SIMD) array processors executing numeric algorithms. ***Data skewing*** algorithms are used to map the data (i.e., vectors or matrices) into the memories in such a way so that when the set of $N$ processors issues a request for a vector, the resulting request pattern is a permutation (of the memory module indices). One approximate figure of merit for a network's ability

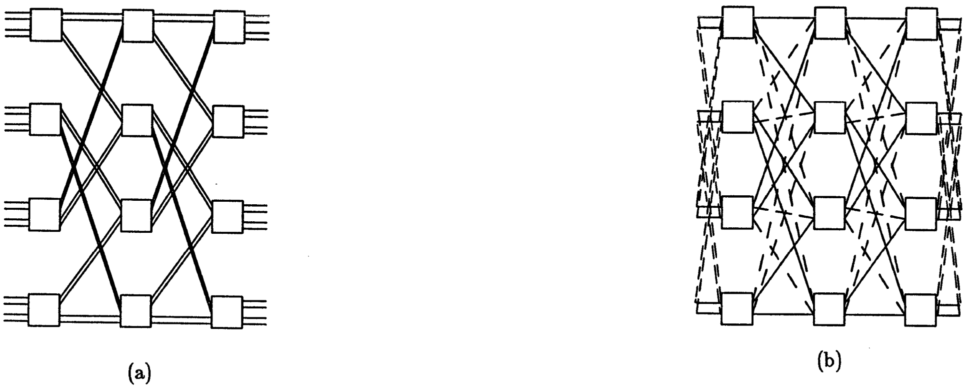


**Fig. 2:** a) a 2-dilated $2^3 \times 2^3$ delta network. b) a 2-augmented $2^3 \times 2^3$ delta network. (Delta networks [22] are a subclass of banyan networks [8]).

to supply data vectors is its universality. However, most permutations generated by an array processor are highly structured (i.e., they are not randomly selected permutations) [12] and a better figure of merit would be a model that yields the universality of a MIN when certain permutation classes are used. Lawrie has shown that the omega network can realize many permutation classes frequently arising in array processors [12]. Other architectures where the permutation patterns are not necessarily highly structured apparently include MIMD parallel reduction machines and logic machines [9].

In this paper, we present a simple analytic approximation for the blocking probability of a network when the requests submitted during each cycle form a randomly selected permutation. We present an analytic approximation for the number of conflict free permutations realizable by a network (i.e., the number of permutations that can be realized in one cycle), and we present an analytic approximation for the expected number of cycles required to realize an arbitrary permutation, i.e., the universality of a MIN.

## 3. The Permutation Capability of Multistage Interconnection Networks

We present a conceptually simple analytic model for the performance of generalized $p$-path $k^n \times k^n$ banyan networks, under either random or unique memory request distributions, in this section. It is first necessary to derive an analytic model for the blocking probability of generalized $p$-path $k^n \times k^n$ MINs under the assumption of random, uniform request distributions. This model is then modified (in the next sub-section) to yield the blocking probability under the assumption of unique memory request distributions (or simply *under unique distributions*).

### 3.1. Analysis of Multistage Interconnection Networks Under Random Distributions

In this section, an analytic model for the blocking probability of generalized $p$-path $k^n \times k^n$ MINs under random distributions is presented.

The assumptions for the analysis are as follows: 1) during the beginning of each cycle every processor issues a request with probability $u(0)$; 2) requests are randomly and uniformly distributed among the memories; and 3) unsatisfied requests in any cycle are ignored, and a new set of requests is issued during the next cycle subject to 1) and 2).

We assume that any physical link between stages $i$ and $i+1$ carries one request with probability $u(i)$, the aggregate link utilization. Hence, the probability that a switch in a particular stage receives $i$ requests can be modelled as a simple binomial distribution. Since each $p$-path $k \times k$ switch has $pk$ physical inputs, the probability that $i$ requests arrive at the switch in stage $m+1$ is given by

$$\binom{pk}{i} u(m)^i (1-u(m))^{pk-i}$$

The probability that $j$ of these requests select a particular logical output port is given by

$$\binom{i}{j}\left(\frac{1}{k}\right)^j \left(1-\frac{1}{k}\right)^{i-j}$$

In general, due to path multiplicity the first $f$ stages will not block, where $f$ is given by

$$f = \left\lfloor \log_k p \right\rfloor$$

The aggregate link utilization $u(f)$ after stage $f$ is simply $u(0)/p$.

In general, after any stage $m+1$ we are interested in $p(i)$, the probability that $p$ logically equivalent links carry $i$ requests. $p(i)$, for $0 \leq i \leq p$, can be calculated as follows

For $j < p$:

$$p(j) = \sum_{i=j}^{pk} \binom{pk}{i} u(m)^i (1-u(m))^{pk-i} \binom{i}{j}\left(\frac{1}{k}\right)^j \left(1-\frac{1}{k}\right)^{i-j} \quad (1.1)$$

For $j = p$:

$$p(j) = \sum_{i=p}^{pk} \binom{pk}{i} u(m)^i (1-u(m))^{pk-i} \sum_{t=p}^{i} \binom{i}{t}\left(\frac{1}{k}\right)^t \left(1-\frac{1}{k}\right)^{i-t} \quad (1.2)$$

After each stage $m+1$, we need to calculate the aggregate link utilization $u(m+1)$, as follows

$$u(m+1) = \sum_{i=0}^{p} \frac{p(i) \cdot i}{p} \quad (1.3)$$

The blocking probability of a network of size $N \geq 2^{f+1}$ (assuming each memory services at most 1 request in each cycle) is then

$$pb = 1 - \frac{p(0)}{u(f)} \quad (1.4)$$

The comparison of this model with simulations of various augmented and dilated networks can be found in [26]. The model is exact for unipath networks, but is only an approximation for multipath networks. Simple refinements of this model can also be found in [26].

### 3.2. Analysis of Multistage Interconnection Networks Under Unique Distributions

We can now adapt this model to yield the blocking probability of the network under the assumption of unique memory request distributions, using the technique presented in [27]. The blocking probability under unique distributions is the probability that an issued request will block, when the requests form a randomly selected permutation.

The assumptions for the analysis are as follows: 1) during the beginning of each cycle every processor issues a request with probability $u(0)$; 2) requests are uniquely and uniformly distributed among the memories; and 3) unsatisfied requests in any cycle are ignored, and a new set of requests is issued during the next cycle subject to 1) and 2).

Consider the $2^3 \times 2^3$ banyan network shown in fig. 1b. An output link from the first stage can reach 4 different memories, and a switch in the first stage can reach 8 different memories. The first request arriving at a switch in the first stage selects an output link with probability 4/8. A second request arriving at the same switch selects the same output link with probability 3/7, and selects the other output link with probability 4/7.

The preceding technique can be generalized; consider the probabilities of requests selecting outputs at a particular $p$-path $k \times k$ crossbar switch in stage $m$ of the banyan network. Each output link leads to $k^{n-m}$ memory modules. The first request selects a particular output link with probability $k^{n-m}/k^{n-m+1}$. Given that the first request selects a particular output link, then the second request selects that same output link with probability $(k^{n-m}-1)/(k^{n-m+1}-1)$.

As a notational convenience, define function $p\_s(s,r,m)$ as the probability that a request at a switch in stage $m$ will select a particular output link given that $s$ requests have already selected that link, and that $r$ requests have already selected other links.

$$p\_s(s,r,m) = \begin{cases} \dfrac{k^{n-m}-s}{k^{n-m+1}-s-r}, \text{ for } s < k^{n-m},\ r+s < k^{n-m-1} \\ 0, \textit{ otherwise} \end{cases}$$

In a similar manner, define function $p\_ns(s,r,m)$ as the probability that a request at a switch in stage $m$ will not select a particular output link given that $s$ requests have already selected that link, and that $r$ requests have already selected other links.

$$p\_ns(s,r,m) = \begin{cases} \dfrac{(k-1)k^{n-m}-r}{k^{n-m+1}-s-r}, \text{ for } r < (k-1)k^{n-m},\ s+r < k^{n-m+1} \\ 0, \textit{ otherwise} \end{cases}$$

Assuming that the events occurring at the inputs of each switch in stage $m+1$ of the banyan network are independent, (which will result in a slightly pessimistic blocking probability), then the probability that $i$ requests arrive on $k$ input links of a switch in stage $m+1$ is given by

$$\binom{k}{i} u(m)^i (1-u(m))^{k-i}$$

Given that $i$ requests arrive at a switch in stage $m+1$, the probability that $j$ of these requests select a particular output and that $i-j$ of these requests do not select that same output is given by

$$\binom{i}{j} \prod_{s=0}^{j-1} p\_s(s,0,m) \prod_{r=0}^{i-j-1} p\_ns(j,r,m)$$

Hence, eq. (1) can easily be modified to yield the blocking probability of $p$-path $k^n \times k^n$ banyans under unique distributions. Due to path multiplicity, the first $f$ stages will not block, where $f$ is given by

$$f = \lfloor \log_k p \rfloor$$

Under unique distributions, the last $l$ stages will not block, where $l$ is given by

$$l = \lfloor \log_k p \rfloor + 1$$

We need a value for $u(f)$, the aggregate utilization on links leaving stage $f$; since the first $f$ stages did not block, let $u(f) = u(0)/p$. In general, after any stage $m+1$ we are interested in $p(i)$, the probability that $p$ logically equivalent links carry $i$ requests. $p(i)$, for $0 \le i \le p$, can be calculated as follows

For $j < p$:

$$p(j) = \sum_{i=j}^{pk} \binom{pk}{i} u(m)^i (1-u(m))^{pk-i} \cdot \qquad (2.1)$$

$$\binom{i}{j} \prod_{x=0}^{j-1} p\_s(x,0,m) \prod_{x=0}^{i-j-1} p\_ns(i,x,m)$$

For $j = p$:

$$p(j) = \sum_{i=p}^{pk} \binom{pk}{i} u(m)^i (1-u(m))^{pk-i} \cdot \qquad (2.2)$$

$$\sum_{t=p}^{i} \binom{i}{t} \prod_{x=0}^{j-1} p\_s(x,0,m) \prod_{x=0}^{i-t-1} p\_ns(i,x,m)$$

After each stage $m+1$, we need to calculate the aggregate link utilization $u(m+1)$, as follows

$$u(m) = \sum_{i=0}^{p} \frac{p(i) \cdot i}{p} \qquad (2.3)$$

The blocking probability of the network is then

$$pb = 1 - \frac{u(n-l)}{u(f)} \qquad (2.4)$$

This model is compared with simulations in the next subsection.

#### 3.2.1. Comparison of Model with Simulations

We now compare this simple analytic model with a number of simulations, for various unipath and multipath MINs.

Fig. 3 illustrates the blocking probabilities for various augmentations of $k^n \times k^n$ banyans of size $N$ under unique distributions. (The $pb$ of dilated and augmented networks, with the same path multiplicity, differ by only a few percent [26]; these simulations happen to be for augmented networks, although the model is more accurate for dilated networks). These figures indicate that the model is reasonably accurate.

There are two major sources of error in this model. First, the analytic model for the $pb$ under the assumption of random, independent requests is only an approximation itself, and has an error of a few percent. Secondly, when the model is adapted for unique distributions, the assumption that requests arriving at the inputs of a switch are statistically independent introduces some inaccuracies. However, more refined analyses can take these considerations into account (see [26]).

Hence, eq. (2) can be used as a quantitative measure of a $p$-path $k^n \times k^n$ banyan network's permutation capability. The techniques presented here are general, and hence models for other MINs are easily created.

### 4. On the Expected Number of Conflict-Free Permutations in Multipath MINs

A simple technique for determining the exact number of conflict free permutations in unipath $2^n \times 2^n$ MINs has been presented in [1]. The extension to unipath $k^n \times k^n$ MINs is simple and is presented here.

In a unipath MIN with $N$ processors and $N$ memories, there are exactly $N$ links between each stage. If a permutation (of $N$ requests) is conflict-free, every link must carry one request. Hence,

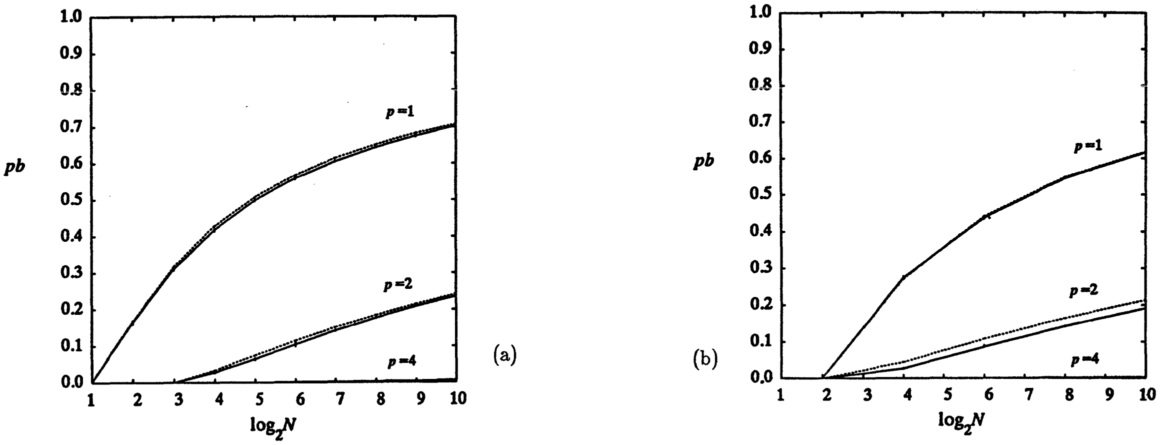


Fig. 3: (a) *pb* for various augmentations of $2^n \times 2^n$ networks with $N$ sources under unique distributions. (b) *pb* for various augmentations of $4^n \times 4^n$ networks with $N$ sources under unique distributions. (solid curves are simulations with 95% confidence intervals shown; dashed curves are analytic.)

each $k \times k$ switch will have $k$ requests arriving at its inputs, and must have $k$ requests leaving on its outputs (so that no requests block). Each switch can therefore perform $k!$ mappings of its inputs to its outputs, i.e., there are $k!$ distinct settings for each switch. Since there are $N/k$ switches per stage, and $\log_k N$ stages, then the exact number of conflict free permutations in a unipath network is simply

$$(k!) \exp \left(\frac{N}{k} \cdot \log_k N\right) \tag{3}$$

In a multipath network, the analysis is much more difficult. Since there are more than $N$ links between each stage, each link does not necessarily carry a request, and each switch does not necessarily have $k$ requests arriving at its inputs. The previous model can be adapted to estimate the number of distinct switch settings in a multipath MIN (see [20]), however this estimate "does not come close to estimating the number of permutations realizable by the [gamma] network. Estimating the actual number of permutations seems to be much harder, and is left as a challenging open problem" [20]. In this section, we present an analytic model for the expected number of conflict free permutations in multipath MINs.

Let $X_i$ be the event that $i$ requests block in one cycle, given that $N$ requests which form a permutation are submitted initially. In this section, we wish to find an analytic model for the probability distribution function *pdf* [6] of $X_i$, for $0 \leq i \leq N$. Let $P(X_i)$ represent the *pdf* of $X_i$.

The simplest approach for the *pdf* is to assume that all events are independent, and that $P(X_i)$ is simply binomially distributed;

$$P(X_i) = \binom{N}{i} pb^{\,i} (1-pb)^{N-i} \tag{4}$$

Fig. 4 illustrates the observed $P(X_i)$ versus the computed values for various networks. (Note that we used the *pb* obtained from simulations when computing these *pdf* 's.)

From fig. 4, we observe that the independence assumption is very accurate for the multipath banyans, and reasonably close for the unipath banyans. We can actually test the quality of the *pdf* by using the $\chi^2$ ***goodness of fit*** test [6]. At a 5% level of significance, the hypothesis is rejected for most unipath networks, and accepted for all multipath networks that we tried.

Given that the *pdf* is reasonable for multipath networks, we can accurately estimate the number of conflict-free permutations in these networks $N_p$ as follows;

$$N_p = N! \cdot (1-pb)^N \tag{5}$$

For unipath networks the *pdf* is less accurate, primarily because the independence assumption is not valid. A slightly more complicated multinomial distribution that explicitly models the effects of the statistical dependence is required for the *pdf* of unipath networks (see [26]).

In general, we are not interested in simulating a network to obtain a measurement for the *pb* used in eq. (5). Eq. (2.4) gives a reasonably accurate estimate of *pb* , which can be used in eq. (5).

### 5. The Expected Number of Network Cycles Required to Realize an Arbitrary Permutation

In this section, we develop an analytic model for the expected number of network cycles required to realize an arbitrary permutation, i.e., the universality of a network. The universality of various networks has been examined by numerous researchers. When global (i.e., precomputed) routing algorithms are used, the universalities of some networks have been established, and upper or lower bounds for others have been established. When distributed routing algorithms are used, the only results to date are simulations for the universality [15][31].

A proof for an upper bound for the number of cycles required to realize any arbitrary permutation in unipath $2^n \times 2^n$ banyans has been presented in [2]. The basic idea was to count the number of requests that must use a specific link in the network in the worst-case, i.e., if the worst case permutation would require $m$ requests to use a specific link, then the upper bound for the number of cycles is $m$ . The upper bound on the number of cycles in a $2^n \times 2^n$ banyan was shown to be $2^{\lfloor n/2 \rfloor}$. An identical bound (but using a slightly different technique) has been derived in [28].

However, the implied assumption in this bound is that the critical link will be utilized by a request in each cycle, and that the request that used it will indeed be established. To ensure this assumption would require precomputed routing for each individual permutation to optimally schedual requests over the critical link.

If the usual simple, distributed banyan routing algorithm is used, this upper bound does not apply (it will be exceeded occasionally). This situation occurs when a request uses the critical link during a cycle and is blocked at a later stage in the same cycle, hence requiring the use of the critical link again. Numerous examples of this situation can be found for networks of size $N \geq 8$. The correct upper bound (assuming a distributed routing algorithm) must also consider the conflicts that may occur at the

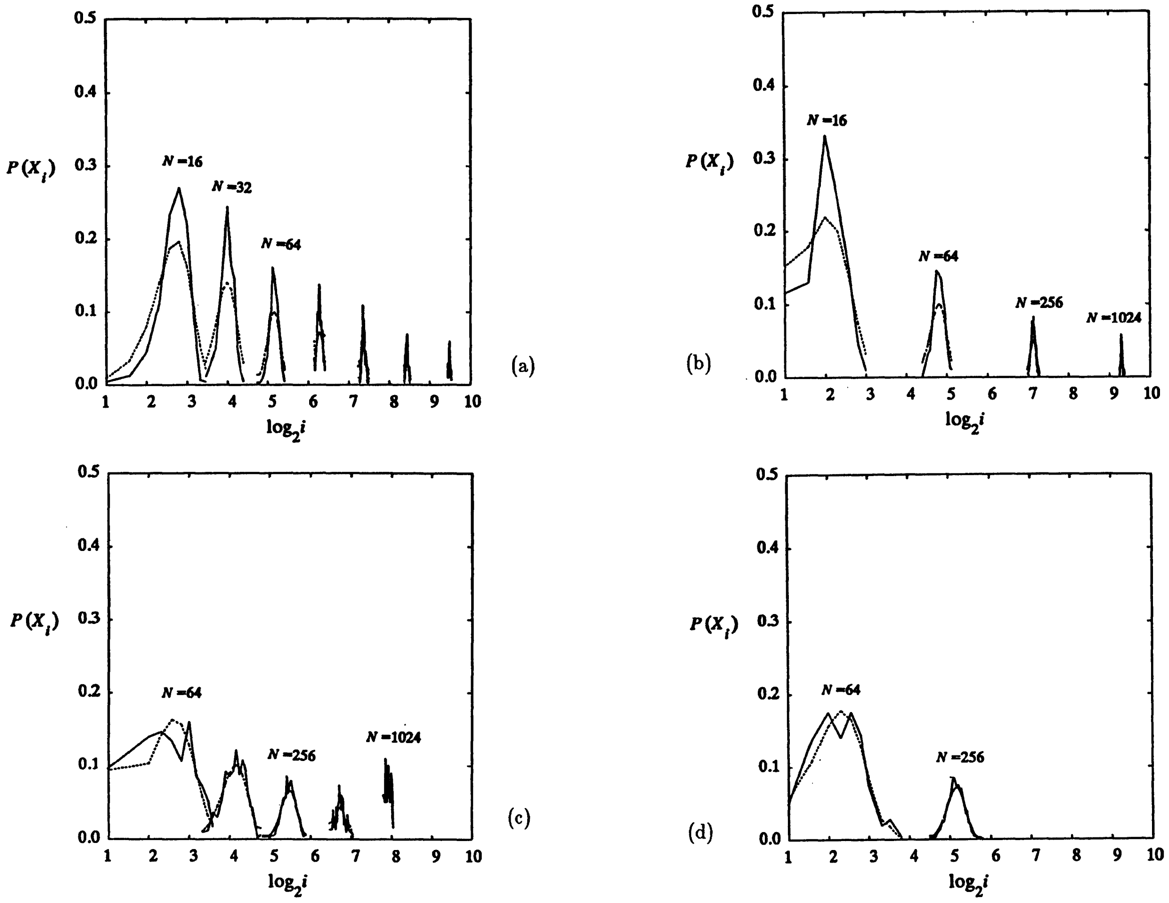


Fig. 4: a) $P(X_i)$ for unipath $2^n \times 2^n$ banyans. b) $P(X_i)$ for unipath $4^n \times 4^n$ banyans. c) $P(X_i)$ for 2-augmented $2^n \times 2^n$ banyans. d) $P(X_i)$ for 2-augmented $4^n \times 4^n$ banyans. (solid curves are simulations; dashed curves are analytic.)

switch after the critical link, and is simply $2 \cdot 2^{\lfloor n/2 \rfloor}$ [26].

In this section, we present an analytic model for the expected number of (synchronous, circuit switched) network cycles required to realize an arbitrary permutation. The accuracy of this model depends on the accuracy of the *pdf* for $X_i$. As we will show, the model is very accurate for multipath networks, and optimistic for unipath networks.

The analytic model is very simple; we use a time-varying *markov* model with $N+1$ states, denoted $S_i$, for $0 \leq i \leq N$, as shown in fig. 5. We also maintain a counter $c$, which represents how may network cycles have been executed; initially, $c = 0$. Each state $S_i$ represents the probability that $i$ requests are still unsatisfied after $c$ cycles and must be resubmitted in the next network cycle. This model has one source state, $S_N$, and one sink state, $S_0$. Initially, all $N$ requests in a permutation are unsatisfied, so $S_N = 1$, and all other states are 0. A variable $E_c$ represents the expected number of cycles required to realize any permutation, and $E_c = 0$ initially.

For each network cycle, we update every state, as follows. Consider updating state $S_i$; this state is the probability that $i$ requests remain unsatisfied, and this state can be reached from any state $S_j$, where $j > i$ and where $S_j > 0$. The physical events that correspond to this state transition are simple; the processor array submits $j$ requests in one cycle, $i$ of them are blocked, and the rest are accepted. The blocking probability of the network, when $j$ requests are submitted, can be computed from eq. (2.4). The probability that $i$ of these requests are blocked can be computed from eq. (4). Hence, the state transition rates for the entire markov model can easily be computed during each network cycle.

During each cycle $c$ ( $c = 1,2,3....$), some number of requests $j$ are satisfied (by reaching $S_0$); these contribute the value $c \cdot j / N$ to the expected number of cycles $E_c$. After a number of cycles (iterations), the model will converge to a stable state (in absence of numeric errors, the steady state will have $S_0 = 1$, and all other states equalling 0). The process can be terminated when $E_c$ remains relatively constant, and the resulting $E_c$ is the expected number of network cycles required to complete an arbitrary permutation.

Fig. 6 illustrates the analytic results and simulations for various networks.

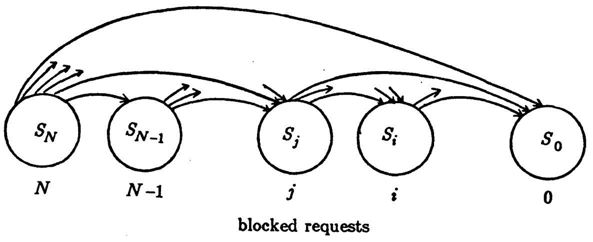


Fig. 5: markov model for an $N \times N$ MIN.

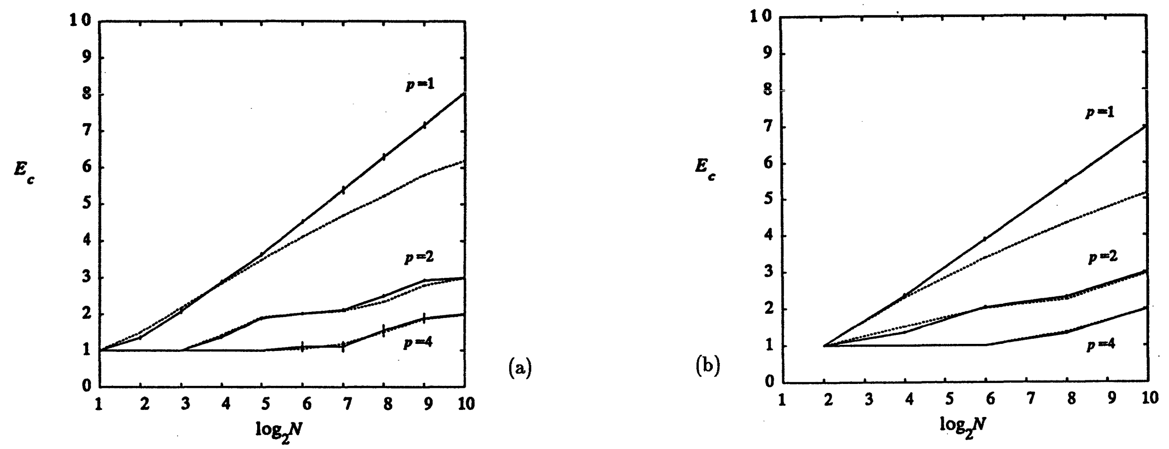


Fig. 6: a) $E_c$ for various $p$-path $2^n \times 2^n$ banyans. b) $E_c$ for various $p$-path $4^n \times 4^n$ banyans. (solid curves are simulations with 95% confidence intervals shown; dashed curves are analytic.)

Fig. 6 indicates that the model is very accurate for multipath networks, and optimistic for large unipath networks. However, most practical networks will almost certainly have some path multiplicity, hence these models can be used.

A more accurate analytic model for the probability distribution function of unipath networks can be found in [26] (this model accounts for the effects of the statistical dependence of requests arriving at one switch). This more refined $pdf$ model can be used in this markov model to create a more accurate approximation for the unipath case [26].

Perhaps the most suprising observation is the performance of the 4-path networks. The 4-path $2^n \times 2^n$ network has a blocking probability of about .007 at $N=1024$; with such a low blocking probability, we would expect that the average number of passes required to realize a permutation would be about 1. However, fig. 6 indicates that two passes (on average) are required at $N=1024$. Similarly, the 4-path $4^n \times 4^n$ banyan has a blocking probability of about .004 at $N=1024$, and yet still 2 passes (on average) are required to realize an arbitrary permutation.

The true performances of $p$-path networks must reflect on the bandwidth of the links used in the network. We assume that each link in a $p$-path $k^n \times k^n$ network has $1/p$ times the bandwidth compared to a link in the unipath $k^n \times k^n$. (The number of pins available on an integrated circuit is limited: by doubling the number of logical input/output links, each link will have half as many pins, and hence will have approximately half the bandwidth.) Hence, assume that each pass in a $p$-path network takes $p$ times as long as a pass in the unipath network (when the switch degree remains the same).

From fig. 6 it is clear that a 2-path network actually outperforms a unipath network by about 25% (for switches of fixed degree 2 or 4). Consider a network of size $N=1024$ made with 2×2 switches. The unipath network will require about 8 cycles to realize an arbitrary permutation, and the 2-path network will require about 3 cycles (each about twice as slow) to realize an arbitrary permutation. However, this estimate can be pessimistic: The unipath network will require about 8 memory access times whereas the 2-path network will require only about 3 memory access times. If the path establishment and data transfer delays are insignificant components of the network cycle time (i.e., the network cycle time is dominated by the memory access time), then the performance improvement could be much higher. Hence, the true performance improvement should also consider the memory access times, plus time spent on path establishment and error checking, etc.

In addition to offering significant performance improvements (when the realization of arbitrary permutations is the performance criterion), the 2-path network is also significantly more fault-tolerant [23].

## 6. Conclusions

We have presented a simple analytic model for the blocking probability of circuit switched MINs when the set of requests submitted during each cycle is a randomly selected permutation. In general, such models are created by adapting an analytic model of a network's blocking probability when the requests are assumed to be random and independent. The techniques are general, and hence analytic models for the performance of arbitrary MINs, under the assumption of unique distributions, are relatively easy to create.

We have presented a general technique to determine an analytic approximation for the expected number of conflict free permutations in multipath MINs (when simple distributed routing algorithms are used). This technique can be used for multipath networks such as the Gamma network [20] and the ADM [1]. No such general technique has been presented in the literature previously, although lower and upper bounds (and in one cases an exact number) on the number of conflict free permutations realizable in certain MINs have been presented. The number of conflict-free permutations realizable by a MIN when distributed routing algorithms are used is a quantitative measure of the connectivity of the network.

We have presented a general technique to determine an accurate analytic approximation for the expected number of passes required to realize an arbitrary permutation in a network, i.e., the universality of a network, when simple distributed routing algorithms are used.

An interesting result concerning network performance was observed. A 2-path network (of degree 2 or 4) will require 3 or fewer passes on average to realize an arbitrary permutation for reasonably sized networks (i.e., network size ≤ 1024). A 2-path network will outperform the corresponding unipath network significantly, and offer significant fault-tolerance improvements aswell. In addition, the performance of a 2-path MIN compares very well with the best known universality of 3 passes through a unipath omega network using global, precomputed routing [21][25][30] (where both networks have switches of degree 2).

These models solve many interesting theoretical problems in the realm of interconnection network modelling. However, these models also have a very practical aspect. They give a quantitative measure of the connectivity of a MIN. Previously, when comparing interconnection networks for their suitablility in SIMD array processors, numerous techniques have been used. One such technique was to estimate the number of cycles one network requires to simulate another [24]. Using the models presented in this paper, one can evaluate the average performance of a particular network analytically (when realization of arbitrary permutations is the performance criterion). However, it must be pointed out that real SIMD machines typically use a small number of permutation classes where the permutations tend to be highly structured, and a better figure of merit would be the network's ability to realize those frequently occurring permutations.

## 7. References

[1] G.B.Adams III and H.J.Siegel, "On the Number of Permutations Performable by The Augmented Data Manipulator", IEEE Trans. Comput., Vol C-31, April 1982, pp. 270-277

[2] D.P. Agrawal, "Graph Theoretical Analysis and Design of Multistage Interconnection Networks", IEEE Trans. Comput., Vol C-32, July 1983, pp. 637-648

[3] G.H. Barnes and S.F. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems", IEEE Computer, Dec. 1981, pp. 31-41

[4] V.E. Benes, "On Rearrangeable Three-Stage Connecting Networks", Bell Sys. Tech. Journal, Sept. 1962, pp. 1481-1492

[5] L.N. Bhuyan and D.P. Agrawal, "Design and Performance of Generalized Interconnection Networks", IEEE Trans. Comput., Vol. C-32, Dec. 1983, pp. 1081-1090

[6] I.F. Blake, "An Introduction to Applied Probability", Wiley, 1979

[7] C. Clos, "A Study of Nonblocking Switching Networks", Bell Sys. Tech. Journal, March 1953, pp. 406-424

[8] G.R. Goke and G.J. Lipovski, "Banyan Networks for partitioning multiprocessor systems", Proc. 1st Annual Symp. Computer Architecture, 1973, pp. 21-28

[9] T. Hsiao-Nan, "RSESS Interconnection Network", Proc. 1985 Intl. Conf. on Parallel Processing, pp. 466-473

[10] C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", IEEE Trans. Comput., Vol C-32, Dec. 1983, pp. 1091-1098

[11] T. Lang, "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network", IEEE Trans. Comput., Vol C-25, May 1976, pp. 496-503

[12] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans. Comput., Vol C-24, Dec. 1975, pp. 1145-1155

[13] D.H. Lawrie, "The Prime Memory System for Array Access", IEEE Trans. Comput., Vol C-31, May 1982, pp. 435-442

[14] K.Y. Lee, "On the Rearrangeability of $2(log_2N)-1$ Stage Permutation Networks", IEEE trans. Comput., Vol C-34, May 1985, pp. 412-425

[15] M-D. P. Leland, "Properties and Comparisons of Multistage Interconnection Networks for SIMD Machines", Ph.D. Thesis, Univ. Wisconson-Madison, 1983

[16] M-D. P. Leland, "On the Power of the Augmented Data Manipulator Network", Proc. 1985 Intl. Conf. on Parallel Pro cessing, pp.74-78

[17] G.F. Lev, N.Pippenger and L.G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks", IEEE Trans. Comput., Vol C-30, Feb. 1981, pp. 93-100

[18] D. Nassimi and S. Sahni, "A Self-Routing Benes Network and Parallel Permutation Algorithms", IEEE Trans. Comput., Vol C-30, May 1981, pp. 332-340

[19] K. Padmanabhan and D.H. Lawrie, "A Class of redundant Path Multistage Interconnection Networks" IEEE Trans. Comput., Vol C-32, Dec. 1983, pp. 1099-1108

[20] D.S. Parker and C.S. Raghavendra, "The Gamma Network: A Multiprocessor Interconnection Network With Redundant Paths", Proc. 9th Annual Symp. on Computer Architecture, 1982, pp. 73-80

[21] D.S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks", IEEE Trans. Comput., Vol C-29, March 1980, pp. 213-222

[22] J.H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors", IEEE Trans. Comput., Vol C-30, Oct. 1981, pp. 771-780

[23] S.M. Reddy and V.P. Kumar, "On Fault-Tolerant Multistage Interconnection Networks", Proc. 1984 Intl. Conf. on Parallel Processing, pp. 155-164

[24] H.J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks", IEEE Trans. Comput., Vol C-26, Feb. 1977, pp. 153-161

[25] D. Steinberg, "Invariant Properties of the Shuffle-Exchange and a Simplified Cost-Effective Version of the Omega Network", IEEE Trans. Comput., Vol C-32, May 1983, pp. 444-450

[26] T.H. Szymanski, Ph.D. Thesis, in preparation, Univ. of Toronto

[27] T.H. Szymanski and V.C. Hamacher, "On the Permutation Capability of Multistage Interconnection Networks", submitted for publication

[28] A. Varma and C.S. Raghavendra, "Realization of Permutations on Generalized Indra Networks", Proc. 1985 Intl. Conf. on Parallel Processing, pp. 328-333

[29] "The Reverse-Exchange Interconnection Network", C-L Wu, T-Y Feng, IEEE Trans. Comput., Vol C-29, Sept. 1980, pp. 801-811

[30] C-L Wu and T-Y Feng, "The Universality of the Shuffle-Exchange Network", IEEE Trans. Comput., Vol C-30, May 1981, pp. 324-331

[31] P-C. Yew, "On the Design of Interconnection Networks for Parallel and Multiprocessor Systems", Ph.D. Thesis, Univ. Illinios at Urbana-Champaign, 1981

# Synthesis of a Family of Cellular Permutation Arrays

By
A. Yavuz Oruc
Electrical, Computer and Systems Engineering Department
Rensselaer Polytechnic Institute, Troy, NY 12180

## Abstract

Group theory provides a convenient means for formulating and solving some nontrivial problems in permutation network theory. This paper uses group theoretic techniques to provide a unified approach for cellular permutation array design. More specifically, the networks of Kautz et al and Bandyopadhyay et al are shown to be geometric reperesentations of iterative coset decompositions of symmetric groups. This result was used by Oruc and Oruc to develop a linear-time set up algorithm for cellular permutation arrays. This paper examines another application, namely, the design of a family of cellular permutation arrays.

## 1. Introduction

Permutation networks have been extensively investigated in the parallel processing literature. By now, the researchers in the field know how to design such networks with aysmptotically optimal cell counts[1], and extensive research results have also appeared about how these networks are incorporated into parallel processing[2]. The objective of the present paper is to exhibit some interesting relations between a family of permutation networks known as cellular permutation arrays, and coset decompositions of symmetric groups. Cellular permutation arrays were investigated primarily by Kautz et al[3] and Bandyopadhyay et al[4]. Unlike optimal permutation networks, cellular permutation arrays use redundant cells, and thus they are not optimal in terms of cell counts. Nevertheless, they have certain attractive features. One such feature is that the links or wires connecting the programmable cells in these networks are local and require no crossing. More significantly, it was recently shown[5] that an n-input cellular permutation array can be set up in O(n) time, whereas the best known set up time for an n-input Benes network is $O(n\log_2 n)$.

These properties of cellular permutation arrays call for a careful investigation of such networks. As an effort in this direction, we show that cellular permutation arrays proposed in the papers [3,4] can all be characterized as iterative decompositions of symmetric groups into cosets. A direct consequence of this fact is a parametrized design technique for cellular permutation arrays, that is, a design technique by which it is possible to specify the cell sizes and permutations based on the design constraints in question. In the remainder of the paper we shall formalize these ideas and elaborate on them in detail.

## 2. Network Characterization

We first establish the relation between the coset decompositions of symmetric groups and cellular permutation arrays. Let $\Sigma_i$ denote the symmetric group over the set of symbols $(1,2,\ldots,i)$; $1\le i\le n$. It can be shown that[6]:

$$\Sigma_i = \Sigma_{i-1}e+\Sigma_{i-1}(1\ i)+\Sigma_{i-1}(2\ i)\ldots+\Sigma_{i-1}(i-1\ i) \quad (1)$$

for all i; $3\le i\le n$ where $\Sigma_2 = (e,\ (1\ 2))$, and e denotes the identity permutation. Similarly,

$$\Sigma_i = e\Sigma_{i-1}+(1\ i)\Sigma_{i-1}+(2\ i)\Sigma_{i-1}\ldots+(i-1\ i)\Sigma_{i-1} \quad (2)$$

(1) and (2) are, respectively, the iterative right and left coset decompositions of $\Sigma_n$ into $\Sigma_{n-1}$, $\Sigma_{n-2}$, ..., $\Sigma_2$. In both cases, the map e and transpositions $(i\ j)$; $1\le i\le j-1$ are the right (or left) coset leaders of $\Sigma_{j-1}$ in $\Sigma_j$; $3\le j\le n$. Moreover, these two decompositions are the group-theoretic representations of what are known as regular and reverse KLW permutation arrays[3,6]. Fig. 1 depicts the two networks for n=4. If each cell in the either network can perform the only two permutation maps possible over its inputs as indicated, then one easily verifies the firm equivalence between regular and reverse KLW permutation arrays and the iterative decompositions given in (1) and (2).

One can also decompose $\Sigma_n$ into $\Sigma_i$; $2\le i\le n-1$ by using cycle maps as coset leaders instead of transpositions. That is, it can be shown that[6]:

$$\Sigma_i= \Sigma_{i-1}e+\Sigma_{i-1}(i\ i-1)+\Sigma_{i-1}(i\ i-2\ i-1)\ldots+ \Sigma_{i-1}(i\ \ 1\ 2\ldots i-1) \quad (3)$$

and,

$$\Sigma_i= e\Sigma_{i-1}+(i\ i-1)\Sigma_{i-1}+(i\ i-1\ i-2)\Sigma_{i-1}\ldots+ (i\ i-1\ldots 2\ 1)\Sigma_{i-1} \quad (4)$$

Equations (3) and (4) are the group-theoretic representations of what are known, respectively, as regular and reverse BBC networks[4,6]. As an example, we depict these two networks for n=5 in Fig. 2. The cycle maps shown inside the cells with three or more inputs are coset leaders while the cell with two inputs in each network corresponds to $\Sigma_2 =(e,\ (1\ 2))$.

A direct implication of the four decompositions stated above is that the corresponding networks are symmetric, i.e., they can realize all of $\Sigma_n$, and that the networks can be set up for an arbitrary permutation in linear time. The reader may refer to [5] for a detailed description of the set up procedures. Another consequence of these characterizations of cellular permutation arrays is a parametrized design technique which we shall consider in the following section.

0190-3918/86/0000/0324 $01.00 © 1986 IEEE

## 3. Network Design

It should be noted that the KLW and BBC networks described above are two extreme examples of realizing symmetric groups by iterative coset decompositions. They are extreme in the sense that while the KLW networks use $n(n-1)/2 = O(n^2)$ cells, BBC networks need n-1 or O(n) cells to construct. Of course, the complexity of the cells in the two cases are quite different. In any event, however, we may be subject to certain physical constraints in realizing $\Sigma_n$ by a cellular permutation array. For example, it may be cost ineffective to place each cell of a KLW network in a separate chip, and yet it may be infeasible to contain all of the permutations of a large cell of a BBC network within a single chip. Thus, what is needed is a design technique to realize symmetric groups by cellular permutation arrays subject to the specifications of the coset leaders, and the size and the number of cells to be used by the network in question.

We can formalize the above problem as follows. Given positive integers n and m; $2 \leq m \leq n$; design an n-input cellular permutation array consisting of N(n,m) k-input cells where $2 \leq k \leq m$ and $n-1 \leq N(n,m) \leq n(n-1)/2$. We shall describe two constructions both with pseudo triangular geometries.

In the first construction[7], we shall permit cells with 2, 3, 4,..., up to k inputs. The pseudo triangular permutation network is then directly obtained by grouping together the permutations of 2-input cells of a KLW network columnwise into 2, 3, ..., up to k-input cells. As an example, such a network for n=11 is constructed as shown in Fig. 3. It is easily verified that the column of cells with the jth vertical input in the network can perform all of the transpositions (i j); $1 \leq i \leq j-1$, and hence the symmetricity of the network immediately follows. It can also be shown that the number of cells in an n-input pseudo triangular permutation array which is so constructed is

$$N(n,m) = (f+1)(n-1-(m-1)f/2) \qquad (5)$$

where $f = L\ n-1/m-1$ . Finally, we note that, instead of transpositions, one can easily map the cycles of BBC networks into the columns of this type of cellular permutation array after some algebraic manipulations. Whether one uses transpositions or cycles is an implementation problem which can be investigated independently from the considerations discussed here.

In the second construction we shall assume that all cells have the same number of inputs. This assumption leads to a pseudo triangular cellular permutation array which is depicted in Fig. 4 for n=11. In this case, we must let the leftmost cell generate $\Sigma_m$ over its inputs. The cells in subsequent columns will then need only to realize the required coset leaders. The reader can again verify that the column with the jth vertical input can be made to realize all of the transpositions (i j); $1 \leq i \leq j-1$. Thus, the symmetricity of the network is guaranteed. The construction is easily generalized to n inputs, and furthermore it can be shown that such a network consists of N(n,m) cells where:

$$N(n,m) = (f+1)(n-m+1-(m-1)f/2) + g \qquad (6)$$

where $f = L\ m-m+1/m-1$ and g=1 if $n-m+1 \bmod m-1 \neq 0$ and g=0 otherwise.

The above network constructions underscore the degrees of freedom in cellular permutation array design. We shall not describe the detailed implications of these constructions here. It should be obvious how one can manipulate the formulas in (5) and (6) to explore various design alternatives subject to constraints on n, m and N(n,m).

## 4. Summary and Conclusions

The paper has established the structural equivalence between cellular permutation arrays and coset decompositions of symmetric groups. Based on this equivalence, we have constructed two new families of cellular permutation arrays, and shown that BBC and KLW networks are special cases of these networks. We have also outlined a parametrized design technique which can be used to synthesize cellular permutation arrays when there are certain constraints on the cell sizes and permutations. We note that the networks described here all have triangular geometries, a fact that follows from the linear iterative nature of the symmetric group decompositions considered in the paper. It will be worthwhile to determine whether there exist other decompositions of symmètric groups which may lead to new families of cellular permutation arrays.

## REFERENCES

[1] V. E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, New York 1965.

[2] T. Feng, "A survey of interconnection networks," Computer, Vol. 14, pp. 12-27, Dec. 1981.

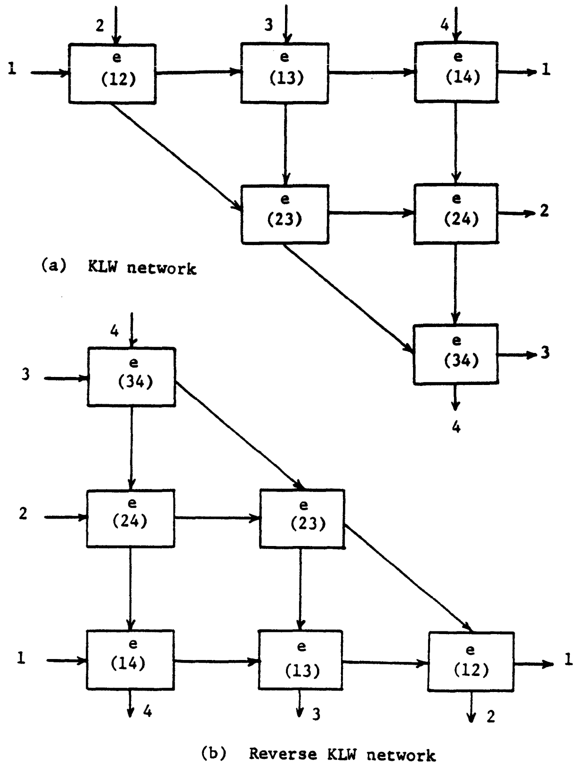
[3] W. H Kautz, K. N. Levitz and A. Waksman, "Cellular interconnection arrays," IEEE Trans. Comput., pp. 443-451, May 1968.

[4] S. Bandyopadhyay, S. Basu and K. Choudhury, "A cellular permuter array," IEEE Trans. Comput., pp. 1116-1119, Oct. 1972.

[5] A. Y. Oruc and M. Y. Oruc, "Linear-time algorithms for programming cellular permutation arrays," In Proc. of ACM Nat'l Comp. Sci. Conf., pp. 129-136, Cincinnati, OH, 1986.

[6] A. Y. Oruc, "Symmetric groups, cosets and cellular permutation arrays," To appear in Proc. of the 20th Inform. Sci. and Sys. Conf, Princeton, NJ, 1986.

[7] S. A. Nadkarni, "The design and performance evaluation of hybrid cellular permutation arrays," M.Sc. Thesis, Rensselaer Poly. Ins., Troy, NY, Aug. 1985.



(a) KLW network

(b) Reverse KLW network

Figure 1. 4-input Triangular Networks

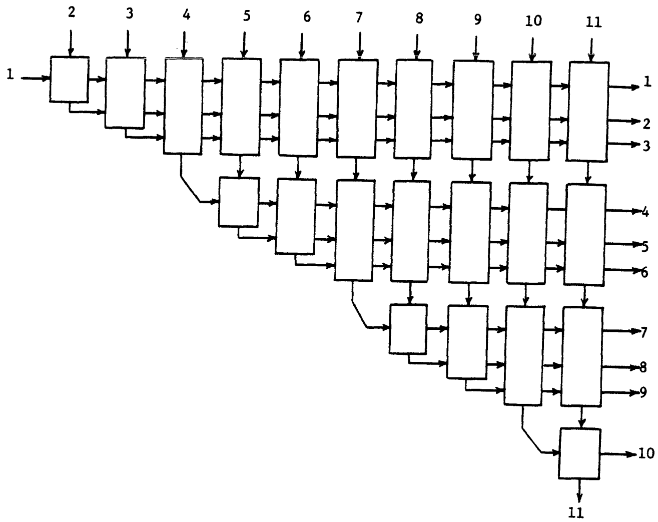


Figure 3. Pseudo Triangular Permutation Network, Construction 1; n=11, m=4.

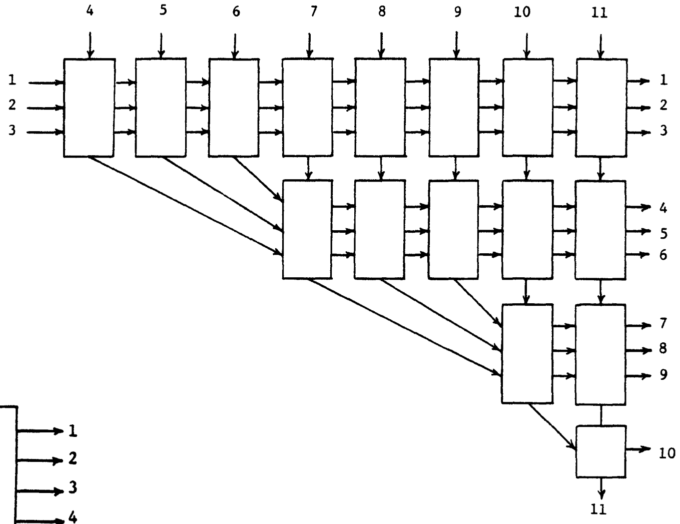
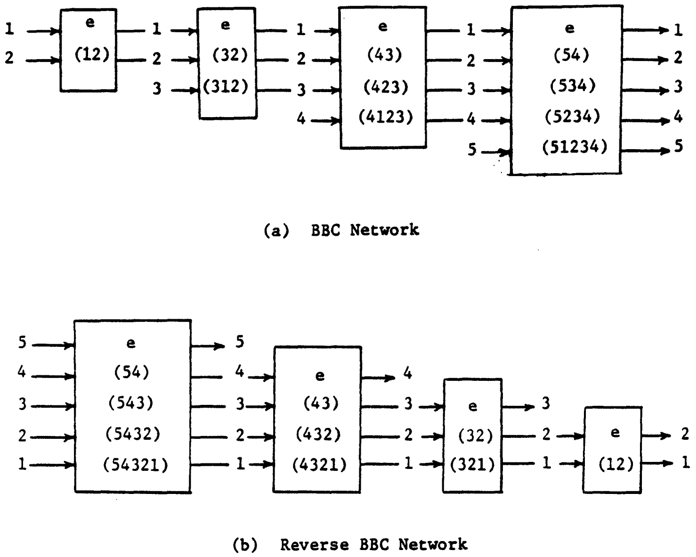


Figure 4. Pseudo Triangular Permutation Network, Construction 2; n=11, m=4.



(a) BBC Network

(b) Reverse BBC Network

Figure 2. 5-input Cascade Networks

# A FAULT-TOLERANT INTERCONNECTION NETWORK SUPPORTING THE FETCH-AND-ADD PRIMITIVE

*Abhijeeet Dugar and Prithviraj Banerjee*

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL-61801

## ABSTRACT

Fault-tolerant network architectures for multiprocessor systems are emerging as an important area of study. Theoretical studies of algorithms, simulation of the NYU Ultracomputer architecture and construction of a small prototype machine led to a joint proposal by NYU and IBM (Yorktown Heights) to implement the architecture in a system with up to 512 processors, including an implementation of the "Fetch-and-Add" shuffle network that is key to the success of this architecture. Techniques proposed so far to achieve fault-tolerance in interconnection networks cannot support the "Fetch-and-Add" primitive satisfactorily. In this paper, we present a multistage interconnection network based on the Omega network that supports the "Fetch-and-Add" instruction and also provides fault-tolerance. Basically, the approach uses 4×4 switches as switching elements in a multistage network, and uses an extra stage of such switches to enable four independent paths to be set up between any source and any destination. Conventional approaches to fault tolerance in interconnection networks involve using only one of the paths at a particular time for propagation of a message, with the redundant paths being used only if failure is detected in the first path. In our scheme, we propose transmisson of four copies of a message through the network simultaneously, with voting being performed on the copies at the memory network interface. The design of the switching elements constituting the network is described. Properties of the Omega network and a rigorous scheduling discipline enforced in the implementation of the switches in the proposed network make it possible to send four copies of every message synchronously through the network. This allows concurrent correction and detection of message transmission errors.

## 1. INTRODUCTION

The current interest in large scale parallel processors has motivated a large amount of research on multistage interconnection networks for parallel processing. Buffered interconnection networks are an integral component of several machines currently under development, including the Cedar machine at the University of Illinois [1], the NYU Ultracomputer [2] at the New York University and the RP3 at IBM [3].

With the advent of VLSI technology, it has become cost-effective to include a large number of processors and memory modules in a multiprocessing system. The hardware system organization is determined primarily by the interconnection structure used between the memory modules and the processors. Many multistage interconnection networks have been proposed such as the baseline [4], delta [5], Generalized Cube [6], Omega [7], STARAN flip [8], etc. for parallel/distributed systems. The Generalized Cube is representative of these networks in that they are topologically equivalent to it.

An important criterion for estimating the performance of a multiprocessor system is its reliability. To a large extent the reliability of the system depends on that of the interconnection network. A fault-tolerant interconnection network can tolerate faults to some degree and still provide reliable communication between any input-output pair. Some amount of redundancy has to be present to achieve fault-tolerance. In what is known as information redundancy, error detecting/correcting codes are used [9]. Such schemes require minimal additional hardware but fault-tolerance is limited to the data being transferred. Faults in the control portion of the network may not be tolerated. Another approach is hardware redundancy in which multiple paths are created between the inputs and outputs of the network. Multiple paths can be created by the addition of an extra stage to the multistage network as in [10], [11], [12], or by providing redundant links as in [13], and [14]. The INDRA network [15] uses a redundant stage as well as redundant links.

The schemes proposed so far consider the design of the interconnection network at a very high level or on a totally theoretical basis. Also, the designs proposed so far are not aimed at any specific multiprocessor system. In this paper we propose an interconnection network suited for the Research Parallel Processor Prototype (RP3) project [3] initiated in the IBM Research Division in cooperation with the Ultracomputer Project of NYU [2].

The proposed network supports the Fetch-and-Add primitive of the NYU Ultracomputer. The Fetch-and-Add instruction is an interprocessor synchronization operation. It permits highly concurrent execution of operating system primitives and application programs. The advantage of Fetch-and-Add over Test-and-Set and Compare-and-Swap to serialize locking and reservation operations in database applications has been discussed in [16]. The format of this instruction is F&A(X,a), where $X$ is an integer variable and $a$ is an integer expression. This is an indivisible operation and it is defined to return the old value of $X$ and to replace $X$ by the sum of $X+a$. The Fetch-and-Add operation follows the serialization principle: If $X$ is a shared variable and many Fetch-and-Add operations simultaneously address $X$, the effect of these operations is as though they occurred in some (unspecified) serial order.

In Section 3, we present in detail the design of the switching elements constituting the proposed interconnection network. In Section 4 we prove certain properties of the network which makes it possible to achieve fault-tolerance.

0190-3918/86/0000/0327 $01.00 © 1986 IEEE

## 2. OVERVIEW OF PROPOSED NETWORK

The proposed interconnection network is based on the Omega network [7]. An Omega network with N inputs and N outputs consists of $n = \log_B N$ stages of $B \times B$ switching elements. A $B^n \times B^n$ Omega network is constructed using $B \times B$ switching elements and $B{*}B^{n-1}$ shuffles interconnecting the stages. A $P{*}Q$ shuffle is the permutation of $PQ$ elements defined as

$$\pi(i) = \left\lfloor Pi + \left\lfloor \frac{i}{Q} \right\rfloor \right\rfloor_{mod\ PQ} \qquad 0 \leqslant i \leqslant PQ-1$$

In the proposed network, 4x4 switching elements are used (B=4). By adding an extra stage to the Omega network obtained as defined above, redundant paths between the processors and memory modules are created. 4x4 switches are used instead of 2x2 switches so that four redundant paths can be obtained. Hence, it consists of ($\log_4$N + 1) stages. It has been proven in [17] that the four paths created are unique i.e. they use independent links and independent switching elements at every stage of the network except at stage 0 and stage $n$ ($n$ = $\log_4$N) of the network. To make the first and last stages also fault-tolerant, the 4x4 switch can be modified as explained in Sections 3.2 and 3.3. In our fault model, a fault constitutes:

(1) Complete failure of one switching element.

(2) Failure of one link between any two stages of the network.

Conventional approaches to fault-tolerance in interconnection networks [10, 13, 14, 17] involve the use of disjoint redundant paths between any source/destination pair. Once a failure of a switch/link in the path of a message is detected, an alternate path is selected. The mechanism of detection of a failure can be performed either off-line through diagnosis [4] or on-line by using error correcting codes [9]. Off-line diagnosis is not applicable in case of transient or intermittent failures which have been shown to occur more frequently than permanent failures [18, 19]. An on-line fault detection/location mechanism may use coding techniques. Simple algebraic codes such as Hamming codes or cyclic codes [20] can be used for detecting errors in the address portion $X$ of a message $(X, a)$ but not for the data portion $a$ because the Fetch-and-Add primitive requires arithmetic operation to be performed on the data portion of the message. For the data portion, arithmetic codes such as the residue codes and the AN codes would have to be used [20]. However, these codes may be used for detecting errors but not for correcting them. Even if the codes are used only for error detection, implementation of the encoding/decoding circuitry is very complex. Besides, such circuits would have to be designed to be totally self-checking. Considering that such circuits would have to be provided in each switching element of the network, it is not cost-efficient to adopt this scheme. A less costly technique would be to place the encoding circuitry at the PNI and the decoding circuitry at the MNI. However, in this technique, once an error is detected, the message has to be sent through an alternate path and it may be impossible to recover from the deleterious effects produced by the erroneous message. Consider what happens if message $(X, a)$ changes to $(Y, c)$. This could have several effects:

(1) A wrong value would be returned to the processor from which $(X, a)$ originated.

(2) Memory location $X$ will not be updated.

(3) Memory location $Y$ will be written into when it should not have been written into.

(4) If there is a message $(Y, b)$ also propagating through the network, combining of $(X, a)$ (changed to $(Y, c)$) and $(Y, b)$ may take place when no combining should have taken place. As a result, a wrong value would be returned to the processor from which $(Y, b)$ originated and a wrong value would be written into memory location $Y$.

(5) In case there are several messages propagating through the network with destination $Y$, the effect of combining the erroneous message $(X, a)$ (changed to $(Y, c)$) with all these messages may be disastrous [*]. Recovery from the effects of combining in that case may be extremely difficult or even impossible since combining of the erroneous message $(X, a)$ with non-erroneous messages may have taken place at various stages of the network.

Thus, due to a single fault, several errors may be generated. In the proposed scheme, all four paths are used to send four copies of a message simultaneously through the network. Voting is carried out on the four copies in the Memory Network Interface. This enables correction of message transmission errors due to any number of faults along a single path, or, detection of message transmission errors due to any number of faults along two paths, as messages propagate through the network.

## 3. DETAILED NETWORK DESIGN

This section describes in detail the design of the switching elements constituting the network. First, message transfers within and between stages is described for stages 1 through $(n-1)$ and then for stages 0 and $n$.

### 3.1. Stages 1 through $(n-1)$

Each switch has four input and four output ports. One queue is associated with each input-output port pair. Hence, there are four IN queues associated with each output port (Fig.1). These are labeled as IN0, IN1, IN2, and IN3. IN0 is connected to input port 0, IN1 to port 1 and so on. When a message arrives at an input port it is first determined which output port the message is bound for by decoding the relevant two bits in the destination address of the message. Then the message is placed in the appropriate IN queue of the particular output port. Messages are sent to the next stage of the network depending upon their priority. The priority is established by following a round-robin discipline, starting with the queue having the smallest label. This round-robin scheduling scheme is illustrated by the following example.

EXAMPLE 1: Messages $a, b, c, d, e, f, g, h, i, j$ arrive at various input ports of a switch over a period of five cycles and are routed to output port p as shown in Fig.2. For the purpose of this example assume that none of these messages can be combined. Messages $a$, $b$ and $c$ arrive in the same cycle at input ports 0, 2 and 3 respectively. Suppose this is cycle C. Messages $d$ and $e$ arrive in cycle C+1 at ports 1 and 3 respectively. In cycle C+2 messages $f$, $g$, $h$, and $i$ arrive at input ports 0, 1, 2, and 3 respectively and are routed to output port p. In cycle C+3 message $j$, arriving at input port 3, is routed to output port p. These messages are sent to the next stage in ten cycles in the order $a, b, c, d, e, f, g, h, i$, and $j$. Note that after message $c$ is sent to the next stage, message $d$ is sent to the next stage and not message $f$. Any blank slots in the IN queues are skipped while following the round-robin scheduling of messages. □

[*] The possibility of there being several requests to the same memory location simultaneously has been discussed in [21].

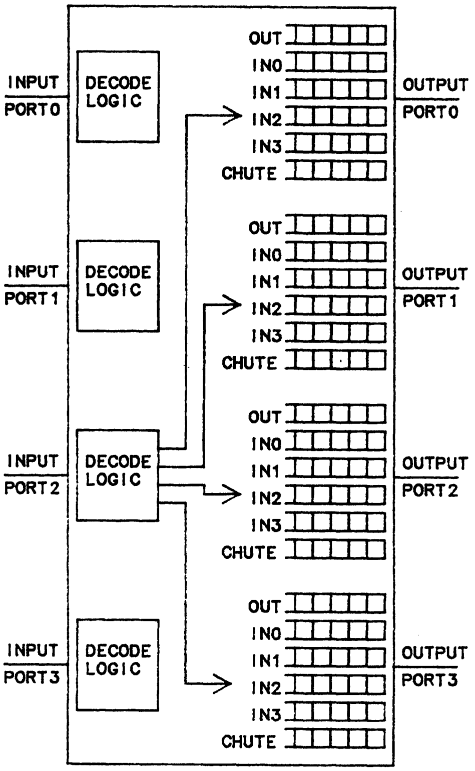


Fig. 1. *A 4x4 Switch in an Intermediate Stage of the Network*

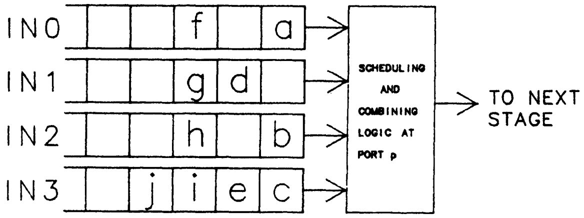


Fig. 2. *Round-Robin Scheduling Discipline at Output Port of a Switch in the Network*

Implementation of the correct ordering of the messages and possible combining of messages is based on the scheme proposed in [22]. For the benefit of the reader, the scheme proposed in [22] is now described briefly. The scheme has been used for a network consisting of 2x2 switching elements called Ultraswitches. Three columns of shift registers called the IN column, the OUT column and the CHUTE column are associated with each output port of a switch. These columns of shift registers are connected as shown in Fig.3. Messages arrive at the IN column and shift up one position at each cycle. Similarly, messages shift down one position in the OUT column at each cycle. If a message in the IN column is adjacent to a slot on the OUT column that is empty, then it shifts to that slot. In addition, this scheme detects a message in the IN queue going to the same address as another message already in the OUT queue. The message in the IN queue is then placed in the CHUTE column. The two messages move synchronously and arrive at the combine logic simultaneously. The combine logic detects the possibility of combining and combines the two messages so that only one message is sent to the next stage of the network.

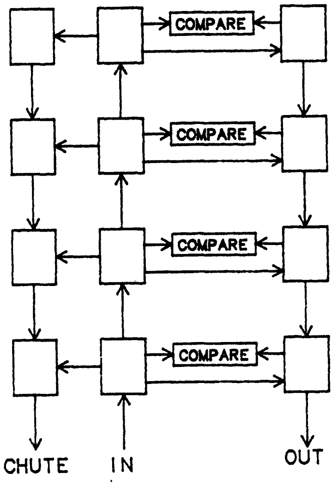


Fig. 3. *Design of the Output Port of the Ultraswitch*

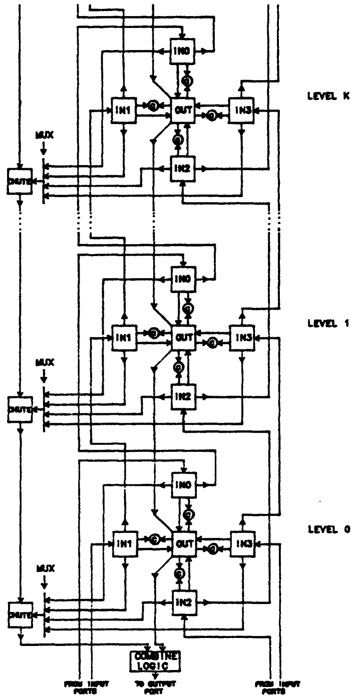


Fig. 4. *Design of the Output Port of a Switching Element in the Proposed Scheme*

In the proposed scheme, at each output port there are four IN queues labeled 0, 1, 2, and 3 as described earlier. In addition, at each output port there is one OUT queue and one CHUTE queue. These queues are connected as shown in Fig.4. Possible movements of messages is indicated by the arrows. Control signals have been omitted from the figure for the sake of clarity. A message arriving at an input port is placed in the appropriate IN queue of an output port at level 0. Messages in the IN queues then move to the OUT queue, to the CHUTE (if combining is possible), or one level up in the IN queue. The movements of messages in one cycle can be described by dividing the cycle into three phases:

**Phase One:** The messages at level 0 of the OUT queue and the CHUTE are sent to the combine logic. The function of the combine logic is the same as in the scheme proposed in [22]. Messages arriving at the four input ports are placed at level 0 of the appropriate IN queues. Messages already in the IN queues move to the OUT queue, the CHUTE or the next level up in the IN queue. The movement of messages is decided according to the following conditions:

(1) If the OUT queue slot is empty at any level, the message at that level in the IN queue wih the smallest label moves into the OUT queue at that level. The other messages at that level move up one level.

(2) At every level, if the OUT queue slot is full and the CHUTE slot is empty, then the messages in the IN queues are compared with the message in the OUT queue at that level to find out if any of the messages in the IN queues can be combined with the message in the OUT queue. From the result of the comparison if it is found that none of the messages in the IN queues can be combined with the message in the OUT queue at a level, then all messages in the IN queue at that level move up one level. If one of the messages can be combined, that message is moved to the CHUTE at that level. The rest of the messages at that level move up one level. If more than one message can combine with the message in the OUT queue, then among these messages the message in the IN queue with the smallest label is moved to the CHUTE at that level. The rest of the messages at that level move up one level.

(3) If OUT queue slot is full at a level and the CHUTE slot is also full at that level then all messages at that level move up one level.

**Phase Two:** Messages in the OUT queue and the CHUTE do not move. Messages in the IN queues move to the OUT queue, the CHUTE or the next level up in the IN queue. Movement of the messages in the IN queues is decided according to the conditions described for phase one.

**Phase Three:** Messages in the OUT queue and the CHUTE do not move. Messages in the IN queue may move only within the level they are in; they cannot move to a higher level. If any message in an IN queue can be combined or moved to the OUT queue it is moved to the CHUTE or the OUT queue at its own level.

EXAMPLE 2: The movement of messages in the three phases is illustrated in Figs.5a and 5b. During four clock cycles, messages *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i* and *j* reach an output port p. Assume that none of these messages can be combined. It is easy to extend the example when combining is possible. Hence, only IN queues and the OUT queue at output port p are shown. Messages *a*, *b*, *c*, *d* arrive at the same time at the four input ports of the switch and are all bound for output port p. They are placed in the four IN queues associated with output port p at level 0, in phase one of cycle one. In phase two of cycle one, message *a* moves to the out queue, and messages *b*, *c*, *d* move up one level. In phase three of cycle one, message *b* moves to the OUT queue. Messages *c* and *d* do not move. This is the end of the first cycle. In phase one of the second cycle, message *a* is sent to the next stage; message *b* moves one level down in the OUT queue; message *c* moves to the OUT queue; message *d* moves up one level; messages *e* and

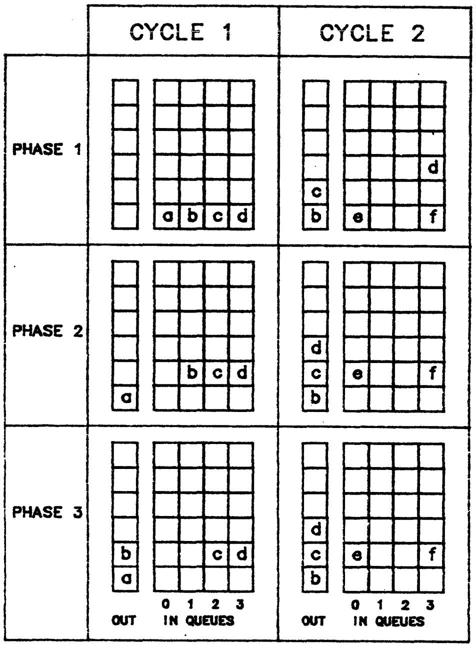


Fig. 5a. *Movement of Messages in the Three Phases of a Cycle.*

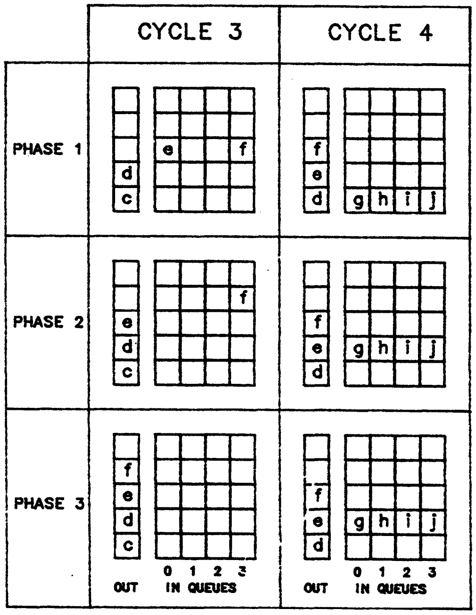


Fig. 5b. *Movement of Messages in the Three Phases of a Cycle.*

$f$ at input ports 0 and 3 of the switch are placed in IN queues 0 and 3 at level 0. Further movements of messages are also depicted in Fig.5. □

Synchronization of the the movement of messages in the three phases can be achieved by using a suitable clock. When different switches receive clock signals by different paths, they should receive clocking events at the same time. Synchronization errors due to clock skews can be avoided by lowering clock rates and/or adding delay to the circuits. Another way is to take advantage of the propagation delay down a long wire by having several clock cycles in progress along its length. This behaviour can be simulated by replacing long wires with strings of buffers. This will restore signal levels and prevent backward noise propagation [23].

### 3.2. Stage 0

Stage zero is the extra stage of the network. Logically, stage zero switches have four inputs and four outputs. A message arriving at any input port in this stage is broadcast to all four output ports of the switch. Thus a message arriving at input port 2 of a switch in stage zero is put into IN queue 2 of each output port of the switch. It is then sent to stage one as described above.

Physically, the replication of messages can be done by the Processor Network Interface (PNI) logic. Thus stage zero receives 4N inputs and it consists of N 4x1 switches i.e. one switch for each output of stage zero. The four IN queues associated with each output port are in one switch. Thus, each switch has four IN queues, one OUT queue and one CHUTE queue as shown in Fig.6. The four copies of a message enter stage zero at different 4x1 switches and are placed in the appropriate IN queues. Movement of messages and any possible combining can go on as in stages 1 through $n-1$. Essentially, the 4x4 switch has been broken up into four 4x1 switches so that each is on a separate chip. Failure of any one of these chips can be tolerated.

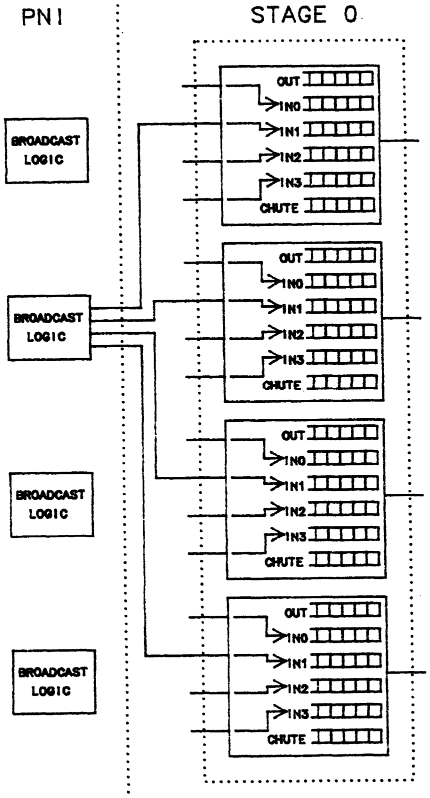


Fig. 6. *Physical Implementation of a 4x4 Switch in Stage 0 of the Network.*

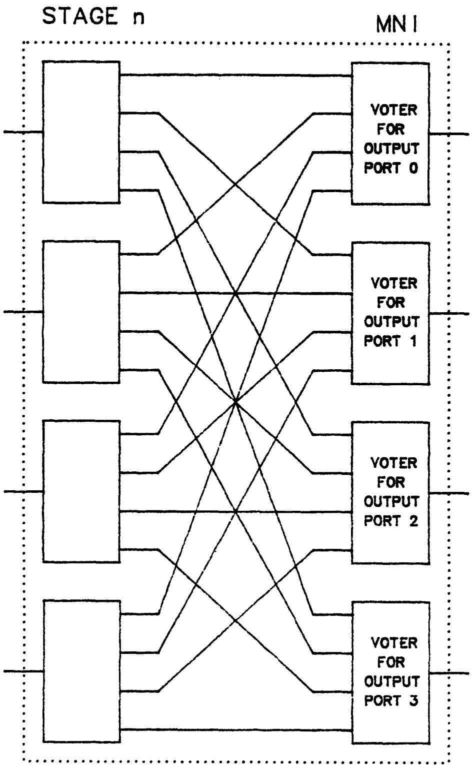


Fig. 7. *Physical Implementation of a 4x4 Switch in Stage n of the Network.*

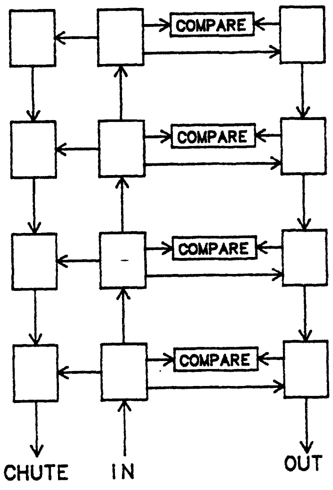


Fig. 8. *Output Port of a Switch in Stage n of the Network.*

### 3.3. Stage n

Logically, each switch in the last stage of the network (stage $n$) receives the four copies of a message, one copy arriving at each input port. The topology of the network and the scheduling of the movement of messages guarantees that the four copies arrive simultaneously at the last stage. Voting is carried out on the four copies. If at most one copy is erroneous, the correct message is routed to the appropriate output port. If two copies are erroneous, the error is detected.

In the physical design of the network, voting can be carried out by the Memory Network Interface (MNI). Each switch in the last stage has one input and four outputs. The physical implementation of a logical 4x4 switch in the last stage is depicted in Fig.7. There are N such 1x4 switches. In each switch, there are four OUT queues, one for each output port. Each output port of a switch has one OUT queue, one IN queue and one CHUTE queue. These queues are connected as shown in Fig.8. The four copies of a message arrive at four 1x4 switches. A message arriving at the input port of a switch is placed in one of the IN queues depending upon the address of its destination. Two messages in the same queue may combine if their destination address within a module are same. Each output of the N 1x4 switches is hardwired to a voting unit in the MNI. There are N such voting units, each being hardwired to four outputs of stage $n$ on the network side and to one memory module on the other side. The MNI receives the four copies, carries out the voting, and routes the message to the appropriate memory module.

## 4. NETWORK PROPERTIES

In this section we will show that the four copies of every message move synchronously through the network. This will be proved by making use of the topology of the network and the scheduling discipline described in Section 3. For the purpose of this section, the switches in stages 0 and $(n-1)$ will be considered to be 4x4 switches.

NOTATION : To distinguish between the four copies of a message, we introduce the notation $(X,a)^0$, $(X,a)^1$, $(X,a)^2$, $(X,a)^3$, to denote the four copies of a message $(X,a)$.

NOTATION : A switch $i$ in stage $s$ of the network will be denoted by $(s,i)$.

LEMMA 1: All copies of a message enter different switches in stages 1 through $(n-1)$ at the same input port of the switches.

PROOF: The inputs and outputs of all the stages are labeled from 0 to N-1 in binary as $m$-bit addresses ($m$ = $\log_2$N). Suppose a message originates at processor

$$a_0\, a_1\, a_2 \dots\, a_{m-2}\, a_{m-1}$$

and the address of its destination is

$$x_0\, x_1\, x_2 \dots\, x_{m-2}\, x_{m-1}.$$

The path of the message through the network is traced in Table 1. It can be seen from Table 1 that for every stage between 1 through $(n-1)$ of the network, the least two significant bits of the address in the input port address column are same for all the copies. Since these two bits specify the input port within a switch, the four copies of a message do enter at the same input port of different switches in the same stage. □

LEMMA 2: In an intermediate stage, if the four copies of a message $(X,a)$ enter a stage at switches $(s,i)$,$(s,j)$,$(s,k)$, and $(s,l)$ at port p and one copy of a message $(Y,b)$ enters switch $(s,i)$ at port q, then the other copies of $(Y,b)$ must enter at switches $(s,j)$,$(s,k)$, and $(s,l)$ at port q.

Table 1

| Stage No. | Address of Input Port | Address of Output Port |
|---|---|---|
| Stage 0 | $a_2a_3a_4a_5 \cdots a_{m-2}a_{m-1}a_0a_1$ | $a_2a_3a_4a_5 \cdots a_{m-2}a_{m-1}00$<br>$a_2a_3a_4a_5 \cdots a_{m-2}a_{m-1}01$<br>$a_2a_3a_4a_5 \cdots a_{m-2}a_{m-1}10$<br>$a_2a_3a_4a_5 \cdots a_{m-2}a_{m-1}11$ |
| Stage 1 | $a_4a_5a_6 \cdots a_{m-2}a_{m-1}00a_2a_3$<br>$a_4a_5a_6 \cdots a_{m-2}a_{m-1}01a_2a_3$<br>$a_4a_5a_6 \cdots a_{m-2}a_{m-1}10a_2a_3$<br>$a_4a_5a_6 \cdots a_{m-2}a_{m-1}11a_2a_3$ | $a_4a_5 \cdots a_{m-2}a_{m-1}00x_0x_1$<br>$a_4a_5 \cdots a_{m-2}a_{m-1}01x_0x_1$<br>$a_4a_5 \cdots a_{m-2}a_{m-1}10x_0x_1$<br>$a_4a_5 \cdots a_{m-2}a_{m-1}11x_0x_1$ |
| ... | ... | ... |
| Stage n-1 | $00x_0x_1x_2x_3 \cdots x_{m-6}x_{m-5}a_{m-2}a_{m-1}$<br>$01x_0x_1x_2x_3 \cdots x_{m-6}x_{m-5}a_{m-2}a_{m-1}$<br>$01x_0x_1x_2x_3 \cdots x_{m-6}x_{m-5}a_{m-2}a_{m-1}$<br>$11x_0x_1x_2x_3 \cdots x_{m-6}x_{m-5}a_{m-2}a_{m-1}$ | $00x_0x_1 \cdots x_{m-4}x_{m-3}$<br>$01x_0x_1 \cdots x_{m-4}x_{m-3}$<br>$10x_0x_1 \cdots x_{m-4}x_{m-3}$<br>$11x_0x_1 \cdots x_{m-4}x_{m-3}$ |
| Stage n | $x_0x_1 \cdots x_{m-6}x_{m-5}x_{m-4}x_{m-3}00$<br>$x_0x_1 \cdots x_{m-6}x_{m-5}x_{m-4}x_{m-3}01$<br>$x_0x_1 \cdots x_{m-6}x_{m-5}x_{m-4}x_{m-3}10$<br>$x_0x_1 \cdots x_{m-6}x_{m-5}x_{m-4}x_{m-3}11$ | $x_0x_1 \cdots x_{m-3}x_{m-2}x_{m-1}$ |

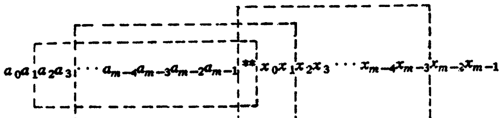

$a_0a_1a_2a_3 \cdots a_{m-4}a_{m-3}a_{m-2}a_{m-1}$ ** $x_0x_1x_2x_3 \cdots x_{m-4}x_{m-3}x_{m-2}x_{m-1}$

Fig. 9. *Tracing the Path of a Message from the Address of its Source and Destination.*

PROOF: This can be proved by first determining the condition under which two messages originating from different processors will meet at a switch in a stage $s$. Consider a standard Omega network (no extra stage). Suppose a message originates at processor

$$a_0\, a_1\, a_2 \dots\, a_{m-2}\, a_{m-1}$$

and the address of its destination is

$$x_0\, x_1\, x_2 \dots\, x_{m-2}\, x_{m-1}.$$

The path of the message can be traced by selecting an $m$-bit window at each stage from the concatenated source and destination address bits (Fig.9). At succesive stages, the window is shifted right by two bits. For two messages to meet at a stage the most significant $m-2$ bits in the windows of the two messages for that stage must match. These bits specify the switch at which the two messages meet in the stage. From Fig.9 and Table 1 ("Address of Output Port" column) it can be seen that when an extra stage is added to the network, the $m$-bit window is chosen from

$$a_0\, a_1 \dots\, a_{m-1} ** \,x_0\, x_1 \dots\, x_{m-1}$$

in any intermediate stage, where ** is 00, 01, 10 and 11. The four windows corresponding to the four values of ** identify the particular copy of a message. Thus, if one copy of two messages meet at a switch, the other copies must also meet. □

LEMMA 3: The four copies of every message exit stage 0 of the network simultaneously.

PROOF: From the design of stage 0 switches discussed in Section 3.2 we know that for a message $(X,a)$ arriving at an input port p of switch $(0,i)$, its copies $(X,a)^0$, $(X,a)^1$, $(X,a)^2$, and $(X,a)^3$ will be placed in IN queue p at each output port of switch $(0,i)$ at the same time. The round-robin scheduling discipline ensures that all the copies have the same priority in being sent to the OUT queue or to the CHUTE of the respective output ports. Hence, they will exit stage zero and enter stage one simultaneously. □

LEMMA 4: If there is a message at level $l$ of the OUT queue at an output port, then there will be messages at all levels less than $l$ in the OUT queue.

PROOF: This is proved by considering the movement of messages in the IN queues during the three phases of a cycle as described in Section 3.1. The following points are noted about the messages in the IN queues:

(1) In all phases of a cycle, messages can move to the OUT queue or to the CHUTE.

(2) In phases 1 and 2, messages may move move to a higher level in the IN queue. However, movement of a message to the OUT queue or to the CHUTE is given priority over moving to a higher level. Thus, if a message can move to the OUT queue, it is sent to the OUT queue instead of one level higher in the IN queue.

(3) During phase 3, messages may not move to a higher level. As a result, when messages in the OUT queue and the CHUTE move in phase 1 of the next cycle, an empty slot between two messages in an OUT queue will never be created.

Hence, if $l$ is the highest level in the OUT queue in which there is a message, then there are messages in all levels from 0 through $(l-1)$ of the OUT queue. □

THEOREM 1: All copies of a message enter/exit any stage between 1 and $(n-1)$ of the network simultaneously.

PROOF: We prove the result by induction on the stage number at which copy 0 of a message enters. Consider a message $(X,a)$ entering stage 1 at switches $(1,i)$,$(1,j)$,$(1,k)$ and $(1,l)$. From Lemma 3 we know that the four copies of $(X,a)$ enter stage 1 of the network simultaneously. Suppose $(X,a)^0$ enters stage 1 at switch $(1,i)$ at port p at time $t$. From Lemma 1 we know that the other copies of $(X,a)$ will enter switches $(1,j)$,$(1,k)$ and $(1,l)$ at port p. Since the destination of all the copies is the same, all copies will be routed to the same output port, say, $q$. $(X,a)^0$, $(X,a)^1$, $(X,a)^2$ and $(X,a)^3$ will be placed in the IN queues $p$ at output ports $q$ of switches $(1,i)$,$(1,j)$,$(1,k)$ and $(1,l)$ simultaneously. Since all copies are in IN queues with the same label, they have equal priority in being sent to the OUT queue or to the CHUTE.

From Lemma 4 we know that there are no empty slots between messages in an OUT queue. Hence, one message can be sent to the next stage in every cycle if there is any message at an output port. Suppose at the time when $(X,a)^0$ arrives at $(1,i)$ there are $d$ messages waiting in the IN queues and the OUT queue at output port $q$. Suppose none of these $d$ messages can be combined with each other. (The case when combining is possible is considered separately in Theorem 2). If $(X,a)^0$ does not combine with any of the d messages, then $(X,a)^0$ will experience a delay of $d$ cycles beore it is sent to the next stage. From Lemma 1 and Lemma 2 we know that if there are $d$ messages in the IN queues and the OUT queue at output port $q$ of switch $(1,i)$, then there must be $d$ messages at output port $q$ of switches $(1,j)$,$(1,k)$ and $(1,l)$. Hence, all copies of $(X,a)$ will experience a delay of $d$ cycles before being sent to the next stage.

Thus, in all stages, all copies move synchronously i.e. delays for all copies of a message in any stage will be the same. □

THEOREM 2: If one copy of a message combines with another message then all the other copies of the two messages are guaranteed to combine.

PROOF: It has been proved in Theorem 1 that all copies of a message are placed at level 0 of the IN queues with the same label. It has also been proved that the four copies are placed in the IN queues simultaneously. Initially, when all queues are empty, if one copy of a message in the IN queue moves to the OUT queue, the other copies would also move to the OUT queue. When there are messages waiting in the OUT queue, if one copy of a message moves up one level in the IN queue, the other copies would also move up one level. Hence, all copies are always at the same level of the IN queue or the OUT queue. All copies of a message in the IN queue will be compared to copies of the same message in the OUT queue. Hence, if one copy of a message moves to the CHUTE, all the copies must also move to the CHUTE at the same time. So if one copy of a message combines, then all other copies of the message must combine. □

## 5. CONCLUSION

Clearly, by sending four copies of a message instead of one, we have increased the network traffic. It may seem that this would greatly degrade the performance of the network. However, it should be noted that in the IBM RP3 computer the Fetch-and-Add instructions are routed to the combining network and all others to the non-combining network. Of all the requests generated by a processor, it has been experimentally determined that the percentage of Fetch-and-Add instructions is less than 25% [24]. Hence the network traffic is not very high when a single copy of every message is sent through the network and increasing the traffic by a factor of four is justified.

The design of a fault-tolerant multistage interconnection network based on the Omega network has been presented in this paper. The design of the switching elements constituting the network was described in detail. It supports the Fetch-and-Add instruction of the NYU Ultracomputer and the IBM RP3 computer. It also provides concurrent correction and detection of message transmission errors. The only additional hardware needed is an extra stage. Compared to the Omega network, this amounts to N/4 extra switching elements if there are N processors. Some properties of the proposed network were proven that are also applicable to the Omega network. The proposed design considers message transmission from the processors to the memory modules. It is assumed that a similar network exists for the return path.

## REFERENCES

[1] D. Gajski et al., "Cedar Construction of a Large Scale Multiprocessor," in *Report No. UIUCDCS-R-83-1123, Department of Computer Science, University of Illinois,* Urbana, February 1983.

[2] A. Gottlieb et al., "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.*, vol. C-32, pp. 175-189, February 1983.

[3] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *1985 International Conference on Parallel Processing,* pp. 764-771, 1985.

[4] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694-702, Aug. 1980.

[5] J. H. Patel, "Performance of Processor Memory Interconnections for Multiprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 771-780, Oct. 1981.

[6] H. J. Siegel and S.D. Smith, "Study of Multistage SIMD Interconnection Networks," *Proc. 5th Symp. Comput. Architecture*, pp. 223-229, April 1978.

[7] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, December 1975.

[8] K. E. Batcher, "The Flip Network in STARAN," *Proc. 1976 International Conf. on Parallel Processing*, pp. 65-71, Aug. 1976.

[9] J. E. Lilienkamp, D. H. Lawrie, and P. Yew, "A Fault Tolerent Interconnection Network Using Error Correcting Codes," *Proc. International Conf. on Parallel Processing*, pp. 123-125, 1982.

[10] G. B. Adams, III and H. J. Siegel, "The Extra Stage Cube: A Fault Tolerant Interconnection Network for Supersystems," *IEEE Trans. Comput.*, vol. C-31, May, 1982.

[11] C. L. Wu, T. Y. Feng, and M. C. Lin, "STAR: A Local Network System for Real-Time Management of Imagery Data," *IEEE Trans. Comput.*, vol. C-31, pp. 923-933, Oct. 1982.

[12] R. J. McMillen and H. J. Siegel, "Routing Schemes for Augmented Data Manipulator Network in an MIMD System," *IEEE Trans. Comput.*, vol. C-31, pp. 1202-1214, Dec. 1982.

[13] N-F Tzeng, P-C Yew, and C-Q Zhu, "A Fault Tolerant Scheme For Multistage Interconnection Networks," *12th Computer Architecture Conference*, pp. 368-375, June 1985.

[14] V. P. Kumar and S. M. Reddy, "Design and Analysis of Fault-Tolerant Multistage Interconnection Networks with Low Link Complexity," *Proc. 12th Comp. Arch. Conf.*, pp. 376-386, 1985.

[15] C. S. Raghavendra and A. Varma, "INDRA: A Class of Interconnection Networks with Redundant Paths," *Real Time Systems Symposium* , pp. 153-165, May 1984.

[16] H. S. Stone, "Database Applications of the FETCH-AND-ADD Instruction," *IEEE Trans. Comput.*, vol. C-33, pp. 604-612, July 1984.

[17] K. Padmanabhan and D. H. Lawrie, "A Class of Redundant Path Multistage Interconnection Networks," *IEEE Trans. Comput.*, vol. C-32, pp. 1099-1108, December 1983.

[18] R. K. Iyer and D. J. Rossetti, "Permanent CPU Errors and System Activity: Measurement and Modelling," in *Proc. Real-Time Systems Symp.*. 1983.

[19] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Trans. Comput.*, vol. C-31, pp. 658-671, July 1982.

[20] J. Wakerly, in *Error Detecting Codes, Self-Checking Circuits and Applications*. New York, New York: Elsevier North Holland Inc., 1978.

[21] G. F. Pfister and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *1985 International Conference on Parallel Processing*, pp. 790-797, 1985.

[22] M. Snir and J. Solworth, "The Ultraswitch-A VLSI Network Node for Parallel Processing," *Courant Institute, NYU, NY, Ultracomputer Note 39*, 1982.

[23] A. L. Fisher and H. T. Kung, "Synchronizing Large Systolic Arrays," *Real Time Signal Processing*, vol. 341, pp. 44-52, 1982.

[24] A. Gottlieb, Private Communication.

# ANALYSIS OF A KIND OF FAULT-TOLERANT INTERCONNECTION NETWORK

Lan Jin

Department of Electrical Engineering
The Pennsylvania State University
University Park, PA 16802

Yuanyuan Yang

Department of Computer Science and Technology
Tsinghua University
Beijing, China

Abstract -- With the aim of designing a highly available distributed computer system, a kind of interconnection network with mixed static and dynamic topologies has been proposed and developed. This paper gives a quantitative analysis of the fault-tolerance performance of the network,including the processor connectivity and the worst-case diameter. The method used in the analysis is constructive in nature so that the result not only shows a higher fault-tolerance capability of the proposed interconnection network than that of other existing schemes, but also may serve as the basis of a distributed fault-tolerant routing algorithm with low space and time overheads.

## INTRODUCTION

A distributed computer system may be judged by many different criteria. Reliability and availability are among those important attributes of the system which should be given the most emphasized consideration. With the aim of designing a highly available distributed computer system, a kind of interconnection network with mixed static and dynamic topologies has been proposed and implemented [1] - [2]. This network connects a great number of geographically dispersed processors relying on the principle of static topology, which is characterized by point-to-point links between processor-nodes [3]. So any two processors which are not directly connected by a link need to have their messages relayed by intervening processors. However,in distinction with the ordinary interconnection network with static topology, the proposed network does not use fixed, passive, and dedicated links between processors, but rather establishes these links through a set of explicit switching elements.Therefore, the system can be dynamically reconfigured by properly setting these switching elements to make the communication links active, changable,and sharable among a number of processor pairs. Introduction of dynamic topology into the static interconnection network significantly enhances its performance, especially the fault-tolerance capability.

As summarized in [4], fault-tolerance is an inherent characteristic of the interconnection network. Fault-tolerance can be achieved only if the network can facilitate multiple-path,multiple-pass, and fault-tolerant switching elements. Redundant communication paths are allowed in most static interconnection topologies. Some static interconnection networks can tolerate multiple node/link failures or failures of subnetworks into which the network is partitionable. More attention has been paid to the improvement of fault-tolerance performance of dynamic interconnection topology. Extra-stage, multiple routing-pass, redundant link,error-correcting code, and fault-tolerant switching elements are various schemes which have been proposed. To take advantages of both static and dynamic topologies, it is,therefore, reasonable to combine them in developing a new kind of fault-tolerant interconnection network.

Compared with similar fault-tolerant networks most recently published in the literature[5] - [6], the network analized in this paper differs in many special features, such as low connection-complexity, high fault-tolerance, short diameter, ease of routing, as well as suitability for distributed computer architecture. All these are the general requirements which should be satisfied by a fault-tolerant interconnection network.

In this paper, after a brief description of the proposed interconnection network in the next section, a qualitative analysis of its fault tolerance will be given in the third section. This analysis contains a few typical examples which may help in deriving a quantitative evaluation of the network in the fourth section. The way of analysis adopted is constructive in nature so that the proof of the theorem will automatically imply, as its direct consequence, the basic idea of routing strategy based on which a distributed fault-tolerant routing algorithm with dynamic reconfiguration of the system could be developed. Finally, as an evaluation of the proposed network, a comparison of its fault-tolerance capability with respect to other existing networks will be given.

## DESCRIPTION OF THE MIXED GROUP-SHUFFLE INTERCONNECTION NETWORK

Interconnection network with mixed static and dynamic topologies combines the idea of interconnecting geographically dispersed processors through point-to-point links and the idea of sharing these links among processors through dynamically reconfigurable switching elements. Such a kind of interconnection network may be constructed on the basis of any existing multi-stage dynamic interconnection network by distributing the switching elements to the processors. In this way, the resulted mixed interconnection network becomes more adaptable to matching the distributed computer system architecture. The dynamic interconnection network which has been chosen as the basis of implementation in our work of developing a highly available distributed computer system is the multi-stage shuffle-exchange network [4].

The resultant network developed in this way is called the mixed group-shuffle interconnection network due to the following structural principle. It consists of m stages, each containing $r^m$ processors, where r is an integer denoting the number of processors in each group. Each processor can be labeled by the combination of two numbers C.P , where C is the stage number for $0 \leq C \leq m-1$ with

0190-3918/86/0000/0335 $01.00 © 1986 IEEE

C = 0 corresponding to the leftmost stage, and P is the processor number within the stage for $0 \leq P \leq r^m - 1$ with P = 0 corresponding to the top row. The integer r is also the radix in which the processor number P can be represented as

$$P_{m-1}P_{m-2}\ldots\ldots P_1P_0, \quad 0 \leq P_i \leq r-1,\ 0 \leq i \leq m-1.$$

The network takes r processors with the labels

$$C.P_{m-1}P_{m-2}\ldots P_1X \qquad X = 0, 1, \ldots, r-1$$

in the same stage as a group and establishes connections between the groups in the successive stages according to the group-shuffle interconnection function defined as follows:

$$\text{group-shuffle}(P_{m-1}P_{m-2}\ldots P_1X) = P_{m-2}\ldots P_1XP_{m-1}.$$

That means: the group of r processors labeled $C.P_{m-1}P_{m-2}\ldots P_1X$ in each stage can be conceptually viewed as to be connected to the group of r processors labeled $(C+1).P_{m-2}\ldots P_1XP_{m-1}$ in the succeeding stage, and all the groups of processors in the last stage are connected to the corresponding groups in the first stage. All the connections may be thought of as to be performed by the r$\times$r switches. Thus the whole network forms a closed loop of m stages,each containing $r^{m-1}$ groups with r processors in each group. Therefore the total number of r$\times$r switching elements for interconnecting $mr^m$ processors is $mr^{m-1}$.

Despite the fact that the radix r can be chosen arbitrarily, we have chosen in our design r=4 for simplicity of implementation based on the 2$\times$2 switching elements only. The interconnection of four 2$\times$2 switching elements for performing 4-to-4 crossbar interconnection is shown in Fig.1. Here, using four 2$\times$2 switching elements to replace one 4$\times$4 switching element, we can take the following additional advantages:

·The in-degree (number of incident arcs) and the out-degree (number of outgoing arcs) of each processor are both equal to 2 instead of 4, but every processor can still send messages directly to any one of the four processors in the succeeding stage, and every processor can still receive messages from any one of the four processors in the preceeding stage.

·Each processor can use two switching elements to communicate with the corresponding processors in the neighboring stage. This guarantees a certain degree of redundancy of switching elements for a higher fault tolerance.

The overall structure of a mixed group-shuffled interconnection network with r = 4 is shown in Fig.2. For simplicity, the details of implementation of switching elements have been neglected.

Another property of the proposed interconnection network which is noteworthy is its short diameter, defined as the maximum of the minimum distances between all pairs of processors measured in the number of links. In normal operation, when no failed links, switches or processors exist in the network, the diameter of an m-stage network with $mr^m$ processors is equal to $2m - 1 = O(\log_r N)$, where N is the total number of processors in the network. This can be derived by the following reasoning: every processor can take m steps to send messages to any processor in the same stage, and if any destination processor in the intermediate stage on the way can not be reached within the first m steps, then it must be reached within additional m - 1 steps from the source stage.

## QUALITATIVE ANALYSIS OF FAULT-TOLERANCE

A great number of redundant paths existing between any pair of processors makes the mixed group-shuffle interconnection network highly fault-tolerant. For the evaluation of this performance, we will take the following criteria in our analysis:

·Processor connectivity -- maximum number of faulty processors (with any worst-case distribution) which can be removed without danger of isolating any working processor from the rest of the network.

·Worst-case diameter -- diameter of the network survived after removing the maximum number of tolerable faulty processors as determined above.

·Simplicity of the routing algorithm adaptable to the requirement of fault-tolerance.

By a qualitative analysis, it is easy to determine the upper bound of the processor connectivity from the following straight argument:

At first, no any stage can be tolerant of more than r - 1 faulty processors, because these r or more faulty processors may contain all the successors of a processor in the preceeding stage and thus completely isolate their parent from the network.

Secondly, the processor connectivity can not be better than r - 1 faulty processors in every stage. In other words, if any r - 1 faulty processors are removed from each stage, part of the working processors would be isolated from the survived network, and no communication paths can be guaranteed between any arbitrary pair of processors. A special case suffices to prove this conclusion. Suppose we have the following worst-case distribution of faulty processors. All the r - 1 faulty processors in the second stage are successors of the source node in the first stage, so only one good processor can be reached in the second stage from the source processor. This good processor may, in turn, have all its successors, except one, becoming malfunctioning, thus only one good processor in the third stage can be reached from it. This situation may exist and propagate from stage to stage until, at last, the single reachable good processor in the last stage may have its single good successor just coincident with the source processor. In consequence, all these good processors, one from each stage, may just form a closed loop which is isolated from the remaining processors of the network. This is obviously a situation contradictory to the requirement of the definition of processor connectivity.

The network, however, can survive this worst situation if there is just one less faulty processor in any one stage than the above mentioned case. This gives, therefore, the upper bound of processor connectivity which equals to r - 2 faulty processors in any one stage and r - 1 faulty processors in every other stage. This constitutes the condition of the theorem which will be proved in the next section for a quantitative analysis of the network.

It was stated above that the diameter of the network under normal conditions without node/link failures is equal to 2m - 1, i.e. any destination processor can be reached from any source processor within two passes. This is true even when there exists some limited number of faulty nodes in the network. To find the corresponding fault-tolerant condition we will follow the routing strategy indicated in Table 1. By this routing strategy, a shortest path is established between any two processors $0.P_{m-1}\ldots P_1P_0$ in stage 0 and $k.Q_{m-1}\ldots Q_1Q_0$ in stage k. When the message is traversing along the selected path from stage to stage, the processor number as a base-r code is cyclically shifted left digit by digit, and during each shift the least significant digit of the current code is replaced by a new digit. Thus the whole process of routing can be turned into a procedure of generating a replacing vector R formed by these new digits as denoted by $R_0, R_1, \ldots R_{k-1}, Q_0, Q_{m-1}, \ldots Q_2, Q_1$ in Table 1. For guaranteeing maximum flexibility and fault-tolerance, we leave the selection of the communication path free at the first k steps of traversal and only at the last m steps let it be fixed by the digits $Q_0, Q_{m-1}, \ldots, Q_1$. Thus we are allowed to have some freedom in selecting the digits $R_0, R_1, \ldots R_{k-1}$. The more we have selectable digits $R_i$, $0 \leq i \leq k-1$, the higher fault-tolerance could be realized. Therefore it is the value of k which determines the fault-tolerant condition.

From the architectural point of view, the network analyzed in this paper can be conceptually thought of as a closed multiple-rooted r-nary tree, in which every node may be viewed as a root and, at the same time, may serve as the leaf of other nodes. So the stages 1 and k-1 are the bottlenecks of the equivalent tree structure, and they can tolerate smallest number of faulty nodes. Therefore, the problem can be stated as follows: A path is to be selected from stage 0 to stage k with two passes through an m-stage network. If in stage 1 there exist r-1 faulty nodes, but no one can be the child of the starting node; in stage k-1 there exist r-2 faulty nodes; and in each of the remaining stages (including stages 0 and k) there exist r-1 faulty nodes; then what is the minimum value of k which makes the fault-tolerance realizable? This question will be answered by the Lemma 2 whose proof will be given in the next section.

Here we just give two examples as shown in Figs.3 and 4. The first example in Fig.3 shows a successful case with r = 4, m = 4, k = 3. Among the total $r^k = 64$ codings of $R_0, R_1, R_2$ digits, only 50 codings are needed to mask all the possible faulty nodes (with worst-case distribution), so that there still remain 64 - 50 = 14 codings which can be used to select the free paths containing only good processors. For the second example with r = 4, m = 8, k = 3, as shown in Fig.4, since k is so small that its $r^k = 64$ possible codings are just enough to mask all the possible faulty processor numbers, no one coding remains to allow any valid path to be selected. Therefore the minimum value of k for r=4 and m=8 for two-pass fault-tolerant routing under the specified conditions is equal to 4.

It should be noted that the two-pass fault-tolerant conditions specified here do not hold for any arbitrary pair of source and destination nodes. We will investigate this general case in the next section.

Table 1 Routing Strategy From Node $0.P_{m-1}\ldots P_1P_0$ to Node $k.Q_{m-1}\ldots Q_1Q_0$ With Distance m+k

| Stage | Processor Number | Replacing digit | No.of free proc. | No.of equiv. codes |
|---|---|---|---|---|
| 0 | $P_{m-1}P_{m-2}\cdots P_1P_0$ | -- | -- | -- |
| 1 | $P_{m-2}P_{m-3}\cdots P_1R_0P_{m-1}$ | $R_0$ | $r$ | $r^{k-1}$ |
| 2 | $P_{m-3}P_{m-4}\cdots P_1R_0R_1P_{m-2}$ | $R_1$ | $r^2$ | $r^{k-2}$ |
|  | ......... | ... | ... | ... |
| i | $P_{m-(i+1)}\cdots P_1R_0\cdots R_{i-1}P_{m-i}$ | $R_{i-1}$ | $r^i$ | $r^{k-i}$ |
|  | ......... | ... | ... | ... |
| k-1 | $P_{m-k}\cdots P_1R_0\cdots R_{k-2}P_{m-k+1}$ | $R_{k-2}$ | $r^{k-1}$ | $r$ |
| k | $P_{m-(k+1)}\cdots P_1R_0\cdots R_{k-1}P_{m-k}$ | $R_{k-1}$ | $r^k$ | 1 |
| k+1 | $P_{m-k-2}\cdot\cdot P_1R_0\cdot\cdot R_{k-1}Q_0P_{m-k-1}$ | $Q_0$ | $r^k$ | 1 |
|  | ......... | ... | ... | ... |
| m-1 | $R_0R_1\cdots R_{k-1}Q_0Q_{m-1}\cdots Q_{k+2}P_1$ | $Q_{k+2}$ | $r^k$ | 1 |
| 0 | $R_1R_2\cdots R_{k-1}Q_0Q_{m-1}\cdots Q_{k+1}R_0$ | $Q_{k+1}$ | $r^k$ | 1 |
| 1 | $R_2R_3\cdots R_{k-1}Q_0Q_{m-1}\cdots Q_kR_1$ | $Q_k$ | $r^{k-1}$ | $r$ |
|  | ......... | ... | ... | ... |
| i | $R_{i+1}\cdot\cdot R_{k-1}Q_0Q_{m-1}\cdot\cdot Q_{k-i+1}R_i$ | $Q_{k-i+1}$ | $r^{k-i}$ | $r^i$ |
|  | ......... | ... | ... | ... |
| k-1 | $Q_0Q_{m-1}\cdots Q_2R_{k-1}$ | $Q_2$ | $r$ | $r^{k-1}$ |
| k | $Q_{m-1}Q_{m-2}\cdots Q_1Q_0$ | $Q_1$ | -- | -- |

## QUANTITATIVE ANALYSIS OF FAULT-TOLERANCE

The final purpose of the quantitative analysis is to prove the following theorem. As a preliminary step, three lemmas will be proved first.

Theorem In a m-stage (m = 2) mixed r-nary group-shuffle interconnection network, if there exist no more than r - 2 faulty processors in at least one stage, and no more than r - 1 faulty processors in every other stage, communication can be performed between any pair of the remaining good processors; and the diameter of the survived network is equal to

$$D = \begin{cases} 3m, & m = 2\ ; \\ 4m - 1, & 3 \leq m \leq 5\ ; \\ 3m + \lceil 2\log_r m \rceil + 2\ , & m > 5\ . \end{cases}$$

Lemma 1 Under the condition specified by the Theorem, if some node has r - 2 faulty children, then no more than 2 steps must be made in order to reach a node whose all children are good nodes.

Proof. Starting from a node in the first stage with r - 2 faulty children, the message can be sent through one step to one of the two good nodes in the second stage, and then, from either of these two nodes with at most r - 1 faulty children, the message can be sent further to one of

$$2r - (r - 1) = r + 1$$

nodes in the third stage. Since the fourth stage has at most r - 1 faulty nodes which could be the children of these r + 1 predecessors, at least two of these predecessors can not be followed by any faulty nodes.

For the case of m = 2, the children of the r + 1 good nodes in the third stage (i.e.the first stage) may coincide with each other, thus resulting in fewer effective good nodes in the first stage. But the Lemma 1 still holds since all the r - 2 faulty nodes in the fourth stage (i.e. the second stage) are children of the same source node.

Lemma 2 Under the condition specified by the Theorem, if there are at most r - 2 faulty nodes in stage k - 1, then starting from any node in stage 0 with no faulty children, the message can be sent to any destination node in stage k through m + k steps, where

$$k = \begin{cases} 0 , & m = 2 ; \\ m - 1 , & 3 \leq m \leq 5 ; \\ \lceil 2\log_r m \rceil + 2 , & m > 5 . \end{cases}$$

Proof.

Case 1 m = 2

Starting from any node in stage 0 with no faulty children, in the first step the message can be sent to all r groups in stage 1, so that in the second step all good nodes in stage 2 (i.e. stage 0) can be reached. Therefore, k = 0.

Case 2 m = 3

The routing paths followed for sending a message from the node $0.P_{m-1}P_{m-2}\ldots P_1P_0$ to the node $k.Q_{m-1}Q_{m-2}\ldots Q_1Q_0$ through m + k steps are listed in Table 1. The replacing digits $R_0,R_1,\ldots,R_{k-1}$ of the replacing vector R must be selected in such a manner that all the processors to be passed through are distinguishable from the faulty nodes. This requirement can be expressed in the form of a set of inequalities for all stages 0 through m - 1 and for both passes. From stage i in Table 1 we have

$$P_{m-i-1}..P_1R_0..R_{i-1}P_{m-i} \neq F \quad \text{for the first pass}$$

$$R_{i+1}..R_{k-1}Q_0Q_{m-1}..Q_{k-i+1}R_i \neq F \quad \text{for the second pass}$$

where F denotes the set of all faulty node numbers in stage i. According to the condition specified by the Lemma 2, this set should contain no more than r - 2 elements for i = k - 1 and no more than r - 1 elements for all other i, $0 \leq i \leq m-1$ except i = k-1.

Great flexibility of selecting routing paths is provided by the R vector, but it has only k digits $R_0,R_1,\ldots,R_{k-1}$ which can be used to serve this purpose. The total $r^k$ different codings of these k digits are used in either of the following two ways:

- Part of the codings of the R vector must be avoided or excluded in order to mask all faulty nodes in the network;
- The remaining part of the codings can be used to choose the good nodes to establish the valid communication paths.

The goal of the proof of this Lemma is to determine the minimum value of k with respect to values of m and r such that after subtracting the first part of the codings from $r^k$ the difference still remains positive, i.e. there remains at least one coding of the R vector for setting up a good connection between any source node and any destination node.

From Table 1 it can be seen that since processor numbers in different stages contain different numbers of R digits, the numbers of equivalent R-codings for masking one faulty node in different stages are also different, as indicated in the last column of Table 1.

From the condition of the Lemma, in each of the stages 0 and 1 a maximum of r - 1 faulty nodes may exist and must be excluded from the R-codings only for the second pass. Therefore,with the worst distribution of faulty nodes, r - 1 different codings of $R_0R_1\ldots R_{k-1}$ must be avoided for stage 0; and r - 1 different codings of $XR_1\ldots R_{k-1}$ must be avioded for stage 1, where X = 0,1,...,r-1. This means an equivalent number of r(r-1) codings of $R_0R_1\ldots R_{k-1}$ must be excluded from $r^k$ for stage 1.

For stages 2 through k - 2, faulty nodes may exist in both passes. Since no digits of the R vector appear at the same digit positions for any stage in the two passes, the worst-case distribution is as follows:

- r - 1 faulty nodes exist in each stage i, $2 \leq i \leq \lfloor k/2 \rfloor$, for the first pass, which is equivalent to $(r-1)r^{k-i}$ different codings of the R vector;
- r - 1 faulty nodes exist in each stage i, $\lfloor k/2 \rfloor + 1 = i = k-2$, for the second pass, which is equivalent to $(r-1)r^i$ different codings of the R vector.

Since the digit positions of the R vector in each stage i, $2 \leq i \leq \lfloor k/2 \rfloor$, for the first pass overlap with the corresponding digit positions of the code of the destination node in the same stage for the second pass, the worst-case distribution may be that one of the r - 1 faulty codings for the first pass just coincides with the faulty coding for the second pass (See the example in Fig.4, where coding 20321310 coincides between the two passes in stage 2). This is equivalent to a number of $r^i$ additional codings of the R vector which must be excluded from $r^k$ for each stage i, $2 \leq i . \leq \lfloor k/2 \rfloor$.

Similarly, a number of $r^{k-1}$ additional codings of the R vector must be excluded from $r^k$ for each stage i, $\lfloor k/2 \rfloor + 1 = i = k-2$, in the first pass, because its source code digits may coincide with the corresponding R digits of one of the r - 1 faulty nodes for the second pass.

However, between the first pass and the second pass for each stage i, $2 \leq i \leq k-2$, r - 1 excluded codings of the R vector are duplicated. They should be added back to the $r^k$ total codings.

Similar argument holds for the stage k - 1 with the only difference that $(r-2)r^{k-1}$ equivalent faulty codings of the R vector may exist in the second pass, r additional equivalent faulty codings of the R vector may exist in the first pass, and between them r - 2 codings are duplicated.

For all the remaining stages from stage k through stage m - 1, r - 1 codings of the R vector must be excluded from each first-pass expression.

After all the different faulty codings stated above are subtracted from the total $r^k$ possible codings of the R vector, the difference obtained should still be positive, indicating that there remains some coding available for selecting good nodes along the communication path. This leads to the inequality given on the next page.

In fact, the inequality (1) involves two inequalities, one for k being odd, and the other for k being even. Transformation of the expression (1) yields the two inequalities (2) and (3) which give the implicit functions of k with respect to the values of radix r and the number of stages m of the network.

$$r^k - \sum_{i=2}^{\lfloor k/2 \rfloor} (r - 1)r^{k-i} - \sum_{i=\lfloor k/2 \rfloor+1}^{k-2} (r - 1)r^i - \sum_{i=2}^{\lfloor k/2 \rfloor} r^i$$

$$- \sum_{i=\lfloor k/2 \rfloor+1}^{k-2} r^{k-i} - (r - 2)r^{k-i} - r - (r - 1)$$

$$- (r - 1)r - (r - 1)(m - k) + (r - 1)(k - 3)$$

$$+ (r - 2) > 0 \qquad (1)$$

$$2r^{\lceil k/2 \rceil} \cdot \frac{r - 2}{r - 1} + \frac{2r^2}{r - 1} - (r - 1)(m - 2k + r + 4)$$

$$- 2 > 0 \qquad \text{for } k = \text{odd} ; \qquad (2)$$

$$r^{k/2} \cdot \frac{(r + 1)(r - 2)}{r - 1} + \frac{2r^2}{r - 1}$$

$$- (r-1)(m - 2k + r + 4) - 2 > 0$$

$$\text{for } k = \text{even}. \qquad (3)$$

Numerical solutions to the last two inequalities (2) and (3) are listed in Table 2, which indicates that for practical values of m and r, the value k equals to 3.

Table 2 Values of k as a function of r and m

| k | | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| m | r=4 | 3 | 4--7 | 8--20 | 21--33 | 34--77 |
| | r=8 | 3 | 4--11 | 12--68 | 69--111 | 112-- |
| | r=16 | 3 | 4--19 | 20--90 | 91--225 | 226-- |

For a rough estimation by the order of magnitude, it can be proved that the above inequalities hold true when we take either of the two values of k:

$$k = \lceil 2\log_r m \rceil + 2 \quad \text{or} \quad k = m - 1.$$

Therefore we have the following solution

$$k = \min\{\lceil 2\log_r m \rceil + 2, \; m - 1\}$$

or

$$k = \begin{cases} m - 1 & 3 \leq m \leq 5 \\ \lceil 2\log_r m \rceil + 2 & m > 5 \end{cases} \qquad (4)$$

From the solution we conclude that when k is larger than or equal to the values listed in Table 2 or calculated from (4) there must exist at least one coding of the reconfigurating vector R necessary for establishing a communication path between any source node $0.P_{m-1}...P_1P_0$ and any destination node $k.Q_{m-1}...Q_1Q_0$ through m+k steps.

The proof of the above Lemma also shows that the condition can be relaxed to permit more than r - 2 faulty nodes in stage k - 1, so long as these faulty nodes have their codes with ≤ r - 2 different values at LSD position. This can be seen from Table 1 when we look at the processor number in stage k - 1 for the second pass, which contains the digit $R_{k-1}$ at the least significant digit (LSD) position so that the excluded codings of the R vector can be expressed as $R_0R_1...R_{k-1} = XX...R_{k-1}$, where X stands for any digit from 0 through r - 1.

Lemma 3 Under the condition specified by the Theorem, starting from any node in stage 0 with no faulty children, i steps are enough to send a message to another node in stage i with no faulty children, where $1 \leq i \leq m - 1$.

Proof. Since the source node in stage 0 has no faulty children, the first step will lead to r good nodes in stage 1. Among these r good nodes, at most r - 1 nodes can have faulty children, therefore at least one node is free of faulty children. The same argument can be extended to the following stages.

Now, having proved the above three Lemmas, we are ready to prove the Theorem stated at the beginning of this section. For simplicity of description, we will label the source stage as stage 0. Of course, this stage 0 may be different from the stage 0 which we labeled in the proof of Lemma 2.

Proof of the Theorem.

Case 1. All the children of $0.P_{m-1}...P_1P_0$ are good nodes.

a) The faulty nodes in stage k - 1 have r-1 different values at LSD position.

The condition of the Theorem tells us that there must be at least one stage with no more than r - 2 faulty nodes. Assume this occurs in the stage (k - 1 + i)modm, where $1 \leq i \leq m-1$. According to Lemma 3, a traversal of i steps will lead to a node in stage i with no faulty children, then according to Lemma 2, taking additional m + k steps will lead further to any node in stage (k + i)modm. Thus the maximum traversing distance will be equal to

$$(m - 1) + (m + k) + (m - 1) = 3m + k - 2.$$

b) The faulty nodes in stage k - 1 have ≤ r - 2 different values at LSD position.

According to Lemma 2, traversing through the first m + k steps can lead directly to any node in stage k, so that the maximum distance will be

$$(m + k) + (m - 1) = 2m + k - 1 .$$

Case 2. The node $0.P_{m-1}...P_1P_0$ has faulty children.

a) The number of faulty children of the node $0.P_{m-1}...P_1P_0 \leq r - 2$ .

According to Lemma 1, traversing through the first two steps can lead to a node with no faulty children, then the routing can follow the path as in case 1, so that the maximum distance is

$$(3m + k - 2) + 2 = 3m + k .$$

b) The number of faulty children of the node $0.P_{m-1}...P_1P_0 = r - 1$ .

Under the condition of the Theorem, there must be at least one stage with a node whose faulty children ≤ r - 2. Assume it is the stage i, $1 \leq i \leq m - 1$, to say the nearest stage from stage 0. At first, the route starts from the node $0.P_{m-1}..P_1P_0$ and arrives this stage i through i steps, then, according to Lemma 1, it can reach a node with no faulty children through 2 additional steps.

It should be noticed, however, that if any node in some stage has r - 1 faulty children, then the faulty nodes of the next stage which must be masked will have their codes of the form $F_{m-1}..XF_0$, where X = 0, 1,...,m-1. The LSD can have only one value $F_0$, determined by the MSD of the parent node in the preceeding stage. Similarly, from stage 1 to stage i, each stage will have faulty nodes with only one value at the LSD position.

After the node in stage i + 2 with no faulty children has been reached via the first i + 2 steps from the source node $0.P_{m-1}\ldots P_1P_0$, the route will be continued under the condition of case 1(b). If $k + i < m$, then additional $m - (k + i)$ steps are needed to reach the stage $(m - k + 2)$ so that $[(m - k + 2) + (k - 1)]\text{mod}m = 1$. If $k + i \geq m$,then $[(i + 2) + (k - 1)]\text{mod}m \geq 1$. Both cases will have the (k - 1)th stage whose faulty nodes take no more than r - 2 values at LSD as required by the condition of case 1(b). Therefore, the maximum distance is equal to

$$(m - 1) + 2 + (2m + k - 1) = 3m + k .$$

In summary, communication between any pair of good nodes can be performed under the fault conditions specified by the Theorem, and the diameter of the network may be increased to 3m + k, where k is determined by Lemma 2. After substitution for k, the diameter is

$$D = \begin{cases} 3m , & m = 2 ; \\ 4m - 1 , & 3 \leq m \leq 5 ; \\ 3m + \lceil 2\log_r m \rceil + 2 & m > 5 . \end{cases}$$

## EVALUATION OF THE FAULT-TOLERANCE PERFORMANCE

For a comparative evaluation of the fault-tolerance capabilities of the proposed interconnection network, we take as the reference two similar networks recently published in the literature [5] -- [6]. The result of comparison is listed in Table 3.

Table 3 Comparative Evaluation of The Fault-Tolerance Performance

| Network | Kumar-Reddy(84) | Pradhan(85) | Mixed group-shuffle |
|---|---|---|---|
| Number of processors | $r^m + r^{m-1}$ | $r^m$ | $mr^m$ |
| Degree | 2r | r or r+1 | r(logical) |
| Fault-tolerant Capability | 2r - 1 | r-1 or r | m(r-1)-1 |
| Normal-case Diameter | m | 2m - 1 | 2m - 1 |
| Worst-case Diameter | 3m + 6 | 6m - 3 | $3m+\lceil 2\log_r m\rceil+2$<br>or 3m + 3 |

The number of faulty processors which the three networks can be tolerant of in each stage is roughly the same, but if we refer the processor connectivity to the total number of processors and the processor-degree, then the mixed group-shuffle interconnection network appears to be more advantageous. It can accomodate m times more processors and connects them with smaller physical degree.

The normal-case diameter of the three networks relative to the degree of topology is roughly the same, but if again we refer it to the total number of processors, then the effective diameter of the mixed group-shuffle interconnection network is the shortest.

The worst-case diameter of the mixed group-shuffle interconnection network is also the shortest, especially when its relative value with respect to the normal-case diameter is taken. Its maximum distance of communication under faulty conditions is only 1.5 times longer than that under normal conditions, whilst for the other two networks the faulty condition would increase the diameter in 3 times.

The mixed group-shuffle interconnection network is simple in implementation. The basic idea of the fault-tolerant routing algorithm can be derived on the basis of the Theorem just proved.From Table 1 it can be seen that under normal operation a reconfigurating or replacing vector R should be found and used to replace the code of the source processor digit by digit on the way forward from stage 0 to stage k, and then the remaining digits of the code of the source node as well as the digits of the R vector are gradually replaced by the code of the destination node digit by digit on the way through stages k+1,...,m-1,0,1,... until the destination node in stage k is reached. Thus the combination of the $R_i$ , $0 = i = k-1$,digits and the destination code digits in the form of

$$R_0R_1R_2\ldots R_{k-1}Q_0Q_{m-1}\ldots Q_2Q_1$$

can be used as the routing vector for the purpose of routing control. Since the reconfigurating vector is independent of the destination and transparent to the intermediate processors, it needs to be calculated only once and reserved in the source processor for later use. Under faulty operation, the only work which the source processor should do is to calculate a modified reconfigurating vector and attach it to the destination code digits. In order to accomplish this, each processor must reserve a complete list of faulty nodes of the system. Each time when this list is updated to add or delete any faulty node(s), the reconfigurating vector should be updated. Since the length of each modified reconfigurating vector $< 3m$, the memory space overhead is small. Since the recalculation of the reconfigurating vector needs to be done only when the faulty conditions change, the time overhead is small too.

## CONCLUSION

The basic principle of constructing a multistage interconnection network with mixed static and dynamic topologies is given, and the group-shuffle interconnection function is defined to develop its typical structure. The resulted m-stage ($m \geq 2$) r-nary mixed group-shuffle interconnection network serves the object of analysis of this paper, where r is the number of processors per group.

Three criteria of fault-tolerance performance of the interconnection network are observed. The processor connectivity allows no more than r - 2 faulty processors in at least one stage and no more than r - 1 faulty processors in every other stage. In total, the maximum number of allowable faulty processors with any worst-case distribution is m(r - 1) - 1. The worst-case diameter under the above faulty condition is shown to be

$$3m + \min\left\{m-1,\ \lceil 2\log_r m\rceil + 2\right\}$$

or practically

$$3m + 3 ,$$

which is approximately 1.5 times longer than the diameter of the network under normal conditions.

The proofs of the Lemmas and the Theorem for the processor connectivity and the worst-case diameter gives the basic idea of the fault-tolerant routing control, which helps deriving a simple distributed fault-tolerant routing algorithm for the proposed interconnection network.

Comparative evaluation of the fault-tolerance performance among three similar interconnection networks reveals the advantageous cahracteristics owned by the proposed network.

The proposed mixed interconnection network has been implemented in an experimental distributed computer system THUDS, developed in Tsinghua university, Beijing, China. Though analysis has shown some potential of achieving high system performance, only experience in its application will prove finally its appropriateness and suitability for a variety of distributed processing needs.

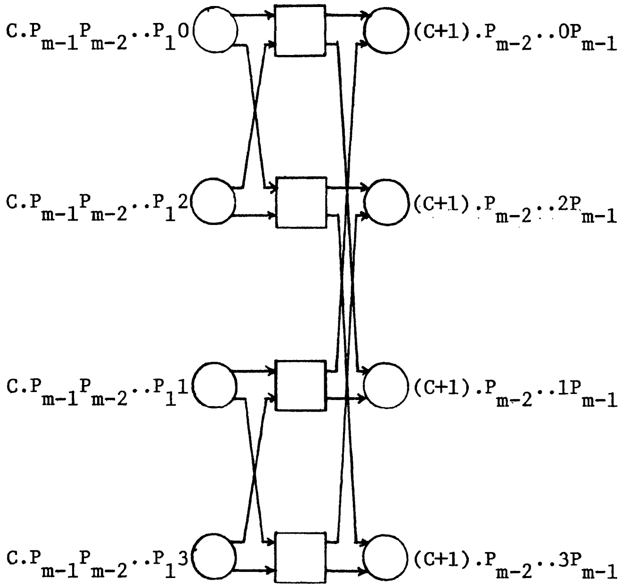

C.P$_{m-1}$P$_{m-2}$..P$_1$0
C.P$_{m-1}$P$_{m-2}$..P$_1$2
C.P$_{m-1}$P$_{m-2}$..P$_1$1
C.P$_{m-1}$P$_{m-2}$..P$_1$3
(C+1).P$_{m-2}$..0P$_{m-1}$
(C+1).P$_{m-2}$..2P$_{m-1}$
(C+1).P$_{m-2}$..1P$_{m-1}$
(C+1).P$_{m-2}$..3P$_{m-1}$

Fig.1. Connection between groups of processors by 2 2 switches

REFERENCES

[1] Lan Jin, et al, "THUDS: A Highly Available Distributed Computer System", Proc.of the 6th IFAC Workshop on Distributed Computer Control Systems (May 1985).

[2] Lan Jin, and Yi Pan, "A Kind of Interconnection Network With Mixed Static and Dynamic Topologies", Proc. of the 6th International Conference on Distributed Computing Systems(May 1985).

[3] Tse-yun Feng, "A Survey of Interconnection Networks", Computer (Dec. 1981),pp.12-27.

[4] Chuan-lin Wu,and Tse-yun Feng, Interconnection Networks For Parallel and Distributed Processing, Tutorial, IEEE Computer Society (1984).

[5] V.P. Kumar, and S.M. Reddy, "A Class of Graphs For Fault-tolerant Processor Interconnection", Proc. of the International Conference on Distributed Computing Systems (May 1984).

[6] D.K. Pradhan, "Fault-Tolerant Multiprocessor Link and Bus Network Architectures", IEEE Transactions on Computers (Jan.1985),pp.33-45.

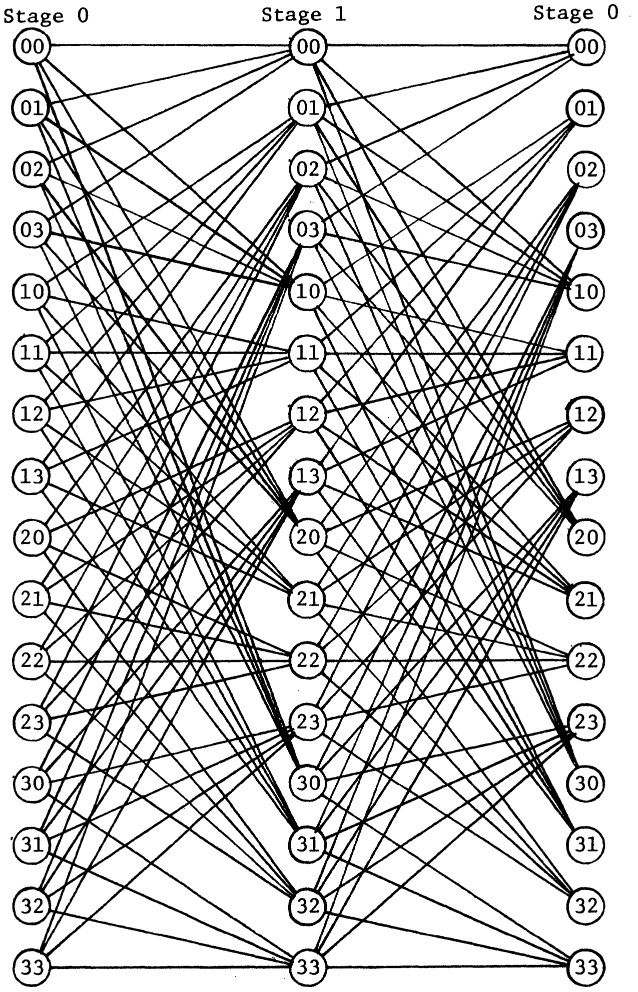
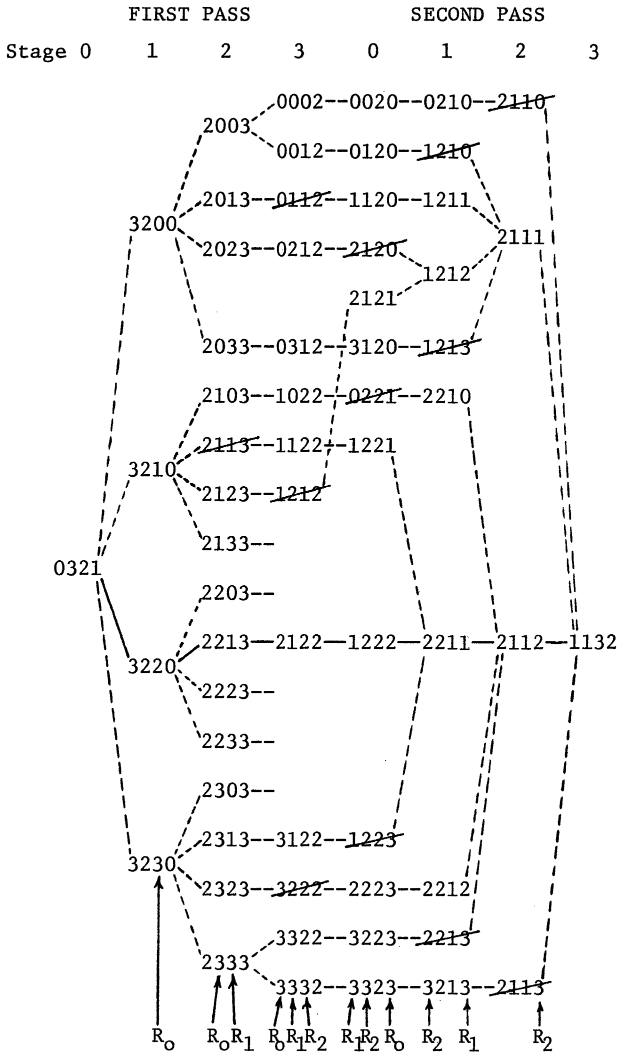


Fig.2. Overall structure of a 2-stage mixed group-shuffle interconnection network (with switches omitted)



| | | | | | |
|---|---|---|---|---|---|
| $R_0R_1R_2\neq$ | 11X<br>Dupli-<br>cated:<br>110<br>113 | 011<br>121<br>322 | 021<br>102<br>312 | X01<br>X31<br>X32 | XX0<br>XX3 |

Fig.3. Routing example 1: r = 4, m = 4, k = 3

| STAGE | FIRST PASS | $R_0R_1R_2\neq$ | SECOND PASS | $R_0R_1R_2\neq$ |
|---|---|---|---|---|
| | | | $R_1R_2$ $R_0$ | |
| | | | ~~11203210~~ | 011 |
| 0 | 10203213 | | ~~21203213~~ | 321 |
| | | | ~~03203212~~ | 203 |
| | $R_0$ | | $R_2$ $R_1$ | |
| | 02032101 | | ~~12032130~~ | X01 |
| 1 | 02032111 | | ~~12032133~~ | X31 |
| | 02032121 | | ~~32032132~~ | X23 |
| | 02032131 | | | |
| | $R_0R_1$ | | $R_2$ | |
| | 20321000 | | ~~20321310~~ | XX0 |
| | ...... | | 20321311 | |
| 2 | ~~20321310~~ | 31X | ~~20321312~~ | XX2 |
| | 20321320 | | 20321313 | |
| | 20321330 | (duplicated: 310,312) | | |
| | $R_0R_1R_2$ | | | |
| | ~~03210032~~ | 003 | | |
| 3 | ~~03210132~~ | 013 | 03213102 | |
| | ~~03210212~~ | 021 | | |
| | $R_0R_1R_2$ | | | |
| | ~~32103320~~ | 033 | | |
| 4 | ~~32110320~~ | 103 | | |
| | ~~32111120~~ | 111 | | |
| | $R_0R_1R_2$ | | | |
| | ~~21113203~~ | 113 | | |
| 5 | ~~21121203~~ | 121 | | |
| | ~~21133203~~ | 133 | | |
| | $R_0R_1R_2$ | | | |
| | ~~12112032~~ | 211 | | |
| 6 | ~~12132032~~ | 213 | | |
| | ~~12212032~~ | 221 | | |
| | $R_0R_1R_2$ | | | |
| | ~~23320321~~ | 233 | | |
| 7 | ~~30320321~~ | 303 | | |
| | ~~33320321~~ | 333 | | |

Fig.4. Routing example 2: r = 4, m = 8, k = 3

# Massively Fault-tolerant Cellular Array

Myoung Sung Lee and Gideon Frieder

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

**Abstract** — *Massively fault-tolerant cellular array* is an array of identical cells with connections only to immediate neighbors, where the cells and the connections may be defective with high probabilities. The cell can function as a processing element, as a memory, or as a switching element that connect to other cells. On the defective array, a large cluster of interconnected working cells is formed and the cells without an adequate number of working neighbors are pruned out from the cluster. The working cells in the cluster are configured into a graph that determines the function of the array. The algorithms for forming the cluster, pruning the cluster, and configuring the cells into a linear array, a two-dimensional array, and a binary tree are described, and simulation results are presented.

## Introduction

With the progress in VLSI technology, we will be able to manufacture huge number of devices on a chip, but we may be unable to avoid many defects. Under current integrated circuit architecture, we can tolerate few defects on a chip. With huge number of devices on a chip, we could afford the redundancy necessary to provide fault-tolerant operations. With enough redundancy, we should be able to devise a fault-tolerant architecture which allows efficient computation in spite of many defects.

Fault-tolerant multiprocessors and configuration of cells on the defective array in the context of wafer-scale integration has been studied by many researchers [6,12,1,14,9,17,7,4,11,15,13]. In fault-tolerant multiprocessor systems, introduction of spare parts and reconfiguration has been used in a limited scale. On the VLSI architecture, where huge numbers of devices are available and connections between them are difficult and defects cannot be eliminated, several different approaches has been tried.

In technology-oriented approaches, discretionary wiring and laser personalization have been used for fault-tolerance on VLSI. In these methods, separate nonreversible processing steps are employed for each chip to reconfigure circuits on the defective integrated circuit. In other approach, programmable switches are provided between cells [17,7], and the switches are used to configure working cells in the defective circuit. Here electrical configuration and reconfiguration is possible, but the switches should be fault-free, and some means to program the switches should be provided. In cellular array approach, cells can be used as switches as well as processing elements [12,1,9,4]. This allows distributed self-configuration, and maintains the general advantages of cellular array which derives from its homogeneous and regular structure.

To take full advantage of reconfigurable defective cellular, we need distributed self-configuration where the global defect pattern of cellular array is not needed. However they were not efficient when many cells were defective. With adequate computing power in each cell, however, distributed self-configuration can be done efficiently. In this paper we describe a cellular array that can be efficiently configured despite many defects. The cellular array, called *massively fault-tolerant cellular array,* is an array of identical cells with connections only to immediate neighbors, where each cell can function as a processing element, as a memory, or as a switching element that connects to other cells. Input and output terminals are connected only at the boundaries of the array. This cellular array anticipates the occurrence of massive defects in the cells and in the interconnections. The cellular array is designed in such a way that there exist a set of working cells that maintains a desired processing capability despite many defective cells and defective interconnections.

To maintain processing capability despite many defective cells and interconnections, cells in the massively fault-tolerant cellular array need to have some mechanism to identify the defective cells and interconnections, and configure themselves around the defective cells. By employing built-in self-testing techniques, we should be able to devise a testing mechanism by introducing additional testing hardware. In this work, we assumed that by some mechanism, each cell knows if its neighbor and the connection to the neighbor is working or defective.

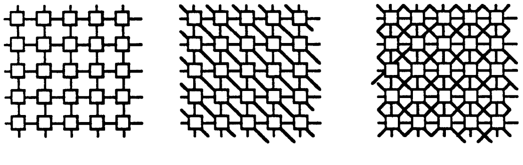
After the defective cells are identified, we form a big cluster of interconnected working cells from a subset of all working cells in the array. The cells without an adequate number of working neighbors are pruned out from the cluster. We then configure the cells in the cluster into a graph that specifies the function of the array.

In the following sections, we describe the architecture of the massively fault-tolerant cellular array, and the algorithms for the formation of the cluster, the pruning of the cluster, and the configuration of the cells into a linear array, a two-dimensional mesh, and a complete binary tree. Simulation of the massively fault-tolerant array and simulation results of the formation of the cluster, pruning, and configuration of cells into a linear array, a two-dimension mesh, and a complete binary tree are shown on arrays of size 40 by 40, 80 by 80, and 120 by 120.

## Massively Fault-tolerant Cellular Array

The three regular interconnection patterns studied in this paper are shown in Figure 1. They are selected to study the effect of the number of the neighbors on the behaviour of the cellular array. The three arrays with the three interconnection patterns of Figure 1 are called *square array, hexagonal array,* and *octal array* respectively. Figure 2 shows a square array with defective cells and defective interconnections. Note that although the initial array is regular, the ensuing array is not (see Figure 2), as the faults cause breakdown in the regularity of the array.

Though there are many defects, there are still many working cells in the array so that useful computation can be per-



(a) square array (b) hexagonal array (c) octal array

**Figure 1.** Interconnection patterns of the cellular array

This work was supported in part by the U.S. Army under the contract no. DAAK20-84-Q-0322.

0190-3918/86/0000/0343 $01.00 © 1986 IEEE

formed. The computations that the array is intended to perform will determine how the working cells are configured. The logical interconnection of cells for any particular computation can be represented by a graph, called the *computation graph* in this paper. The configuration of cells into a computation graph is the process of embedding the computation graph in the defective array. The embedding of the computation graph in the defective array is represented by a graph, called the *connection graph*. For example, the tree in Figure 3(a) is a computation graph; Figure 3(b) shows an embedding of the tree on the defective square array; Figure 3(c) is the connection graph of the embedding of Figure 3(b).

In Figure 3(b), some cells are mapped to the nodes of the computation graph, while other cells are used to connect the cells that are mapped into the nodes of the graph. The cells mapped into the nodes of the graph are called the *computation cells,* and the cells used to connect the computation cells are called the *connection cells.* The connection cells are represented by dots on the connection graph.

A cell in the massively fault-tolerant cellular array is configured into a computation graph by identifying the logical connections of the cell. For example, a cell is configured into a binary tree by saving in special registers the directions of the neighbors as a father, a left child, and a right child of the cell. The configuration can be changed by changing the contents of the registers. The configuration is performed in a distributed fashion: each cell makes a decision only with the information that is available at the cell.

## Architecture of a Cell

Each cell has a processor, local memory, Communication Registers, an ID Register, Neighbor Status Registers, a Status Register, a Pattern Register, and Connection Registers. Figure 4 shows the architecture of a cell in the square array. The registers are explained below:

**ID Register:** Stores the row and column indices of the cell in the array.

**Neighbor Status Register, NS[0..N-1]:** Stores the status of the neighbors. The neighbor can be defective, working, or pruned out. Here N is the number of immediate neighbors.

**Status Register, SR:** Stores the current status of the cell. The states of a cell are "idle", "live", "pruned", "comp", "conn", and "mem". The status of all working cells are initially "idle". The clustering procedure changes the status of the cells belonging to a big interconnected cluster to "live", and the pruning procedure changes the status of the working cells without an adequate number of cells to "prune", so that they are not used in the configuration procedure. The configuration procedure changes the status of the "live" cells to "comp" or "conn", depending upon whether the cell is used as a computation cell or a connection cell. Finally, the status of a cell is "mem" when the cell is used as a memory.

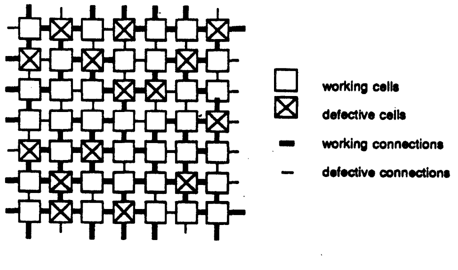
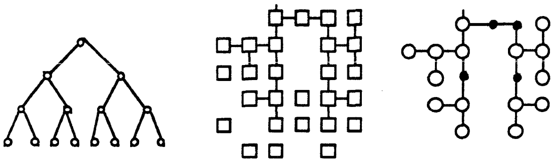


**Figure 2.** A defective square cellular array



(a) computation graph (b) embedding of a tree on a defective array (c) connection graph

**Figure 3.** A computation graph, its embedding and a connection graph

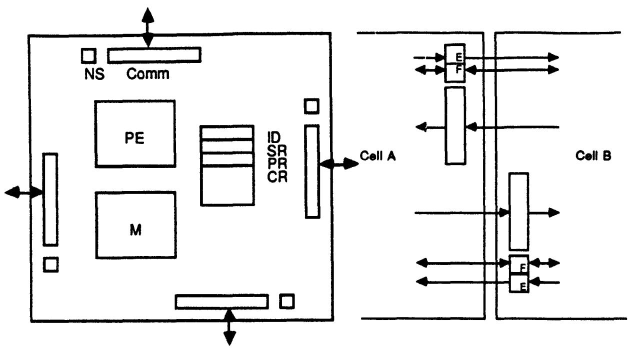


**Figure 4.** Architecture of a cell and communication register of the cellular array

**Pattern Register, PR:** Stores the current configuration pattern. The configuration patterns that are recognized are linear array, two-dimensional mesh, complete binary tree, and spanning tree.

**Connection Register, CR[0..N-1]:** Stores the directions of neighbors in the logical configuration specified by the Pattern Register, PR. The meanings of the CR[0..N-1] are different for each configuration. When PR specifies that the configuration is a linear array, CR[0] is the direction of the predecessor, and CR[1] is the direction of the successor. When PR specifies a binary tree, CR[0..2] are the directions of the father, the left child, the right child, and CR[3] is the level of the cell. When PR specifies a two-dimensional mesh, CR[0..3] are the directions of the up, right, down, left neighbors of the cell, and CR[4..5] are the row and column indices of the cell in the two-dimensional mesh. When PR specifies a graph, CR[0..3] are the directions of the two sources and two destinations. Finally when PR specifies a spanning tree, CR[0] is the direction of the father, and CR[1..N-1] are the directions of the sons.

**Communication Registers, Comm[0..N-1]:** These are used for communication between the cells. Cells communicate with other cells by sending and receiving messages through Communication Registers. Each Communication Register provides a one-way communication between two cells. Between a boundary of two cells, two Communication Registers provide two-way communication. Figure 4 shows a pair of Communication Registers. The arrows on the Figure 4 shows the directions of the information. The fields of the Communication Register are as follows.

| | |
|---|---|
| **Full Bit, FB:** | Shows if the Communication Register is full or empty. |
| **Enable Bit, EB:** | Shows if the other cell is willing to receive the message. If EB is 0, the other cell may not read the Communication Register. EB is used to prevent deadlock. |
| **Messages, Mesg:** | Contains the messages. |

The FB is used to synchronize the communication between two cells. A cell can read a Communication Register when the Communication Register is full, and after the cell reads the Communication Register, FB is reset to 0. If a cell wants to read a message from an empty Communication Register, the cell waits until FB is set again. A cell can write to an empty Communication Register, and if a cell initiates write to a full Communication Register, the cell has to wait until EB is set. This provides the synchronization between two cells.

**Processor:** The processor is an instruction set processor with small instruction set. The instructions includes usual arithmetic, logical, data transfer, program control operations, and send and receive for input and output. "Send (dir, message)" writes the "message" on the Communication Register in "dir" direction: return from "Send" acknowledges that the message is written to the Communication Register. "Recv (dir)" reads a message either from the Communication Register in "dir" direction, or from any full Communication Register if "dir" is "any": return from "Recv" acknowledges that a message is available at the Communication Register in "dir" direction.

**Controller:** In the massively fault-tolerant cellular array, an external machine is attached to the boundary cells to control the operation of the array. The external machine initiates the operations of the array, and provides the data to the array by sending messages to the array. The external machine will be called a *Controller*.

When the massively fault-tolerant cellular array is powered on, Neighbor Status Registers, NS[0..N-1], of each cell are set to indicate the status of the neighbors of each cell (working or defective) by some testing mechanism. The hardware and algorithms for the testing have not been devised yet. We assumed that the testing can be done by some mechanism.

After the testing is finished, the controller initiates the clustering procedure which identifies the largest cluster of interconnected working cells in the array. After all the cells in the cluster are given identification numbers, the controller initiates the pruning procedure which prunes out the cells in the cluster without an adequate number of working neighbors. The controller then initiates the configuration procedure to configure the cells in the cluster into a desired computation graph. When the Controller sends a message to a cell at the boundary of the array, the cell relays the message to its neighbor cell, and the message will propagate through the working cells. The cells will be configured into a graph specified by the operation field of the message. The configuration of the cells is finished when all the cells set their Configuration Registers correctly, and the boundary cell connected to the controller returns an appropriate message to the controller.

## Formation of the Cluster of Cells

Since the cells in the cellular array connect only to the immediate neighbors, when a cell wants to communicate with another cell which is not directly connected to it, the message should be relayed by intervening working cells to the target cell. Therefore, a small cluster of working cells surrounded by defective cells cannot be used. The array can be useful only when a large cluster of of interconnected working cells is formed in the array.

We can use *percolation theory* [16,3] to predict if a large cluster appears on the array, and if the cluster appears, to predict the size of the cluster. According to the percolation theory there exist a *critical probability* such that when the probability that a cell is working is more than the critical probability, there appears an infinite cluster of interconnected working cells in the defective infinite array of cells.

## Percolation Theory and the Cellular Array

Consider a lattice $L$ defined as a graph of $N$ sites (or vertices) and $M$ bonds (or lines). In most cases of practical interest, $L$ will be a regular two or three dimensional lattice of finite or infinite extent. In the *bond problem*, each bond of $L$ is *occupied* (or *open*, or *working*, etc.) with probability $p$ or *vacant* ( or *blocked*, or *defective*, etc.) with probability $1-p$. Occupied bonds are *connected* if they meet at a common site, and a connected set of $s$ bonds form a *bond cluster* of size $s$. In the *site problem*, each site is occupied with probability $p$ and vacant with probability $1-p$. Occupied sites are connected to form *site clusters* if they are adjacent through the bonds of $L$.

For an infinite lattice $L$ there is a critical probability $p_c = p_c(b, L)$ or $p_c(s, L)$ for the bond or site problems such that for $p < p_c$ all clusters will be finite while for $p > p_c$ there will, with positive probability, be an infinite cluster in $L$. The infinite cluster is called a *percolation cluster*. We can define the *percolation probability*, $P(p)$, as the probability that a site, chosen at random, belongs to an infinite cluster. One defines the *critical probability*, $p_c$, as

$$p_c = \sup\{p|P(p) = 0\}.$$

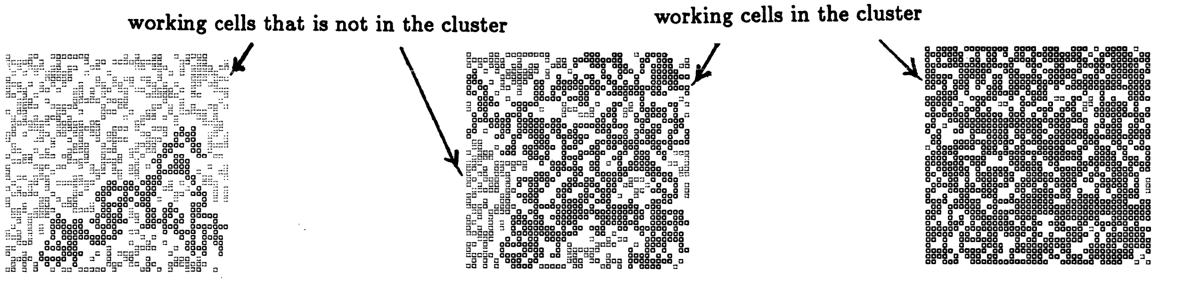
The sites in the percolation model corresponds to the cells of the massively fault-tolerant cellular array, and the bonds in the percolation model corresponds to the connections between cells. In the site percolation problem, all the bonds are assumed to be occupied and only the sites can be vacant; this corresponds to the assumption that all connections are working and only cells can be defective. In the bond percolation problem, all the sites are assumed to be occupied and only the bonds can be vacant; this corresponds to the assumption that all cells are working and only connections can be defective.

Since the area of a cell is much bigger than the area of a connection in the cellular array, and the defect probability of an integrated circuit is at least proportional to the area of the integrated circuit, the defect probability of a cell is much greater than that of a connection. Therefore, as a first approximation, the array can be modeled by the site percolation model. On the site percolation model of the defective array, we can take into account that connections can be defective by associating connections with neighboring cells. When a connection is defective, the cells associated with the defective connection are considered defective. When we need to use the working cells connected to the defective connections, the defective array should be modeled by the combination of site and bond percolation problem [8].

In percolation theory the *percolation cluster* is an infinite cluster that appears on the infinite lattice. On the cellular array of finite size, we define the *percolation cluster* as the largest cluster that is connected to all four borders of the array when the array is rectangular. Using the percolation theory, we can predict that the percolation cluster will appear in the defective cellular array when the probability that a cell is working, $p$, is more than the critical probability, $p_c$ of percolation theory.

Figure 5 shows the formation of the clusters in a square array, where the critical probability is 0.59. Here cells in solid boxes belong to the largest cluster. When $p$ is 0.5, the percolation cluster does not appear (Figure 5(a)); when $p$ is 0.61, a thin percolation cluster appears (Figure 5(b)); when $p$ is 0.75, a thick percolation cluster appears (Figure 5(c)).

However, as we can see in Figure 5(b), even though the percolation cluster appears in the array, when $p$ is not high enough, the percolation cluster is "thin", contains many branches, and does not include many boundary cells. The cells on the thin


working cells that is not in the cluster
working cells in the cluster

(a) p = 0.5; a percolation cluster does not appear.

(b) p = 0.61; a thin percolation cluster appears.

(c) p = 0.75; a thick percolation cluster appears.

**Figure 5.** The formation of clusters on the square array

percolation cluster may not be used effectively because communication between two cells in the thin cluster is difficult, and configuration of cells into a computation graph is not efficient. Furthermore, input and output on the thin percolation cluster is difficult due to the lack of the boundary cells. Therefore the thin percolation cluster in Figure 5(b) may not be useful. When $p$ is high enough, the percolation cluster that appears in the array is "thick", and has many boundary cells. The cells in the thick cluster may be used effectively for configuration and computation. The cluster in this case is shown in Figure 5(c).

The usefulness of the percolation cluster depends on the computation graph that will be embedded on the percolation cluster. For example, a percolation cluster may be considered adequate enough to embed a linear array but the same percolation cluster may not be adequate to embed a two-dimensional array.

## Formation of the Cluster

When $p$ is more than $p_c$, we can form a percolation cluster of working cells in the defective array. The cluster is formed by connecting the working cells into a spanning tree which spans all the working cells connected to a certain boundary cell. The controller sends a message to a working cell at the boundary, where the message specifies that a spanning tree of working cells be formed. The cell that received the message from the controller becomes the root of the spanning tree.

After a spanning tree is formed with the cell that received a message from the controller as the root of the spanning tree, the cell returns the number of cells connected into a spanning tree to the controller. If the number is large enough, the cells in the spanning tree are taken as the percolation cluster. If the number is not large enough, another message is sent to some other working cell on the boundary, and a new spanning tree is formed with the new cell as the root of the spanning tree. This process continues until a percolation cluster is found, or the controller gives up finding a percolation cluster in the defective array.

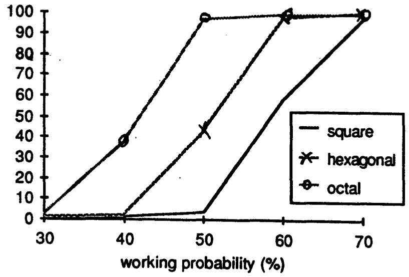


**Figure 6.** Percentage of working cells that are in the largest cluster

When a cell receives a message from a neighbor, the cell changes its state by setting the Status Register, SR to "live", and it saves the direction of the neighbor on CR[0] as the father of the cell. The cell then sends the message to its working neighbors. If a neighbor returns the message that the neighbor is a part of the spanning tree, the direction of the neighbor is saved on CR[1..N-1] as a son. The spanning tree grows in depth first order. The enabling and disabling of communication registers is necessary to prevent deadlock.

Figure 6 show the percentage of working cells that are in the largest cluster in the 120 by 120 square, hexagonal, and octal array, respectively. The experimental results are generally in agreement with percolation theory. From Figure 6 we can see that more than 90% of working cells belong to the percolation cluster when $p$ is more than 0.7 on the square array, when $p$ is more than 0.6 on the hexagonal array, and when $p$ is more than 0.5 on the octal array.

## Assignment of Id

After the working cells are connected into a spanning tree, the cells on the spanning tree are assigned identification numbers. The identification number of a cell is the row and column indices of the cell in the array. The identification number is saved in the ID register.

The controller sends a message to the root cell of the spanning tree, where the first field of the message specifies the operation, and next fields are the row and column indices of the root cell. When a cell receives the message, the cell saves the row and column indices on the Id register, and computes the Id of the son. Then the cell sends the message with the computed Id to the son. When the cell receives the message from the son, it iterates the same operations on the next son.

## Pruning

After a percolation cluster of working cells is formed in the defective array, the cells in the percolation cluster can be configured into a computation graph. However, some cells in the percolation cluster form a single-width dead-end branch, and they may not be configured effectively. Furthermore, they may slow down communications between cells. To facilitate the configuration of working cells and the communication between cells, we may prune out the dead-end branch of cells from the percolation cluster. Figure 7 shows an example of such a pruning.

Pruning of the dead-end branches from the cluster can be generalized to ***pruning-to-k***. The Pruning-to-k operation prunes out the cells which are connected to less than or equal to k working neighbors. Here $k$ is called the ***level*** of the pruning. Pruning of the dead-end branch corresponds to pruning-to-1:

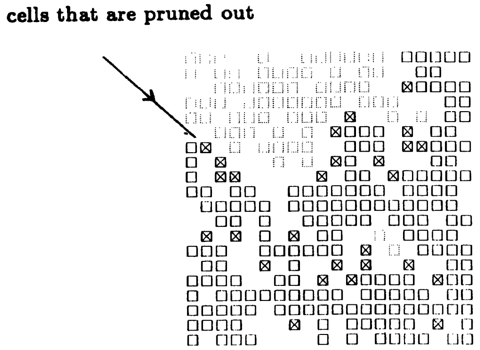


**Figure 7.** Pruning of dead-end branches from a cluster

the cells connected to only one working neighbor are pruned out from the cluster. Pruning is applied repetitively until no more cells are pruned.

By pruning the cells from the cluster, we can have a cluster of tightly connected cells. After the pruning-to-k operation, all the cells in the cluster are connected to at least k+1 working neighbors. This can facilitate the configuration of cells into a graph and the communication among the cells in the cluster.

The Controller initiates the pruning operation by sending a message to a cell in the percolation cluster. The message consists of a field specifying the pruning operation and the level of the pruning, $k$. When a cell receives the message from a neighbor, the cell counts the number of working neighbors, and compares the number of working neighbors with the pruning level $k$ specified on the message. If the number of the neighbor is not greater than the pruning level, the cell prunes itself from the cluster by changing its Status Register, SR, to "pruned" from 'live'. Then the cell sends messages to its working neighbors that it has been pruned out. The neighbors set their Neighbor Status Register accordingly. The cell which received the message from the Controller returns the message containing the number of pruned cells to the Controller. The controller sends the pruning message again until no more cells are pruned out from the cluster.

Figure 8 shows the pruning of cells from the percolation cluster on the square, hexagonal, and octal array of size 40 by 40, 80 by 80, and 120 by 120. The pruning level $k$ was restricted to less than half the number of neighbors. When the pruning level is more than half the number of neighbors on the network, most of the cells are pruned out leaving only a small isolated cluster of cells.

From the experiments, we can see that when the working probability is high enough, most of the cells in the percolation cluster are connected to several working neighbors. When the working probability is 80%, on the square array, more than 95% of the cells have two or more working neighbors, on the hexagonal array, more than 95% of the cells in the cluster have three or more working neighbors, and on the octal array, more than 95% of the cells in the cluster have four or more working neighbors.

## Configuration of Cells

The cells have to be configured into a general computation graph which specifies the function of the array. Before the configuration of the cells into a general computation graph, we studied the configurations into three particular graphs: linear array, complete binary tree, and two-dimensional array (mesh). Many computations can be done efficiently on these graphs.

The configuration of cells into a computation graph is the process of embedding the computation graph on the defective array. Since the cells that does not belong to the percolation cluster cannot be used, computation graph is embedded on the percolation cluster. Note that when the percolation cluster appears on the defective array, most of the working cells belong to the percolation cluster.

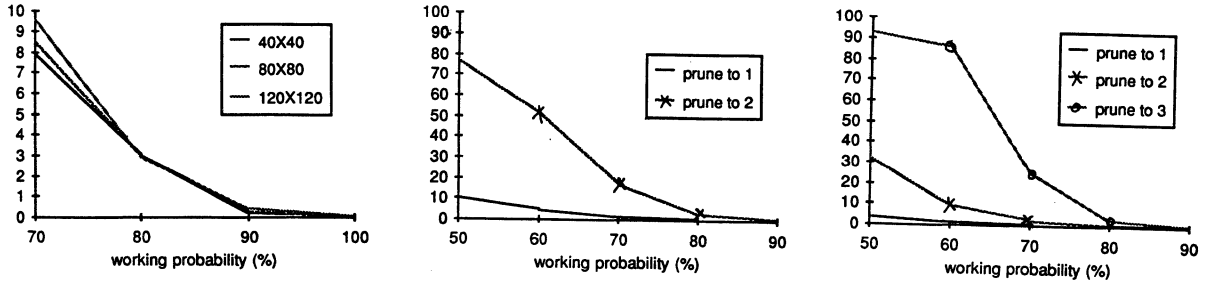
The efficiency of the configuration into a graph $G$, $e_G$, is defined as

$$e_G = \frac{\text{number of cells used as computation cells}}{\text{number of working cells in the cluster}} \times 100$$

The delay $d_G(C_1, C_2)$ between the two cells, $C_1$ and $C_2$, in a configuration into a graph $G$ is defined as one plus the number of connection cells between the two cells $C_1$, and $C_2$. Therefore, the delay between two directly connected cells is 1, and the delay is 2 when there is one connection cell between two cells. The *maximum delay* of the configuration is the maximum delay among all two adjacent computation cells, and the *average delay* of the configuration is the average of all delays among two adjacent computation cells.

We define the *degree* of a graph $G$, $d_G$, as the average degree of the vertices of the graph. To configure the cells in the defective array into a computation graph of degree $d_G$ efficiently, the number of neighbors on the array, or the degree of the array, $d_A$, needs to be greater than $d_G$. When $d_A$ is less than $d_G$, the efficiency of the configuration becomes low. We can expect that cells can be configured into a linear array ($d_G$ = 2), and a tree ($d_G$ = 3) efficiently on the square array ($d_A$ = 4), the hexagonal array ($d_A$ = 6), and octal array ($d_A$ = 8). But the configuration of cells into a mesh ($d_G$ = 4) on the defective square array may not be as efficient as the configuration on the hexagonal array or on the octal array.

On the following sections overview of configuration procedures into a linear array, a tree, and a mesh are described. The detailed algorithms can be found in [10].



(a) square array

(b) hexagonal array

(c) octal array

**Figure 8.** Percentage of cells in the largest cluster that are pruned out

## Linear Array

In the linear array, every cell has two neighbors: the predecessor, and the successor. The configuration of cells into a linear array is the process of identifying predecessors and successors and saving the directions of the predecessors and successors in the Connection Registers, CR[0], and CR[1].

The configuration procedure consists of three parts: Linear, Extend, and Join. Procedure Linear grows the linear array into the defective array of cells. When the linear array is grown in the defective array by the Procedure Linear, Procedure Extend finds the cells which are not in the linear array, but which can be connected into the linear array. Then Procedure Join connect the cells identified by Procedure Extend into the linear array. By combining the three procedures, Linear, Extend, and Join, most of the cells in the cluster are connected into the linear array.

The controller initiates the configuration by sending a messages to a cell at the boundary of the array. The message consists of the fields specifying the operation, the number of cells to be connected into the linear array, the direction of the successor neighbor. When a cell $B$ receives a message from a neighboring cell $A$, the cell $B$ sets the Pattern Register PR to "linear array", saves the direction of the cell $A$ on CR[0] as a predecessor. Then the cell $B$ tries to grow the array by adding a neighbor as its successor. First, if the neighbor $C$ specified as the successor on the message is working, the message is sent to $C$. If $C$ is connected to the linear array, the linear array grows from $C$ again. The cell $C$ returns the message to $B$ with the number of cells on the linear array after $C$. Then cell $B$ saves the direction of the cell $C$ on CR[1] as a successor, and increases the number of cells by one, and returns the message to the predecessor, $A$. If $C$ fails to be connected into the linear array, then the neighbor on the direction of the growth of the linear array as specified on the message is tried. If this fails too, then any working neighbor is tried. If all fail, the linear array retracts to the cell $B$, and growth of linear array is tried at cell $B$ again. Since the cells do not know the global state of the network, the linear array can be grown into the dead-end, and the cells may have to backtrack often.

When Procedure Linear is finished, the cell at the boundary which received the message from the Controller returns the number of cells connected into the linear array to the controller. If the number of cells is less than the number the Controller wants, the controller sends a new message to the cell. The new message consists of the fields specifying Procedure Extend, and the number of cells to be joined to the linear array. The cells which were not part of the linear array but adjacent to the linear array are identified and joined into the linear array.

With the three configuration procedures, Linear, Extend, and Join, most of the cells are configured into a linear array when the working probability of a cell is adequate. Figure 9 shows the cells connected into a linear array on the square array. Figure 10 shows the efficiency of the configuration of linear array on the square array, on the hexagonal array, and on the octal array of size 120 by 120.

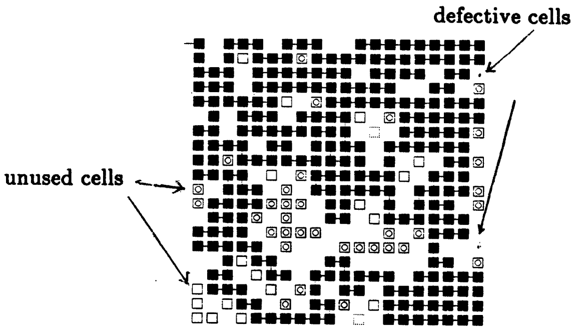


Figure 9. Configuration of cells into a linear array in the defective square array, p = 0.8

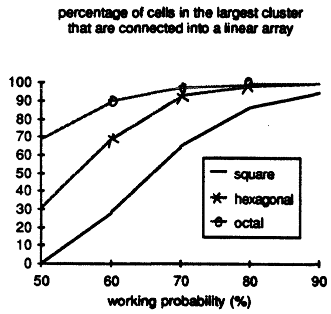


Figure 10. Efficiency of linear array configuration

From the Figure 10, we can see that most working cells are connected into a linear array. Since degree of the linear array is two, the configuration of cells into a linear array should be efficient on all arrays even when working probability of a cell is not high. As the coordination number of the array increases, efficiency of the configuration increases rapidly. On the square array, when the working probability is 80%, more than 85% of the working cells are connected into the linear array. On the octal array, with the working probability 60%, about 90% of the working cells are connected into the linear array.

Since all computation cells are connected directly to the other computation cells without intervening connection cells, no delay has been introduced, and the average delay is 1.

## Tree

In the complete binary tree, every cells have three neighbors: a father, a left child, and a right child. The working cells of the defective array are configured into a complete binary tree by setting their Connection Registers CR[0..2] to the directions of a father, a left child and a right child of each cell respectively, and saving the level of the cell in the tree on CR[3]. (The level of the leaf node is defined as one, and the level of a father is one more than that of its child.) Some working cells are used as the nodes of the tree, which are *computation cells,* and some are used as *connection cells,* which are used to connect computation cells.

Since the topology of the binary tree and that of the array of cells does not match, we need to use many working cells as connection cells even when there is no defects in the network. Koren [9] studied embedding of a tree in a defective square array, but his procedure allows few defects, and its efficiency is very low when there are many defects. The algorithm sets all working cells in the row and the column of the defective cell as connection cells, thereby making a reduced array without defect of one less row and one less column for each defects. Note that when there are many defects, none of the rows and columns will be without defective cells.

The algorithm we devised allows efficient configuration even when many cells are defective. The algorithm has two parts: Tree, and Retract. Procedure Tree connects the cell into a tree, and Procedure Retract retracts the subtree when it can not increase the level of a subtree

The Controller initiates the configuration into a tree by sending a message to a cell at the boundary of the array. The message consists of a field specifying the operation, and the level of the tree desired. If the cells are successfully configured

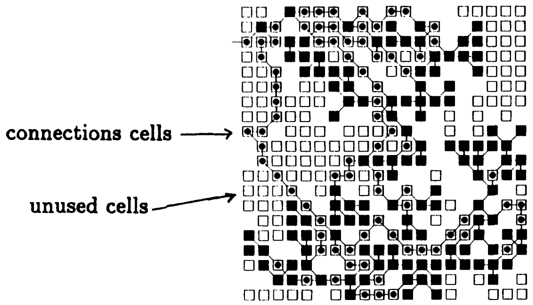


**Figure 11.** Configuration of cells into a tree in the defective octal array, p = 0.7

into a tree of the level specified on the message, the cell returns the level of the tree. The Controller then increases the level of the tree by one, and sends the message again to the cell until the desired level is achieved.

When a cell receives the message from the father cell, the cell tries to increase the level of the left subtree by one. If it is successful, the cell tries to increase the level of the right subtree by one. If it is successful, the level of the tree has been increased by one, and the cell returns the message to its father. But if it fails, the cell is changed into a connection cell, the right subtree is retracted, and the tree expansion is tried at the left subtree again. If the level of the left subtree cannot be increased, the left subtree is retracted, and the cell is changed to the connection cell, and the tree expansion is tried at the right subtree again. The tree is expanded in breadth-first order. Figure 11 shows the trees embedded in an octal array.

Since the average degree of the tree graph is three, configuration of cells into the tree in the square, hexagonal, and octal array could be efficient even when working probability of a cell is not high. Table 1 shows the maximum level of the tree into which cells are configured. Note that to increase the level of a tree by one, the number of computation cells in the tree should be multiplied by two. Table 2 shows the efficiency of the configuration, and Table 3 shows the average delay of the configuration. As the number of neighbors in the array increases, and as the working probability increases, efficiency of the configuration increases, and average delay decreases.

We compared the efficiency of our configuration algorithm with the embedding of H-tree in a defectless square lattice. H-tree is a complete binary tree embedded in a recursive pattern that looks like the letter 'H' [2]. H-tree is known to be the most efficient way of embedding a complete binary tree in a square lattice. The maximum level of H-tree that can embedded in a defectless square lattice is 9 in a 40 by 40 lattice, and 11 in a 80 by 80 lattice, and the efficiency of embedding the largest H-tree on lattice of size 40 by 40, or 80 by 80 is 32%, and the delay on the H-tree is about 3.4 [10]. Comparing these numbers with Table 2 and Table 3, we see that we can configure cells into a defective array without much penalty even when there are many defective cells in the array.

## Mesh

On the mesh of cells, every cells have four neighbors: left, right, up, down. The working cell on the defective array are configured into a mesh by setting its Connection Registers CR[0..3] to the directions of the neighbors connected as up, right, down, and left neighbor of the cell.

Manning [12] describes the algorithm for embedding a mesh on the defective square array, and Green [5] describes embedding a mesh using the channel between the cells. Both uses the knowledge of the global state of the defective array. Here a distributed algorithm where each cell knows only the state of the neighbors (working or defective) is described.

| size | 40X40 | | | 80X80 | | |
|---|---|---|---|---|---|---|
| working | degree | | | degree | | |
| probability | 4 | 6 | 8 | 4 | 6 | 8 |
| 50% | | 5 | 7 | | 6 | 8 |
| 60% | 6 | 7 | 8 | 6 | 8 | 9 |
| 70% | 7 | 8 | 8 | 9 | 9 | 10 |
| 80% | 8 | 8 | 8 | 9 | 9 | 10 |
| 90% | 8 | 8 | 9 | 10 | 10 | 11 |
| 100% | 8 | 8 | 9 | 10 | 10 | 11 |

**Table 1.** Levels of the embeded trees

| size | 40X40 | | | 80X80 | | |
|---|---|---|---|---|---|---|
| working | degree | | | degree | | |
| probability | 4 | 6 | 8 | 4 | 6 | 8 |
| 50% | | 7.0 | 12.4 | | 5.0 | 8.1 |
| 60% | 9.2 | 14.6 | 20.3 | 3.1 | 6.8 | 13.4 |
| 70% | 11.7 | 17.1 | 22.7 | 8.7 | 11.4 | 17.0 |
| 80% | 15.0 | 19.9 | 19.9 | 10.0 | 10.0 | 20.0 |
| 90% | 17.7 | 17.7 | 35.5 | 17.8 | 17.7 | 26.6 |
| 100% | 15.9 | 15.9 | 31.9 | 16.0 | 16.0 | 32.0 |

**Table 2.** Efficiencies of the tree configurations

| size | 40X40 | | | 80X80 | | |
|---|---|---|---|---|---|---|
| working | degree | | | degree | | |
| probability | 4 | 6 | 8 | 4 | 6 | 8 |
| 50% | | 2.13 | 1.95 | | 2.54 | 2.17 |
| 60% | 2.79 | 2.25 | 1.77 | 2.91 | 2.43 | 2.07 |
| 70% | 3.03 | 2.06 | 1.88 | 2.86 | 2.22 | 1.90 |
| 80% | 2.16 | 1.88 | 1.72 | 2.45 | 1.93 | 1.79 |
| 90% | 2.19 | 1.75 | 1.69 | 2.49 | 2.02 | 1.75 |
| 100% | 2.00 | 1.60 | 1.63 | 2.24 | 1.85 | 1.75 |

**Table 3.** Delays on the embedded trees

The Controller sends a message to the cell at a boundary of the massively fault-tolerant cellular array, where the message tells the cell to grow a horizontal line of the mesh. If it is successful, the Controller sends a message to the cell at the other boundary to grow a vertical line of the mesh. Growth of horizontal and vertical line alternates until no more line of the mesh can be grown on the array. The cells at the junction of a horizontal line and a vertical line becomes the nodes of the mesh. The cells at the nodes of the mesh are computation cells, and the cells connecting the nodes are connection cells. The computation cells are given the coordinates of the mesh.

Since the complexity of the mesh is four, and the degree of the defective square array is less than four, efficiency of the configuration may be low on square array. When a growing horizontal line comes across a defective cell, the line should veer around the cell, and this uses the cells which can be used for a vertical line. Veering around the defective cell on the square array while growing a horizontal line blocks the growth of a vertical line, and vice versa. Therefore, bending the line should be done sparingly. On the hexagonal and octal array, the growing horizontal line can use the connection without occupying the cell in the other direction. This increases the efficiency of configuring the cells into a mesh on hexagonal and octal array.

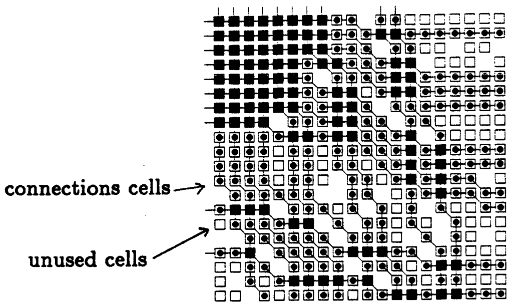


**Figure 12.** Configuration of cells into a mesh in the defective hexagonal array, p = 0.9

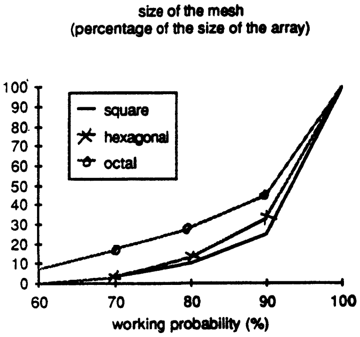


**Figure 13.** Efficiency of mesh configuration

| size | 40X40 | | | 80X80 | | | 120X120 | | |
|---|---|---|---|---|---|---|---|---|---|
| working probability | degree | | | degree | | | degree | | |
| | 4 | 6 | 8 | 4 | 6 | 8 | 4 | 6 | 8 |
| 70% | | 16 | 5.0 | | 40 | 5.3 | | 35 | 5.2 |
| 80% | 9.5 | 5.2 | 3.2 | 13 | 7.0 | 3.3 | 12 | 7.7 | 3.5 |
| 90% | 4.2 | 2.65 | 2.1 | 5.0 | 2.9 | 2.1 | 4.8 | 3.0 | 2.2 |
| 95% | 2.5 | 1.7 | 1.6 | 3.2 | 2.0 | 1.7 | 3.1 | 2.1 | 1.7 |
| 100% | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

**Table 4.** Delays on the embedded meshes

When the cell receive the messages, it tries to grow in the direction of the line. When the neighbor on the direction of the line is defective, it tries to grow on the direction specified on the message. If the cell cannot grow the line, it backtrack to its predecessor cell. Figure 12 shows the cells configured into a mesh.

Figure 13 shows the size of the mesh as the percentage of the array size, and Table 4 shows the delay of the configuration. As shown in the Figure 13 and Table 4, the efficiency of the configuration increases rapidly and the delay decreases rapidly with the increase of the number of the neighbors.

## Conclusion

As shown in this paper, self-configuration of cells into various computation graphs can be done efficiently when cells are adequately powerful. When we use the massively fault-tolerant cellular array for a particular application, we need to configure cells into a particular computation graph, and we can determine the defect rate of cells which allows acceptable efficiency of configuration from the data in this paper. We can change this acceptable defect rate by changing the interconnection patterns or the size of a cell to find the feasible implementation on available technology. Currently we are designing a wafer-scale signal processing chip using the massively fault-tolerant cellular array. The architecture is very homogeneous and simple, and shows the potential for high performance.

## References

[1] R. Aubusson and I. Catt. Wafer-scale integration - a fault tolerant procedure. *IEEE J. Solid State Circuits*, SC-13(3):339–344, June 1978.

[2] R. P. Brent and H. T. Kung. On the area of binary tree layouts. *Info. Proc. Letters*, 11(1):46–48, Aug. 1980.

[3] John W. Essam. Percolation theory. *Rep. Prog. Phys*, 43:833–912, 1980.

[4] D. Fussel and P. Varman. Fault tolerant wafer-scale architectures for VLSI. In *Proc. 9th Ann. Symp. on Comput. Arch.*, pages 190–198, 1982.

[5] J. W. Greene and A. El Gamal. Configuration of VLSI arrays in the presence of defects. *J. of ACM*, 31(4):694–717, Oct. 1984.

[6] John P. Hayes. A graph model for fault-tolerant computing systems. *IEEE Trans. Comput.*, C-25(9):875–884, Sep. 1976.

[7] K. S. Hedlund and L. Snyder. Wafer-scale integration of configurable, highly parallel processor. In *Proc. Int'l Conf. Parellel Processing*, pages 262–264, Aug. 1982.

[8] J. Hoshen, P. Klymko, and R. Kopelman. Percolation and cluster distribution. III. Algorithm for site-bond problem. *Journal of Stat. Phys.*, 21(5):583–599, 1979.

[9] Israel Koren. A reconfigurable and fault tolerant VLSI multiprocessor array. In *Proc. 8th Ann. Sym. on Comput. Arch.*, pages 425–442, 1981.

[10] Myoung Sung Lee and Gideon Frieder. ***Computations on massively defective integrated circuits.*** Technical Report, Univ. of Michigan Computing Research Lab., Ann Arbor, MI, to be published.

[11] F. T. Leighton and C. E. Leiserson. ***Wafer-scale integration of systolic arrays.*** Technical Report MIT/LCS/TM-236, Laboratory for Computer Science, MIT, Feb. 1983.

[12] F. Manning. An approach to highly integrated, computer-maintained celluar arrays. *IEEE Trans. Comput.*, C-26(6):536–552, Jun 1977.

[13] Robert Negrini, Mariagiovanna Sami, and Renato Stefanelli. Fault tolerance techniques for array structure used in supercomputing. *IEEE Computer*, pages 78–87, Feb. 1986.

[14] J. I. Raffel. On the use of nonvolatile programmable links for restructurable VLSI. In *Proc. Caltech VLSI Conf.*, pages 95–104, 1979.

[15] Arnold L. Rosenberg. The Diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comput.*, C-32(10):902–910, Oct. 1983.

[16] Vindo K. S. Shante and Scott Kirkpatrick. An introduction to percolation theory. *Adv. in Phys.*, 20:325–357, 1971.

[17] L. Snyder. Introduction to the configurable, highly parallel processor. *IEEE Computer*, 15(1):47–56, Jan. 1982.

# A Fault-Tolerant VLSI Matrix Multiplier

**P.J. Varman†**
**Department of Electrical and Computer Engineering**
**Rice University**
**Houston, TX-77001**

**I.V. Ramakrishnan††**
**Department of Computer Science**
**State University of New York at Stony Brook**
**Stony Brook, NY 11794**

## Abstract

Several approaches to the design of fault-tolerant arrays of processors with a view towards wafer-scale integration, have been proposed in the past. Using these approaches it is possible to design linear arrays that have several desirable features of an effective fault-tolerant design, that is, they facilitate testability, are easily reconfigurable, have short inter-PE wire lengths and make one hundred percent utilization of non-faulty processors. In contrast, no single scheme or combination of previously proposed schemes result in effective fault-tolerant implementation of 2-D arrays.

In [19] we presented a computing structure that has all the attractive implementation features of a linear array and can multiply matrices with a better time performance than a linear array. In this paper we examine the inter-relationship among the number of processors, the storage within a processor, internal bandwidth and the time complexity of matrix multiplication on such a model and establish lower bounds on the time complexity and the number of processors required as a function of the storage within a processor and the internal bandwidth. We then present a generalized algorithm which matches these bounds for arbitrary storage within a processor and arbitrary bandwidth.

An interesting side effect of our result is that the area and time complexities and the asymptotic complexity of the number of proceesors used by the algorithm in [19] and all previously known linear-array algorithms can be obtained as special cases of our generalized algorithm. Additionally, the techniques and the results of this paper can be readily adapted to obtain such a family of optimal algorithms for several other 2-D systolic algorithms.

## 1. Introduction

The ever growing demand for high performance computing, coupled with advances in integrated circuit fabrication technology has led to considerable interest in the design of VLSI architectures that realize high-performance parallel algorithms directly in silicon at modest costs. One such direction pioneered by Kung and Leiserson [12] is the concept of *systolic arrays* as a VLSI computing structure to implement high-performance parallel algorithms.

A systolic array consists of a collection of processing elements (PE's), interconnected in regular nearest-neighbor geometrical structures like a one-dimensional (1-D) linear array, or two-dimensional (2-D) rectangular or hexagonal array. A data item that has been retrieved from the memory by a systolic array, passes through several PE's before returning to the memory. This feature of using a datum from memory many times over without having to store and retrieve intermediate results gives rise to high computational throughput. Simplicity and regularity of systolic arrays make them suitable for VLSI implementation at minimal design costs. Systolic arrays find natural applications in signal and image processing where large matrix computations need to be performed rapidly.

Still higher performance can be had by application of wafer-scale integration (WSI) to the implementation of systolic arrays. Rather than dice a silicon wafer into chips as is usually done, the idea behind WSI is to assemble an entire system on a single wafer. Such an increased level of integration avoids the costs and performance loss associated with individual packaging of chips. However, since fabrication flaws in a wafer-sized circuit are inevitable it is necessary for these designs to be *fault-tolerant* so that wafers with defective components can still be used. The homogeneity of the PE's that make up a systolic array and their regular interconnection make systolic arrays attractive candidates for the application of WSI. Thus, several approaches to the design of fault-tolerant arrays of processors, with a view towards WSI, have been proposed in the recent past [6,9,11,13,17,21]. The designs resulting from all of these varied techniques, however, either ignore or fall short of meeting some of the desirable features of an effective fault-tolerant design - that the design facilitate testability, be easily reconfigurable, have short inter-PE wire lengths and make good utilization of non-faulty PE's. We will briefly highlight two of these approaches [11,17] as they form the basis for the research presented in this paper.

In [17] Rosenberg proposed the *Diogenes* methodology for the design of fault-tolerant VLSI arrays. The essence of this methodology is *linearization* of processor networks which are mapped onto collinear layouts of processors configured into the desired structure by appropriate switch settings on buses running parallel to the PE's. It provides scan-in/scan-out capability to enhance testability and configuration is achieved using a few control lines per processor. The PE's are scanned serially and are hooked into the network as and when they are determined to be fault free. Hence use of the Diogenes technique results in one hundred percent utilization of the good PE's. The switches can be set dynamically through the control lines which are accessible to the outside world thereby providing dynamic fault tolerance capability. Furthermore, Diogenes designs are modular, that is, chips designed in this manner can be cascaded together by connecting their corresponding buses. This feature can render feasible the use of a chip having only a few fault-free PE's, thereby increasing the effective yield.

A second approach tailored to fault-tolerant design of *systolic* arrays, attempts to alleviate the problem of clock rate degradation caused by the fact that wires connecting logically adjacent fault-free PE's may span a large physical distance. The essence of this approach involved retiming [14] of the systolic algorithm. We say that a systolic design is a retimed version of another design if the former differs from the latter by having additional delays on some of the communication links. Kung and Lam [11] and Varman [21] gainfully employed retiming to run systolic algorithms correctly without any degradation in their throughput even in the presence of faulty PE's. Specifically, for a linear-array systolic algorithm that is comprised of unidirectional data streams, retiming requires that the additional delay encountered by the data elements in each stream when bypassing a faulty PE be identical. This ensures that data elements that met at a PE in the fault-free array will indeed meet even in the presence of faulty PE's.

† Research supported by an IBM Faculty Development Award
†† Research supported by ONR grant N00014-84-K-0530 and NSF grant ECS-84-04399.

0190-3918/86/0000/0351 $01.00 © 1986 IEEE

By combining the Diogenes approach to restructuring processor arrays with the retiming schemes described in [11,21], effective fault-tolerant implementation of linear arrays meeting all of the desiderata enumerated earlier can be achieved, that is, the resulting designs are modular and can be easily tested and configured in the presence of faults. In addition, all the working cells can be utilized and more importantly signals need travel only short wire lengths in every clock cycle.

In contrast, no single scheme or combination of previously proposed schemes for fault-tolerant implementation of 2-D arrays simultaneously achieves all the desirable characteristics of a fault-tolerant linear array implementation. For instance, the Diogenes scheme results in long wire lengths between logically adjacent PE's (even in the absence of faulty PE's) and requires significantly larger area than a two-dimensional implementation. For matrix computations like multiplication of two n×n matrices, such a design makes use of $O(n^3)$ ‡ area (rather than the optimal $O(n^2)$ of a 2-D array implementation). Furthermore, if we assume linear propagation delay for signals (not an unrealistic assumption for the large circuits being considered in a wafer-scale environment [2,3]) then matrix multiplication requires $O(n^2)$ time (rather then the optimal $O(n)$ of a 2-D array implementation). On the other hand, incorporating retiming techniques for fault-tolerant implementation of 2-D arrays requires the use of numerous programmable delay registers and extra wiring channel capacity in the wafer in order to achieve high PE utilization [11]. This makes the technique cumbersome to use.

While a linear array is easy to implement and possesses excellent fault-tolerant properties, simulating 2-D systolic algorithms (like matrix multiplication) on it results in significant degradation in time performance. For instance, multiplication of two n×n matrices requires $O(n)$ time on a 2-D systolic array [12] whereas it requires $O(n^2)$ time on a linear array [1,8,10].

A question that naturally arises is whether there exists a computing structure that has all the attractive implementation and fault-tolerant features of a linear array and can simulate 2-D systolic algorithms with a better time performance that a linear array.

In [19] we gave an encouraging answer to this question by presenting a collinear VLSI array that retains all the desirable fault-tolerant characteristics of Diogenes designs but obviates the need for signals to travel more than a fixed physical distance in any clock cycle. On such an array we established a lower bound on time of $\Omega(n\sqrt{n})$ to multiply two n×n matrices. We also presented an optimal $O(n\sqrt{n})$ time matrix multiplication algorithm that used $O(n\sqrt{n})$ processors, $O(\sqrt{n})$ storage per processor and $O(\sqrt{n})$ internal bandwidth (that is, the number of data items that may be simultaneously transferred in unit time across any vertical line passing through the array). The $O(n^2)$ area and $O(n\sqrt{n})$ time required by our algorithm is significantly lower than the $O(n^3)$ area and $O(n^2)$ time requirement of a straightforward linearization of the 2-D systolic matrix multiplication algorithm on our array.

In this paper we examine the inter-relationship among processors, storage within a processor, internal bandwidth and time complexity of matrix multiplication on our model. We establish lower bounds on the time and processor complexity for matrix multiplication as a function of the storage within a processor and the internal bandwidth. We then present a generalized algorithm that matches these bounds for arbitrary storage within a processor and arbitrary bandwidth. An interesting side effect of our result is that the time and area complexities and the asymptotic complexity of the number of processors used by the algorithm in [19] and all previously described linear systolic array matrix multiplication algorithms in [1,4,8,10,15,16] can be obtained as *special* cases of our generalized algorithm.

To illustrate the key ideas in this paper we focus only on matrix multiplication. However the techniques and results of this paper can be readily adapted to obtain such a family of optimal algorithms for other 2-D systolic algorithmms like LU decomposition, QR factorization, solution of linear systems of equations, etc.

The rest of this paper is organized as follows. In the next section we lay the theoretical framework for the research reported in this paper. In section 3 we will elaborate on our computing structure and outline the design of matrix multiplication algorithms on this model. In section 4 we present some closing remarks. The details of our algorithm, its proof of correctness and issues regarding fault tolerance and modularity appears in [20].

## 2. Theoretical Framework

One of our major objectives is to retain the excellent fault-tolerant characteristics afforded by collinear implementations of processor networks but avoid the degradation in throughput (caused by a lower clock rate) that long inter-PE wires would impose. Making the clock rate independent of the inter-PE wire length in a collinear implementation may be achieved by introducing *buffers* in the wires at fixed physical separation (say between every pair of physically adjacent PE's) so that signals are clocked in and out of these buffers at every clock cycle. Fig. 2.1(a) is a collinear implementation of a 3×3 2-D array obtained using the Diogenes methodology. Fig. 2.1(b) is the same implementation with buffers (denoted by   in the figure) on the connecting wires.

While incorporating buffers in the interconnecting wires may appear to be a relatively minor change, it drastically changes the nature of computations on the architecture. Interconnecting wires in the non-buffered case are now actually *pipelines* through which *several* data items may *simultaneously* be in transit from a source PE to a destination PE. As discussed below, algorithms on such an architecture need to be carefully designed to exploit the potential afforded by having pipelined buses.

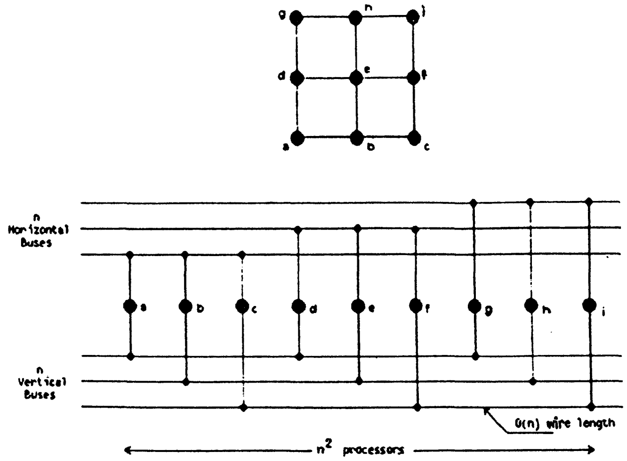


Fig. 2.1 (a): Linearization of 3×3 Mesh (without buffers)

‡ $f(n)=O(g(n))$ iff there exists constants c and $n_0$ such that $f(n) \leq c\, g(n)$ for $n \geq n_0$ and $f(n)=\Omega(g(n))$ iff $f(n) \geq c\, g(n)$ for $n \geq n_0$.

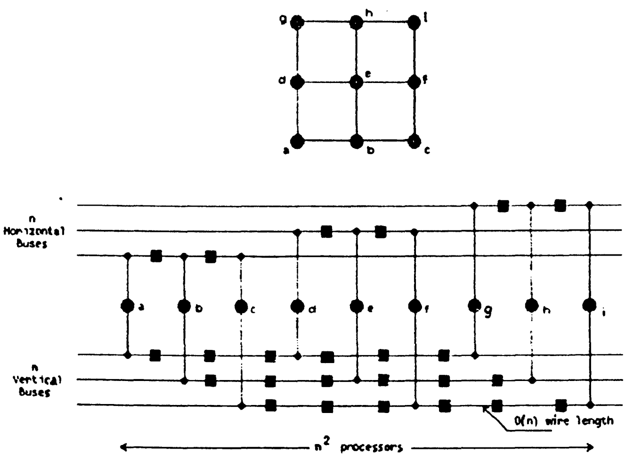


Fig. 2.1 (b): Linearization of 3×3 Mesh (with buffers)

First, consider a naive simulation of the well-known 2-D systolic algorithm for matrix multiplication [12] on our collinear pipelined implementation. It is easy to see that every communication step that transferred data elements between adjacent PE's along a column in the 2-D mesh would now require (n-1) clock cycles to move the element along the (n-1) buffers separating the corresponding PE's in the collinear implementation. Thus, the naive simulation would require $O(n^2)$ time rather than the O(n) required on a 2-D systolic array implementation. Even more interestingly, a theoretical result due to Gentleman [5] implies that as long as the PE's in the collinear implementation have only a constant (O(1)) amount of storage, then *any* matrix multiplication algorithm would require $O(n^2)$ time.

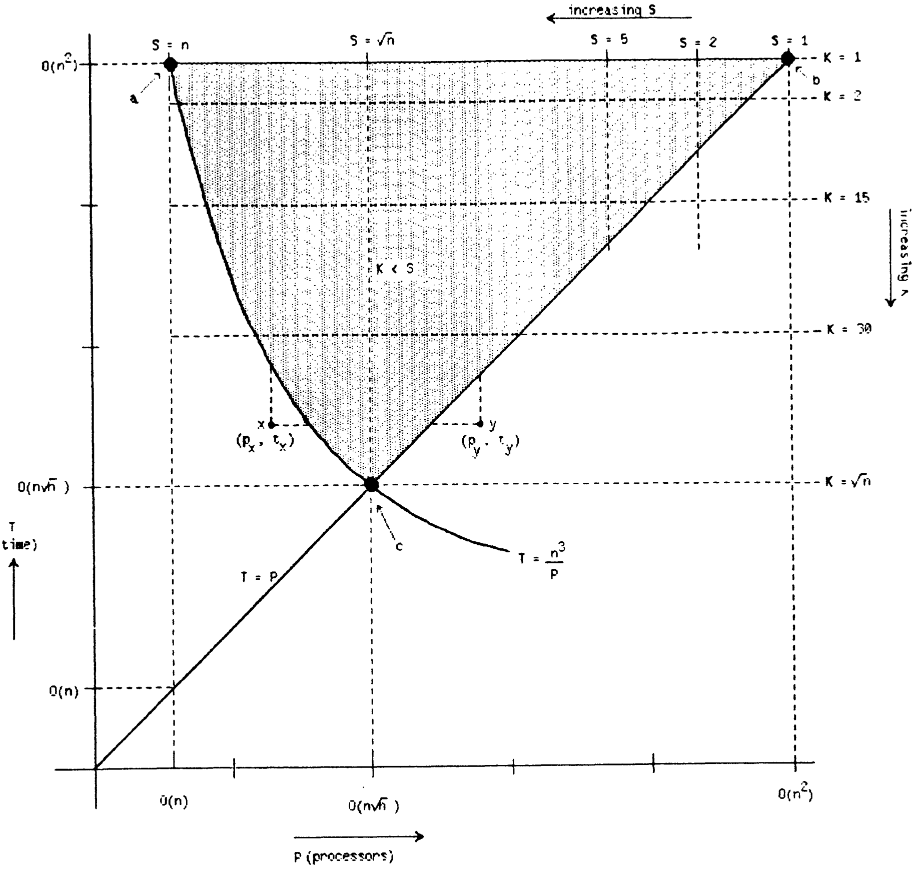
A consequence of the above result is that a minimum requirement for improving the time performance is to *increase* the *storage* within every PE. The intuitive reason for doing so is to *reduce* the *number* of PE's that take part in the computation and thereby decrease the maximum path length that any data item has to traverse during the computation. However, decreasing the number of PE's beyond a certain limit causes the computation to become *compute bound*, as we have too few PE's to perform the computation.

Thus, there exists an interesting tradeoff between the storage per PE and the execution time of the algorithm. In addition, as we shall show, the number of parallel wires in the collinear layout (the number of such wires is a measure of the number of data items that may be simultaneously transferred across any vertical line passing through the array, and will be denoted by the term *internal bandwidth*) also affects the time complexity of the algorithm in an interesting way. Thus we have a four-way tradeoff between the execution time, the storage per PE, the internal bandwidth and the number of PE's.

Our main contribution in this paper is to fully characterize the tradeoffs involved in such a collinear model. We present a generalized *family* of matrix multiplication algorithms on this model *parameterized* by the storage per PE (s) and internal bandwidth (k). All the algorithms in our family are *optimal* in that they make most efficient use of the given hardware resources and perform the computation in the minimum time possible.

The interplay between these resources (execution time, the storage per PE, the internal bandwidth and the number of PE's) can be summarized by the following graph (see Fig. 2.2).

Fig. 2.2



The curve *ac* denotes a lower bound on time of $\frac{n^3}{p}$ whereas the line *bc* denotes a lower bound of p which is the time it takes for an element to travel from one end of the collinear array to the other. Observe that the lower bound on processing time ($\frac{n^3}{p}$) matches the lower bound on the communication time (p) when p is $n\sqrt{n}$. For p either less or greater than this, one of the two times dominates. A rigorous proof of this $\Omega(n\sqrt{n})$ lower bound on time for multiplication of two $n \times n$ matrices appears in [19].

Note that to store $n^2$ elements in $n\sqrt{n}$ PE's we require $\sqrt{n}$ storage per PE. Moreover, $\sqrt{n}$ internal bandwidth suffices to accomplish transfer of the $n^2$ elements across any point in the array in $n\sqrt{n}$ time. Therefore both s and k are $\sqrt{n}$ at point *c*.

Feasible algorithms (that is, those that do not violate the time lower bound) are bounded within the region enclosed by the curve *ac* and the lines *bc* and *ab* for p in the range O(n) to O($n^2$) and T in the range O($n\sqrt{n}$) to O($n^2$) (the shaded region in the figure).

From information-flow arguments [18] it can be easily established that for any internal bandwidth k, the time required to multiply two $n \times n$ matrices must be at least $\frac{n^2}{k}$. The intuitive reason for this is that in any (approximately) equal-sized partition of the chip at least $n^2$ bits of information must flow between the two partitions and hence for any internal bandwidth k the lower bound follows. To store the $n^2$ elements we require at least $n^2$ storage (see [15,22] for a proof). Therefore, within the feasible region, the lower bounds on time and number of processors are $\Omega(\frac{n^2}{k})$ and $\Omega(\frac{n^2}{s})$ respectively.

Observe that any point x ($p_x$,$t_x$) in the region below *ac*, represents the situation where we have too few processors ($p_x$) to compute the $n^3$ scalar products in time $t_x < \frac{n^3}{p_x}$. Moving horizontally to the right (that is, keeping k fixed) would correspond to increasing the number of PE's and underutilizing the storage per PE. On the other hand, moving vertically up (that is, keeping s fixed and therefore the number of PE's fixed) we underutilize the inter-PE bandwidth (k) and also incur a penalty in time.

Observe also that any point y ($p_y$,$t_y$) in the region below line *bc*, corresponds to a situation where in time $t_y$ ($t_y < p_y$) a data item has to be distributed to $p_y$ PE's ($p_y = \frac{n^3}{s}$). Moving horizontally to the left corresponds to decreasing the maximum path length by reducing the number of PE's. This can only be achieved by increasing the storage per processor which may not always be possible for PE's embedded in a chip. We would therefore pay the time penalty of moving vertically to the top.

In this paper we will describe the design of a generalized *family* of systolic algorithms in the feasible region, parameterized by s and k that *simultaneously* meets both the time and processor lower bounds. Now the algorithms in [1,8,10] use O(n) PE's, O(n) storage per PE and require O($n^2$) time. These algorithms operate at point *a* in Fig. 2.2. When s is n our generalized algorithm will have the same processor, storage and time complexities. The modular algorithm in [15] requires O($n^2$) PE's, uses O(1) storage per PE and also requires O($n^2$) time. It operates at point *b* in Fig. 2.2. Our generalized algorithm will have identical processor, storage and time complexities when s is 1. Using a single bus (k=1) in our generalized algorithm results in the family of algorithms in [4,16] that operate along the line *ab* in Fig. 2.2. Lastly, the algorithm in [19] that operates at point *c* is obtained from our generalized algorithm when both s and k are $\sqrt{n}$.

## 3. Computing Model

Let A and B be two $n \times n$ input matrices and let C be the result matrix. Let s ($1 \le s \le n$) be the storage per PE and k ($1 \le k \le \sqrt{n}$) be the number of buses. (Observe in Fig. 2.2 that we never require more than $\sqrt{n}$ buses in the feasible region.) The cells used for multiplication are arranged on a straight line and communicate by multiple buses as shown in Fig. 3.1 below for the case n=4 and k=s=2.

The $n^2$ elements of matrix C are stored in the local memories of the PE's (s of them in each PE) and updated in-situ as elements of matrices A and B are made available to the PE's. There are four distinct types of buses which transport elements of matrix A (ABUS), elements of matrix B (BBUS), control signals (CNTRLBUS) and address signals (ADDRBUS) from the I/O port at the left end of the array to the different PE's. There are k buses of each type and each PE is hooked to *exactly one* ABUS, BBUS, CNTRLBUS and ADDRBUS. This makes our structure attractive for implementation purposes.

We can visualize all the cells in the layout as being subdivided into $k^2$ contiguous blocks where each block is a contiguous sequence of w cells ($w = (\frac{n}{k} - 1) \left\lceil \frac{n}{ks} \right\rceil + \frac{n}{k}$. See [20] for more details). Each ABUS is connected to kw (physically) consecutive PE's. The first ABUS is connected to PE's 1,2,..,w,w+1,..,2w,..,kw, the second one is connected to PE's kw+1, kw+2,..,2kw and so on. Each BBUS is connected to each of the cells in k different blocks where each block is separated by a run of (k-1)w PE's that are connected to the remaining k-1 BBUS's. Thus the first BBUS is connected to each of the PE's in the first block comprising of PE's 1,2,..,w and then to each of the PE's in the block consisting of PE's kw+1,kw+2,..,(k+1)w and so on. The second BBUS is connected to the block with PE's w+1,w+2,..,2w and then to the block with PE's (k+1)w+1,(k+1)w+2,..,(k+2)w and so on. The organizations of the CNTRLBUS and ADDRBUS are similar to the ABUS. Observe that the above interconnection transforms each of the $k^2$ blocks into a linear-array block. As an illustration, observe in Fig. 3.1 (w is 3 as n is 4 and s is 2) that $(\text{ABUS})_1$ is connected to the first six PE's whereas $(\text{ABUS})_2$ is connected to the remaining six PE's. $(\text{BBUS})_1$ is first connected to the block consisting of PE's 1, 2 and 3 then to the block with PE's 7, 8 and 9 whereas $(\text{BBUS})_2$ is connected first to the block with PE's 4, 5 and 6 followed by the block consisting of PE's 10, 11 and 12.

Finally, all the processors operate synchronously and are driven by a global clock.

Associated with each bus is a delay equal to the number of clock ticks required to move a data element traveling along the bus between consecutive PE's. The delay associated with the ABUS, CNTRLBUS is 1, as shown by the "unit-delay" buffers on these buses (  's in Fig. 3.1). whereas the delay encountered on the BBUS is 2 (  's in Fig. 3.1). Therefore, in our model no element has to traverse more than a fixed physical distance in one clock cycle and this distance is *independent* of the *size* of the array. In our model therefore, signal delay is proportional to the distance it has to propagate. Such a delay model appears appropriate for the large circuits produced by wafer-scale integration. See [3] for a justification of this delay model and [2] for a detailed discussion of the various delay models.

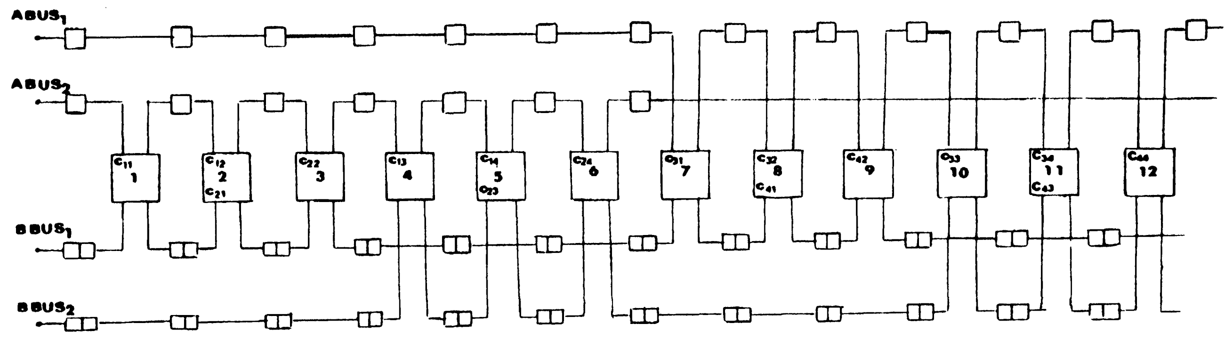


Fig. 3.1: Collinear Network of Processors (n=4)

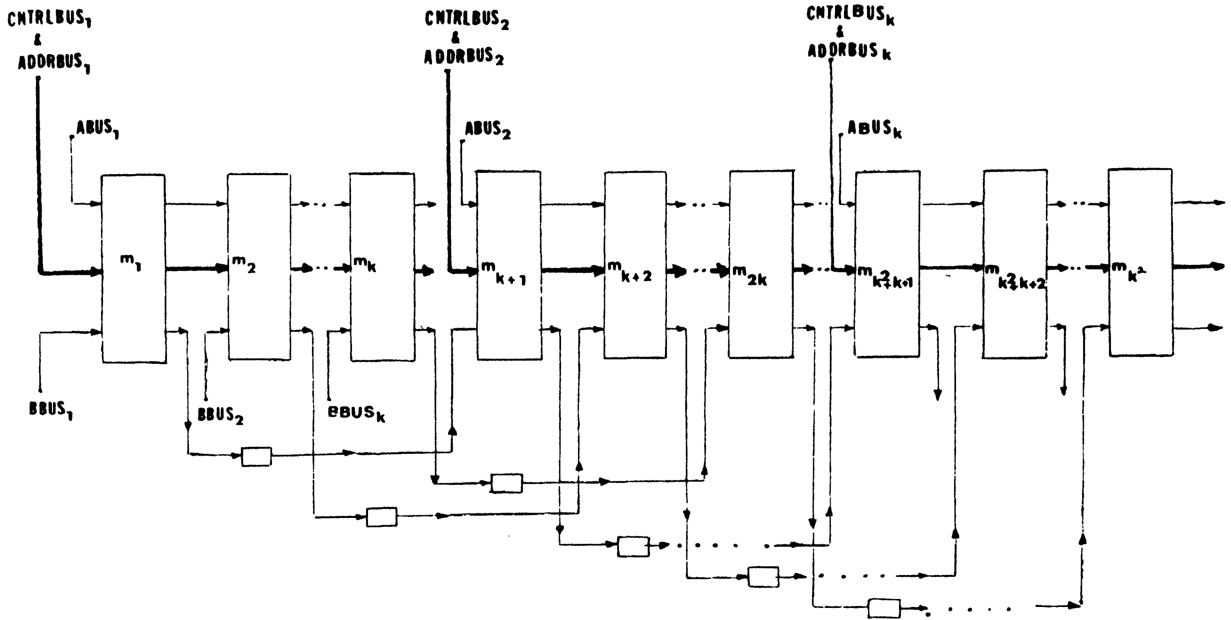


Fig. 3.2: Modular Arrangement

Fig. 3.2 is a logical arrangement of our computing structure. The linear array blocks are indexed $m_1, m_2, .., m_{k^2}$.

Note that we connect together the BBUS of blocks whose indices differ by k whereas the ABUS,CNTRLBUS and ADDRBUS of k adjacent blocks are connected. The difference in indices between the last processor in block $m_i$ and the first processor in block $m_{i+k}$ is (k-1)m+1. Recall that a data element that is travelling along the BBUS requires two ticks to move between consecutively indexed processors. Hence an element of matrix B that emerges from the last cell in $m_i$ reaches the first cell in block $m_{i+k}$ in 2((k-1)m+1) clock cycles. The on the BBUS between blocks models this delay. By distributing the delay along the BBUS (as shown in Fig. 3.1) the clock rate is made independent of the physical separation between the blocks.

## 3.1. Multiplication Algorithm

We will now outline our algorithm for multiplying two n×n matrices whose elements are pipelined through the buses in our computing structure. The elements of the result matrix C are stored in the local storage of PE's, one per word and updated in situ as elements of matrices A and B are made available to the PE's.

We first divide matrix C into $k^2$ submatrices each of size $\frac{n}{k} \times \frac{n}{k}$. Each of these submatrices are computed in a linear-array block of w cells. The first block of k submatrices are computed in the first k linear-array blocks, the second block in the next set of k linear-array blocks and so on.

If $s = \frac{n}{k}$ then we can compute the product of these submatrices using a linear array of $O(\frac{n}{k})$ PE's and $O(\frac{n}{k})$ storage per PE [1,8,10]. These algorithms store one of the input matrices (one row per PE) and pipeline the columns of the other matrix through the array. After all these columns pass through a PE, an entire row of the result matrix gets computed in that PE and there is sufficient storage within it to store the computed row. However the difficult and interesting case arises when $s < \frac{n}{k}$. This is because we have less storage within a PE to store an entire row of the input or result matrix. In contrast to the previous case (wherein a PE must use all the elements of the second input matrix passing through it in order to compute an entire row of the result matrix) a PE now will be required to use only certain of these elements in order to compute part of an entire row of the result matrix. We can handle both these cases by using the linear-array algorithm in [16] to compute the submatrices. Note that the $a_{ij}$'s and $b_{ij}$'s are required in different linear-array blocks. The final details of our algorithm involves carefully scheduling their arrival to the different blocks so that the correct product terms meet at the correct time in the appropriate PE and accomplish the entire computation in optimal $O(\frac{n^2}{k})$ time.

We estimate the time and processor complexity of our algorithm as follows (a detailed analysis appears in the appendix).

Recall that in the feasible region $\Omega(\frac{n^2}{k})$ and $\Omega(\frac{n^2}{s})$ are the lower bounds on time and processor complexity respectively.

We use $k^2w$ PE's and w is $O(\frac{n^2}{k^2 s})$ and hence the processor complexity is $O(\frac{n^2}{s})$.

For estimating the time complexity of our algorithm, first note that we need to compute $\frac{n^3}{k^3}$ scalar products of a submatrix in each linear-array block. This requires $O(\frac{n^2}{k^2})$ time which is $\leq O(\frac{n^2}{k})$. Secondly, an element from matrix B need travel a maximum path length of $O(\frac{n^2}{s})$. Now $k \leq s$ in the feasible region, and hence the time complexity is bounded by $O(\frac{n^2}{k})$.

Thus, for any s and any k, our algorithm achieves *optimal* time and *processor* bounds in the feasible region.

An interesting byproduct of our generalization is that the area and time complexities and the assymptotic complexity of the number of processors used by all known linear-array matrix multiplication algorithms [1,4,8,10,15,16] and the algorithm in [19] appear as special cases of our generalized algorithm. We show this as follows.

The algorithms in [1,8,10] use O(n) PE's, O(n) storage per PE and require $O(n^2)$ time. These algorithms operate at point *a* in Fig. 2.2. The algorithm resulting from our generalized algorithm when s is n will have the same processor, storage and time complexities. The modular algorithm in [15] requires $O(n^2)$ PE's, uses O(1) storage per PE and also requires $O(n^2)$ time. It operates at point *b* in Fig. 2.2. Our generalized algorithm will have identical assymtotic complexities for time, area and number of processors when s is 1. Using a single bus (k=1) in our generalized algorithm would result in the family of algorithms in [4,16] that operate along the line *ab* in Fig. 2.2.

Lastly, when s and k are both $\sqrt{n}$ (that is, they are *balanced* ), our generalized algorithm achieves the optimal time bound of $O(n\sqrt{n})$ using $\sqrt{n}$ buses, $\sqrt{n}$ storage per PE and $O(n\sqrt{n})$ PE's. All these complexities are identical to the algorithm in [19]. Some features of this algorithm are worth noting. Our balanced algorithm operates at point *c* in Fig. 2.2. Observe in the figure that the lower bound on communication time matches that of the processing time when n is $\sqrt{n}$ (point *c* in the figure). We compare this balanced algorithm with the naive simulation of a 2-D systolic matrix multiplication algorithm on the collinear structure in Fig. 2.1(b).

As mentioned earlier, the primary drawback with such a simulation is that n×n matrix multiplication on it requires $O(n^3)$ area, 2n buses and $O(n^2)$ time. Note that this is the case despite the *buffers* on the buses to pipeline several elements along them.

Fig. 2.1(a) is the collinear implementation obtained using the Diogenes design methodology. However, a major drawback with these designs is the long wire lengths between logically adjacent PE's (even in the absence of faulty PE's) resulting in significant degradation in throughput (due to a slower clock speed). Simulating two n×n matrices on this structure requires $O(n^3)$ area. Furthermore, if we assume that signals propagate a fixed physical distance in every cycle, then the time required for multiplication becomes $O(n^2)$.

In contrast, the time complexity of our balanced algorithm is only $O(n\sqrt{n})$. Furthermore, we require only $n\sqrt{n}$ processors (rather than the $n^2$ PE's required for simulation) and only $4\sqrt{n}$ (rather than the 2n required for simulation). The total area occupied by the $n\sqrt{n}$ processors is $O(n^2)$ (as each processor requires $\sqrt{n}$ storage). Also, each bus is $n\sqrt{n}$ long and hence the wiring area occupied by all the $\sqrt{n}$ buses is $O(n^2)$. Thus the total area required by our balanced algorithm is $O(n^2)$. In contrast, the simulated 2-D systolic algorithm requires $O(n^3)$ area. Thus our balanced algorithm succintly demonstrates that we can preserve the ease of testability and configurability that are characteristics of the linearization approach and also have better resource utilization and better time performance as well.

## 4. Conclusions

Several approaches to the design of fault-tolerant arrays of processors, with a view towards wafer-scale integration, have been proposed in the past. The designs resulting from all of these varied techniques, however, either ignore or fall short of meeting some of the desirable features of an effective fault-tolerant design, namely, that the design facilitate testability, be easily reconfigurable, have short inter-PE wire lengths and make good utilization of non-faulty PE's.

Some of these methodologies can be combined to produce effective fault-tolerant implementation of linear arrays meeting all of the desiderata stated earlier. In contrast, no single scheme or combination of previously proposed schemes for fault-tolerant implementation of 2-D arrays simultaneously achieves all the desirable characteristics of a fault-tolerant linear array implementation.

Diogenes designs proposed by Rosenberg are very attractive for fault-tolerant implementation of 2-D systolic algorithms. However, a major drawback with Diogenes designs is the long wire length between logically adjacent PE's (even in the absence of faulty PE's) resulting in significant degradation in throughput. Moreover, these designs require significantly large area for simulating 2-D systolic array algorithms. For matrix computations like multiplication of two n×n matrices, such a design makes use of $O(n^3)$ area. Furthermore, if we assume that signals propagate a fixed physical distance in every cycle then matrix multiplication requires $O(n^2)$ time.

In [19] we presented a collinear VLSI array that retains all the desirable fault-tolerant characteristics of Diogenes designs but obviates the need for signals to travel more than a fixed physical distance in any clock cycle. In our model, all signals travel a fixed physical distance in any clock cycle, *even* in the presence of faulty processors (that is, the clock rate is independent of both the number of faulty and non-faulty PE's). This feature of our model requires us to use retiming to ensure the correctness of our systolic algorithm despite the presence of faulty PE's. Their presence only contribute to an *additive* increase (equal to the number of bypassed faulty PE's) in the time complexity of our algorithm. We also described an algorithm to mutiply two n×n matrices in optimal $O(n\sqrt{n})$ time and $O(n^2)$ area.

In this paper we examined the inter-relationship among processors, storage within a processor, internal bandwidth and time complexity of matrix multiplication on our model. We established lower bounds on the time complexity and the number of processors required for matrix multiplication on our model as a function of the storage within a processor and the internal bandwidth. We than presented a generalized algorithm that matches these bounds for arbitrary storage within a processor and arbitrary bandwidth. An interesting side effect of our result is that the time and area complexities and the assymptotic complexity of the number of processors used by the algorithm in [19] and all previously described linear systolic array matrix multiplication algorithms in [1,4,8,10,15,16] can be obtained as *special* cases of our generalized algorithm.

To illustrate the key ideas in this paper we focussed only on one matrix computation (namely dense matrix multiplication) although our techniques can be easily extended to handle other important problems including LU decomposition, QR factorization and solution of linear systems of equations.

## References

[1] J. Bentley and T. Ottmann, "The Power of One-Dimensional Vector of Processors," Universitat Karlsruhe, Bericht 89, (April 1980).

[2] G. Bilardi, M. Pracchi and F.P. Preparata, "A Critique and Appraisal of VLSI Models of Computation," *VLSI Systems and Computations*, Computer Science Press, (1981), pp. 81-88.

[3] B. Chazelle and L. Monier, "A Model of Computation for VLSI with Related Complexity Results," *JACM*, Vol. 32, No. 3, (July 1985), pp. 573-588.

[4] A.L. Fisher, "Memory and Modularity in Systolic Array Implementations," *Proceedings of the 1985 International Conference on Parallel Processing*, (August 1985), pp. 99-101.

[5] W.M. Gentleman, "Some Complexity Results for Matrix Computations on Parallel Processors," *JACM*, 25(1978), pp. 112-115.

[6] J.W. Greene and A. Gamal, "Area and Delay Penalties in Restructurable Wafer-Scale Arrays," *JACM*, (October 1984).

[7] G.H. Hardy and E.M. Wright, "Introduction to the Theory of Numbers," Oxford University Press, Fifth Edition, (1978).

[8] A.V. Kulkarni and D.W.L. Yen, "Systolic Processing and an Implementation for Signal and Image Processing," *IEEE Transactions on Computers*, C-31, No. 10, (October 1982), pp. 1000-1009.

[9] I. Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," *Proceedings of the Eighth Annual Symposium on Computer Architecture*, (August 1982), pp. 262-264.

[10] H.T. Kung, "Systolic Algorithms for the CMU WARP Processor," *Seventh International Conference on Pattern Recognition*, (July 1984), pp. 570-577.

[11] H.T. Kung and M. Lam, "Wafer-Scale Integration and Two-Level Pipelined Implementation of Systolic Arrays," *Proceedings of the MIT Conference on Advanced Research in VLSI*, (January 1984).

[12] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," Sparse Matrix Proceedings 1978, I.S. Duff and G.W. Stewart (editors), SIAM (1979), pp. 256-282.

[13] F.T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Transactions on Computers*, C-34, No. 5, (May 1985), pp. 448-461.

[14] C.E. Leiserson and J.B. Saxe, "Optimizing Synchronous Systems," *Journal of VLSI and Computer Systems*, Vol. 1, No. 1, (1983), pp. 41-68.

[15] I.V. Ramakrishnan and P.J. Varman, "Modular Matrix Multiplication on a Linear Array," *IEEE Transactions on Computers*, C-33, No. 10, (November 1984), pp. 952-958.

[16] I.V. Ramakrishnan and P.J. Varman, "An Optimal Family of Matrix Multiplication Algorithms on Linear Arrays," *1985 International Conference on Parallel Processing*, (August 1985), pp. 376-383, (accepted for publication in the IEEE Transactions on Computers).

[17] A. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Networks of Processors," *IEEE Transactions on Computers*, C-32, No. 10, (October 1983), pp. 902-910.

[18] J.E. Savage, "Area-time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," *Journal of Computer and System Sciences*, 20:3, pp. 230-242.

[19] P.J. Varman and I.V. Ramakrishnan, "On Matrix Multiplication Using Array Processors," *Twelfth International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 194, (July 1985), pp. 487-496.

[20] P.J. Varman and I.V. Ramakrishnan, "A Fault-Tolerant VLSI Matrix Multiplier,", Technical Report 85/29, Department of Computer Science, SUNY at Stony Brook, (November 1985).

[21] P.J. Varman, "Wafer-Scale Integration of Linear Arrays,", PhD Thesis, University of Texas at Austin, (August 1983).

[22] J.E. Vuillemin, "A Combinatorial Limit to the Computing Power of VLSI Circuits," *Proceedings of the Twenty-First Annual IEEE Symposium on Foundations of Computer Science*, pp. 294-300.

# FAIL SAFE DISTRIBUTED FAULT DIAGNOSIS OF MULTIPROCESSOR SYSTEMS*

Chung Yang Chiang and Chuan-lin Wu
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin,TX 78712

Abstract -- Techniques for distributively, reliably diagnosing multiprocessor systems are presented in this paper. Based on these techniques,trustworthy diagnostic results on the status,faulty or fault free,of every processor node will be revealed by fault free processors. An assumption which has been commonly made in most papers and might lead to totally incorrect results is eliminated for fail safe purpose. The assumption puts a limit on the number of existing faulty processors. Capabilites of the diagnosis system is analyzed in terms of trustability,diagnosability,coverage and mean diagnosability. Comparison to an existing diagnosis approach is also provided on diagnosing system using a multistage topology. The results single out the uniqueness of our approach on the fail safe diagnosis.

## 1. Introduction

A general guideline has been established to avoid introducing extra faults which can lead a fault tolerant system to earlier destruction whenever system recovery schemes, such as fault detection,diagnosis, isolation and reconfiguration,for restoring the system to operating mode are called for[1].

A lot of research works have been done to mold processor nodes together to form parallel processing systems. Yet,only a little effort has been exerted on how to reliably manage the system whenever faults occur. Fault detection and location are the first two steps to take to avoid further damages to the system. Basically there are two kinds of fault diagnosis methods. The first is the centralized one in which an external processor takes the responsibility of locating the faulty processors. The second is the distributed one in which every processor involves the execution of the diagnosis procedure. For a large and/or widely dispersed system,centralized diagnosis is not feasible due to the limitation of communication links. Besides,those diagnosis processors will form the hardcore of the system. For nonrepairable and/or autonomous systems,only distributed diagnosis is feasible.

Valid fault detection and location methods should imply one hundred percent reliability in carrying out the job. They should never come up with false diagnostic results. Otherwise,these methods would be as harmful as those faults which call for these methods since they could introduce more faults into the system. Previous papers in fault diagnosis tend to put a limit on the number of existing faults[2-8]. If more faults than the assumed limit occur,often times it is quite possible,depending on the reliability and size of the system,the system will simply yield false results. This is especially true for a large, distributed system in which only distributed fault diagnosis is feasible. There should be no room at all for uncertainty. A very small percentage of increase of uncertainty will result in dramatic decrease of system reliability as can be observed in [9]. This paper reports new techniques and concepts for reliably diagnosing multiprocessor system.

In section 2,we present a system graphs model and a fault model which forms the basis of a new fault diagnosis algorithm. Section 3 consists of the diagnosis algorithm and proof of the validity of this algorithm. Comparisons of coverage,trustability and mean diagnosability of systems with different diagnosability are presented in section 4. Section 5 concludes the work in this paper.

## 2. System Graphs and Fault Model

### 2.1 System Graphs

A system interconnection graph is an undirected graph showing the interconnections among processors of the system. Yet,test graph is a graph which shows the relationship between testing and tested processors. Interconnrction graph is the underlying graph of test graph. An example of these graphs is shown in Fig. 1. A directed test graph is depicted in Fig. 1.a in which the two processors linked by the arrowheaded link are associated with each other with the processor to which the arrowhead is incident defined as the tested processor and the other as the testing processor. Fig. 1.b is an undirected test graph in which every processor is both a testing and tested processor to its neighboring processors. In this particular case,the system test graph is the same as the interconnection graph.

A processor has two roles : it is a tester at one time,and a testee at another time. As a testing processor,it initiates its tested processors to commence self test and monitors the results from its tested processors. As a tested processor,it receives start signals from its testing processors to start self

* This work is supported in part by a grant from IBM Corporation.

0190-3918/86/0000/0358 $01.00 © 1986 IEEE

test,reports its own status and reroutes received fault vectors from its testees to its testers.

## 2.2 Fault Model

The fault model is similar to most of the ones presented in previous papers[2-8] except one point,that is,no assumption on the number of allowable faults is ever made in order to avoid false diagnostic results and therefore possible system catastrophy.

The fault model is defined as follows :

(1) a faulty processor might modify the fault vectors sent to it by its testees.

(2)only permanent fault is considered. That is,once a processor is identified as faulty by any fault free processor,it will be regarded as faulty even though other processors might identify it as fault free.

(3)a fault free processor can become faulty during diagnosis period.

Link faults will be treated as proceesor faults since it is technically difficult to differentiate between link faults and processor faults. If that happens,the system simply loses those fault free processors which have association with the faulty links.

Since the diagnosis system is based on a distributed algorithm and no global clock has ever been provided as a system wide synchronization mechanism,it is impossible for processors to know the exact time of occurences of any link or processor faults. The only condition for the processors to differentiate link faults from processor faults is that they know exactly when faults have occurred. That implies a global,nonskewed clock should be implemented,which is impossible. The following paragraph explains in a circumstance on how link faults and processor faults can be differentiated by systems adopting distributed diagnosis algorithms.

For instance,we have 2 processors testing a common processor,the first tester identifies it as fault free and the second one identifies it as faulty due to the broken or stuck link between them. Following situations are possible :

(1)If the second tester identifies the link fault before the first one,then every processor will receive time-stamped messages from both the second tester indicating the tested processor as faulty and the first tester indicating it as fault free. Since only permanent fault is considered and fault condition was located first in terms of global clock,processors shall resolve this situation as a link fault. And the faulty link is the one that connects the second tester and the tested processor. In this case,the useful resource in this tested processor will not be dispensed. Only the faulty link will be isolated.

(2)Yet, if the first tester identifies the fault free processor first and the second one identifies the tested processor as faulty due to broken or stuck link between them after the first tester, then it is impossible to tell if it is a link fault or processor fault since the situation is the same as that of a fault free processor becoming faulty later before it is tested by the second testers. Unless the first tester,after receiving this conflicting message,can test this tested processor again and issue a later-time-stamped message indicating this supposedly faulty processor is still fault free.

# 3. Reliable Distributed Diagnosis Algorithm

## 3.1 Fail Safe Distributed Algorithm

The initiation and execution of the system fault diagnosis is assumed fully distributed. The following algorithm is for fault diagnosis only. Fault detection at individual processor is assumed. In this diagnosis system,we assume a certain number of processors can periodically initiate fault diagnosis cycles to locate faulty processors. The number of these watchdog-type processors should be greater than the expected degree of system fault tolerance in order to provide the necessary fault tolerance. Whenever the first processor initiates current cycle,every other processors which receive the initiation message,in addition to starting the diagnosis cycle,should also adjust its own internal clock to prepare for initiating next cycle. For the case in which faults did occur before the cycle is started,the effect of faults will be confined to only one cycle.

Definition 1 : Diagnosability -- A system is said to be t-fault self diagnosable if and only if each processor in the system can correctly identify all t faulty processors.

Diagnosability should not be taken as the sole gauge of the system diagnosis capability. The degree of confidence on the yielded results should be seriously taken into consideration in addition to diagnosability. Some might argue that we can always construct a system with diagnosability as high as possible. Yet no matter how complicated the system and therefore how high the diagnosability might be,it is always possible that the number of faults might surmount the diagnosability. On the other hand, for systems in which the diagnosability is low due to physically restricted structure of the system,the possibility of more fatal faults is even higher.

The algorithm is equally applicable to every processor in the system.The algorithm is written in C-like language.

Notations

(1) f[i] == fault vector maintained locally by p(i) (processor with logical id of i).

  (a) f[i][k] == kth element in f[i] representing the status of p(k) with the following possible values if k is not equal to i :

    0 : if p(k) is fault free.
    1 : if p(k) is faulty.
    x : status of p(k) is undetermined.

  (b) f[i][i] is the status of p(i) itself with only two possible values of 0 or 1.

(2) F[i] == fault vector sent by p(i) == f[i] ; F[i][k] == kth element in F[i].

Node p(j),j=0,1,....,n-1,after receiving diagnosis initiation messages,will perform following algorithm :

```
{
INITIALIZATION :
/*self test and initialization*/
broadcast diagnosis initiation message to
neighboring(both tested and testing) processors;
SELF_TEST :
if(self test passed)
    {
    f[j] = [$i_{n-1}$,..,$i_j$,..,$i_1$,$i_0$] = [x,x,..,0,..,x,x];
    send self test status to p(j)'s testers;
    }
else {
    f[i] = [x,x,..,1,..,x,x];
    send self test status to p(j)'s testers;
    break;
    }/*faulty processor makes self dormant if
      possible*/
while ((at least one element in f[j] undetermined) and
        (time not up yet))
    {
TEST : /*testing testees p(i)'s of p(j)*/
    for (every p(i) tested by p(j))
        switch (status of p(i)){
        case FAULT_FREE :
            f[j][i] = 0;
            break;
        case FAULTY:
        case NO_RESPONSE and TIMED_OUT :
            f[j][i] =1;
            break;
        case NO_RESPONSE and TIME_NOT_UP :
            break;
        } /*switch*/
BROADCAST : /*broadcast newly modified fault
vector to testers p(k)'s of p(j) if any*/
   while((any F[i] received from fault free p(i))) {
     if(F[i] results in changes of any element in f[j] from
       x to 1 or from 0 to 1)
       {
       initiate self test of p(i);
       if (p(i) is found fault free and F[i][i] ==0)
            update f[j] with F[i] from p(i);
       else    /*p(i) found faulty*/
            f[j][i]=1;
       }/*if*/
     broadcast newly changed element in f[j] to
     nonfaulty p(k)'s which test p(j);
     }/*while*/
   }/*while*/
if(at least one element in f[j] is undefined)
FAILED:
   for(nonfaulty p(k)'s which test p(j))
         broadcast failure message to p(k);
else{
   for(nonfaulty p(k)'s which test p(j))
         broadcast done message to p(k);
   while(at least one done message not yet received
   from fault free processors and time not up yet and
   no failure message received) ;
   if(timed out)
         go to INITIALIZATION;
   else
         if(failure message received)
           go to FAILED;
   }/*else*/
}/*main*/
```

The failure message indicates some of the fault free processors fail to generate decisive and unanimous result,although other fault free processors can locate all faulty processors.

### 3.2 Definition and Theorem

Definition 2 : Connectivity -- For an undirected(directed) test graph,the (vertex) connectivity $K_v$ is the minimun number of vertices whose removal from the graph results in a disconnected (weakly connected or disconnected)graph [10].

The test graph of Fig. 1.a is a directed graph with connectivity of 2. Yet ,the test graph of Fig. 1.b is an undirected graph with connectivity of 4.

When the number of faults is equal to the connectivity of the test graph,then the diagnosis system will fail us in some cases as the following paragraphs show. By failing us,we mean the diagnosis system found that current fault situation is undiagnosable due to lack of unanimous and decisive diagnostic result.

Assume the test graph of a multiprocessor system has connectivity of t,then the following cases are the situations in which the diagnosis system fails when the number of faults is not less than t.

case 1: Any processor with all its t testers being faulty. In this situation,the status of this particular processor will be unknown to other processors in the system.

case 2: Any processor with all its t testees being faulty. This particular processor will be totally blocked from the diagnostic information flowing around the system.

case 3 : There are some special cases with not less than t faults which also result in failed system.

As long as the number of faults is less than t,the diagnosis system always comes up with decisive and unanimous result.

From the above cases,we observe that the diagnosability of the system proposed in this paper is always 1 less than the connectivity. The main reason behind this decrease of diagnosability,as compared with that of the diagnosis systems proposed in other papers,is to achieve fail safety. Yet,it doesn't at all imply the system has an inferior capability of locating existing faults. On the contrary, it has a better capability.

The following theorem verifies that when the number of faults is less than the connectivity of the system,the system can always locate all faults by adopting the previous algorithm. And when the number of faults is eqaul to or greater than the system connectivity, the algorithm never comes up with false diagnostic results.

Theorem 1 - Using the above algorithm,a system with test graph T of $K_V$ equals t will either correctly locate all faulty processors or not incorrectly locate any faulty processors.

Proof : We will prove the validity of the algorithm in three cases.

(1)if the number of faults is less than t :

As the system has a test graph with $K_V$ equals t,therefore,it will still be connected(or strongly connected). There will be at least one path between every pair of vertices according to the definition of connected graphs. Therefore,for a directed test graph system,every fault free vertex can receive the diagnostic information from its fault free testees and broadcast new information to its nonfaulty testers. For a directed test graph,the set of testers and the set of testees of a vertex do not have any element in commom. Yet for an undirected test graph,the set of testers is the same as the set of testees for a vertex. There is still at least one path which consists of fault free vertices,and correct diagnostic information can be received and broadcast along this path.

(2)if the number of faults is greater than t :

According to the definition of connected graph,we encounter two situations :

(A)it is possible that the test graph is disconnected(or weakly connected),then there will be no path at all between some pairs of vertices. These vertices will fail to either collect status of other vertices or be known to other vertices of its status. Therefore,we can sort all fault free vertices into two categories : those which fail to collect status of every vertex and those which can collect status of every vertex. For those which fail,they will broadcast failure message to every other fault free vertex according to the algorithm. And since the other set consists of those which can collect status of every vertex,they will be able to collect this newly broadcast failure message. The result is no vertex in the system will ever go into a faulty state in which it thinks the system yields decisive and unanimous results. We conclude that no vertex will ever incorrectly locate faulty vertices.

(B)if the test graph is still connected(or strongly connected),then the situation is the same as that of (1).

(3)if the number of faults is equal to t :

This case is the same as (2) except the possibility of the test graph being connected is higher than that of (2). The only occassions that the test graph will be disconnected are : (a)when either all testers or all testees of a vertex are faulty,and (b)few special cases.

Q.E.D.

## 4.System Coverage,Diagnosability, Trustability and Mean Diagnosability

### 4.1 Analysis of Diagnosis Trustability

Definition 3 : Trustability of Diagnosis -- Trustability of fault diagnosis is the probability that the diagnosis system either yields correct or does not yield incorrect diagnostic results regardless the number of faults in the system.

Trustability is actually a reliability measure. A diagnosis system which guarantees locating up to k faults when the number of faults is not more than k will have diagnosis trustability of 1 when the number of faults is less than or equal to k. Yet,when the number of faults is greater than k,the trustability decreases as the number of faults increases. For example,in [2], the system will yield false results when the number of faults is greater than the assumed one. Take the 5-node completely connected graph system in Fig.1.a as an example. According to the structure of this system,diagnosis algorithm in [2] can diagnose up to 2 faults,yet,if the number of faults is greater than 2,the trustability of the diagnosis system will be the probability of any 3 or more existing faults in the system. According to the algorithm presented in this paper,the diagnosability is 1 instead of 2,which of course is 1 less than that of [2],yet the trustability of diagnosis is still 1 regardless the number of faults in the system. As we can see,the increase in diagnosis trustability is at the expense of diagnosability.

Analysis -- Trustability measures for both fail-safe and non-fail-safe diagnosis systems are analyzed below. We assume there is an exponential failure distribution for every processor in the system and the failure distributions are independent. The reliability function is : $R(t) = e^{-qt}$ . We will use R instead of R(t) just for simplicity. Equation for trustability is stated as follows :

$$T(t) = \sum_{i=0}^{n} p_i{*}C(n,i)R^{n-i}(1-R)^i \tag{1}$$

where C(n,i) is the number of combinations for choosing r faulty processors out of a total of n processors. The product term of C(n,i),R and (1-R) is the probability of exactly i faulty processors out of n processors. $p_i$ is the probability of correctly locating all i faulty processors or not incorrectly locating any faulty processor.

(1)Fail safe diagnosis system : A system with n processors can correctly locate up to k faults and doesn't incorrectly locate faults if the number of faults is greater than k. Then the trustability of diagnosis will be 1 as described below :

$$T_{fs}(t)= 1{*}R^n+1{*}C(n,1)R^{n-1}(1-R)^1+1{*}C(n,2)R^{n-2}(1-R)^2+...+1{*} C(n,k)R^{n-k}(1-R)^k + 1{*}C(n,k+1)R^{n-k-1}(1-R)^{k+1} + ... + 1{*}C(n,i)R^{n-i}(1-R)^i +... +1{*}(1-R)^n \tag{2}$$

which is the binomial expansion of $(R + (1-R))^n = 1$. The jth (j <= k) item is the probability of correctly locating all faults if there are j faults. The ith(i > k) item is the probability of not incorrectly locating any faults if there are i faults.

(2)Non-fail-safe diagnosis system : If the system can correctly locate only up to k faults,and can not guarantee the correctness of the results when the number of faults is greater than k,then the trustability will be :

$$T_{nfs}(t) = 1{*}R^n + 1{*}C(n,1)R^{n-1}(1-R)^1 + 1{*}C(n,2)R^{n-2} (1-R)^2+ ... +1{*}C(n,k)R^{n-k}(1-R)^k+p_{k+1}{*}C(n,k+1)R^{n-k-1} (1-R)^{k+1}+...+p_i{*}C(n,i)R^{n-i}(1-R)^i+... +p_n{*}(1-R)^n \tag{3}$$

where $p_i$'s are the probabilities that the diagnosis system yields correct diagnostic results if there are i faults(i > k) in the system. The probability that the system being led to incorrect state is 1 minus the trustability as follows :

$$\overline{T_{nfs}(t)} = (1-p_{k+1}){*}C(n,k+1)R^{n-k-1}(1-R)^{k+1} + ... + (1-p_n){*}(1-R)^n \tag{4}$$

### 4.2 Comparison of Trustability , Diagnosability , Coverage and Mean Diagnosability

We will compare the diagnosis system in [2] and the one proposed here. The system in [2] is the first one proposes a fully distributed diagnosis system. We call it R system and ours S system. The comparison is done on the diagnosis of a system employing multistage topology[11],which is also employed for functional language architectures[12]. An example topology is shown in Fig. 2.

(A) Trustability :

$$(1)\ \text{R system} = \sum_{i=0}^{2} 1 * C(n,i)R^{n-i}(1-R)^i + \sum_{i=3}^{n} 0 * C(n,i)R^{n-i}(1-R)^i$$

(2) S system = 1

$p_i$'s(i >=3) are 0's in R system simply because it assumes there are at most 2 existing faults in the system. If there are more than 2 faults,then,according to its diagnosis algorithm,every processor will conclude that it has already located all faulty processors whenever there are 2 1's in its fault vector even though there are still a few don't care terms. This is especially truly for those processors which fail to determine status of every processor,since they have no way of determining status of some processors. They can yield decisive results only by assuming 2 faults in the system. Trustability of S system is 1 because it determines to yield decisive and unanimous result,no guessing is ever involved.

Remember that part of trustability is just the probability of not incorrectly locating any faults. When the system can correctly locate all faults no matter how many existing faults in the system,it is equal to the probability of correctly locating the faults. Yet,when the system can locate the faults for some of the fault situations and falsely locates the faults in other fault situations,then the trustability is simply the portion in which the system can locate the faults.

(B) Diagnosability :

According to PMC model,we have :

(1) R system = 2 ; and

(2) S system = 1

Even though the diagnosability of the S system is 1,it does not imply that it can't locate 2-fault incidents at all. As the way in which diagnosability is defined,it simply means it can not locate all 2-fault incidents,although it is capable of locating most of the 2-fault incidents as we shall see in the next paragraph. Therefore,it should be coupled with coverage to give a better idea of the probability of correctly locating faults.

(C) Coverage :

Definition 4 : Coverage of Diagnosis -- The percentage of fault incidents in which diagnosis system can correctly locate all faulty processors.

Simulation Features -- Coverage measure for S system is acquired by simulation which is executed in the following manner :

(1)specify size of the system , i.e. , number of stages.

(2)specify number of faulty processing elements (PE's).

(3)for every possible combination of the number of faults , check if every fault free PE can receive fault vector broadcast by other fault free PE's.

If at least one PE is incapable of receiving broadcast fault vectors from at least one other fault free PE,then it is regarded as a failed diagnosis situation since it implies this particular PE has a fault vector with at least one element undertermined. Therefore,the whole system is incapable of coming up with a decisive and unanimous diagnostic results.

According to simulation,we have the following results :

(1) R system =

1 for all system size if # faults <= 2;
0 for all system size if # faults > 2.

(2) S system =

1.0000 for 3-stage system if # faults =1;
0.7879 for 3-stage system if # faults = 2;
0.4727 for 3-stage system if # faults = 3.

1.0000 for 4-stage system if # faults = 1;
0.9395 for 4-stage system if # faults = 2;
0.8218 for 4-stage system if # faults = 3.

The numbers shown above for 2-fault cases can be estimated quite easily as following example shows.

Example 1 -- For a 4-stage multistage interconnection network,there are three components which contribute to undiagnosable fault situations : (1)all testers of a particular PE are faulty,(2)all testees of a particular PE are faulty,and (3)special cases,as stated in the proof of Theorem 1.

For stage 0 PE's there are 2*8 such cases. As an example,in Fig. 2,assume PE(0,0) is the faulty PE addressed (0,0) with the first 0 representing its stage address and second 0 representing its address in that stage starting from top. If PE(0,1) is also faulty,then they will block the status of their two common testees in stage 1,PE(1,0)and PE(1,4),from being known to other PE's. Yet,if we assume PE(0,4) instead of PE(0,1) is faulty besides PE(0,0),then they will block their two common testers in stage 3,PE(3,0) and PE(3,1),from receiving diagnostic information. Therefore,for PE(0,0),there are 2 cases in which the diagnosis system fails. It is the same for other PE's in stage 0. Therefore,we have 2*8 cases in stage 0.

The numbers for stage 1 and 2 will still be the same. For stage 3,the number will be 1*8 only. We get 56 for cases covered by (1) and (2) situations stated at the begining of this example. Besides that,there are 4 special cases.

Therefore,there are 56+4 cases in which 2-fault situations make the diagnosis system fail. The total combinations for choosing 2 faulty PE's out of 4*8 PE's is 992. Thus,the coverage is (992-60)/992,which is 0.9395. The same estimation process can be applied to 2-fault cases for 3-stage system.

As the size of the system grows,the difference in coverage diminishes as can be observed in Fig. 3.a and 3.b. That implies S system can correctly locate almost the same number of faulty PE's when the size of system grows,as compared with those of R system,although diagnosability of S system is still 1.

As we pointed out earlier,when one considers the positive capability of a diagnosis system,he should take both coverage and diagnosability into account. We define the next measure accordingly.

Definition 5 : Mean Diagnosability -- The expected value of faults which can be correctly located regardless the number of faults in the system.

The mean diagnosability is as follows.

$$E[D] = \sum_{i=0}^{n} i * c_i \qquad (5)$$

where $c_i$ is the coverage of the diagnosis system provided there are only i faults.

(D) Mean Diagnosability :

$$\text{(1) R system} = 1 * 1 + 2 * 1 + \sum_{i=3}^{n} 3 * 0$$
$$= 3$$

$$\text{(2) S system} = 1 * 1 + 2 * 0.9395 + 3 * 0.8218 +$$
$$+ \sum_{i=4}^{n} 3 * c_i$$
$$= 5.3444 + \sum_{i=4}^{n} 3 * c_i$$

It is obvious that the mean diagnosability of S system is better than that of R system.

From the above comparisons,we acquire the following conclusions :

(1) Advantages of S system are :

(a) It either correctly locates all faulty processors or doesn't incorrectly locate any faulty processor,which is the basic requirement for a truly fault tolerant computing machine. This can be perceived from the trustability of S system. Implication of this capability is S system can afford to have a longer diagnosis period since it doesn't set an upper limit on the number of allowable faults. For R system,diagnosis period should be as short as possible in order to avoid extra faults occuring during diagnosis period.

(b) The actual number of fault incidents in which faults can be located by S system is far more better than that of R system. This is justified by coverage and mean diagnosability of S system.

(2) Disadvantage of S system is :

(a) Diagnosability is always 1 less than that of R system. Yet this is not an actual disadvantage as it is a false indicator of diagnosis system capability.

## 5. Conclusion

We have presented a technique which is suitable for fail safely, distributively,locating faults in multiprocessor systems. That the system is both competent and fail safe is justified by the measures defined in this paper in terms of mean diagnosability and trustability. As we perceived,the assumption which has been widely used in several research papers does not necessarily make diagnosis systems more competent nor reliable. On the other hand,it is always fail safe to make as few assumptions as possible. The simulation and comparison done on a multistage system proves that,by emphasizing fail safety,we improve not only reliability but also capability of the diagnosis system.

## 6. References

[1] Special Issue on Fault Tolerant Computing,Computer,Vol. 13,Mar. 1980.

[2] J.G. Kuhl and S.M. Reddy."Distributed Fault-Tolerance for Large Multiprocessor Systems",Proc. Symposium on Computer Architecture,1980,pp. 23-30.

[3] J.G. Kuhl and S.M. Reddy,"Fault Diagnosis in Fully Distributed Systems",Proc. Symposium on Fault Tolerant Computing,June 1981,pp 100-105.

[4] F. Preparata,G. Metze and R. Chien,"On the Connection Assignment Problem of Diagnosable Systems",IEEE Trans. on Computers,Vol. EC-16,Dec. 1967,pp 848-854.

[5] F Barsi,F. Grandoni and P. Maestrini,"A Theory of Diagnosability of Digital Systems",IEEE Trans. on Computers,Vol. C-25,June 1976,pp 585-593.

[6] J. Russel and C. Kime,"System Fault Diagnosis",IEEE Trans. on Computers, Vol. C-24, Nov. 1975,pp 1078-1089.

[7] G. Meyer and G. Masson,"An Efficient Fault Diagnosis Algorithm for Symmetric Multiple Processor Architectures",IEEE Trans. on Computers,Vol. C-27,Nov. 1978,pp 1059-1063.

[8] S.N. Maheshwari and S.L. Hakimi,"On Model for Diagnosable Systems and Probabilistic Fault Diagnosis",IEEE Trans. on Computers,Vol. C-25, Mar. 1976,pp 228-236.

[9] K.M. Falavarjani and D.K. Pradhan,"Fault Diagnosis of Parallel Interconnection Network",Proc. Conference on Parallel Processing,1981,pp 209-212.

[10] T.F. Arnold,"The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System",IEEE Trans. on Computers,Vol. C-22,Mar. 1973,pp 251-254.

[11] Narsingh Deo,Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Inc,1974,478 pp.

[12] C.L. Wu,T.Y. Feng,"On a Class of Multistage Interconnection Networks",IEEE Trans. on Computers,Vol. C-29,Aug. 1980,pp 694-704.

[13] F.S. Wong and M.R. Ito,"A Loop-Structured Switching Network",IEEE Trans. on Computers,Vol. C-33,May 1984,pp. 450-455.

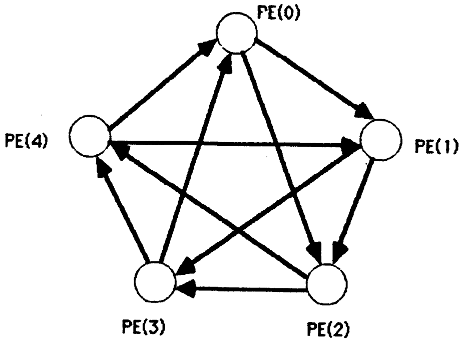


Fig. 1.a Directed Test Graph for a 5-processor System

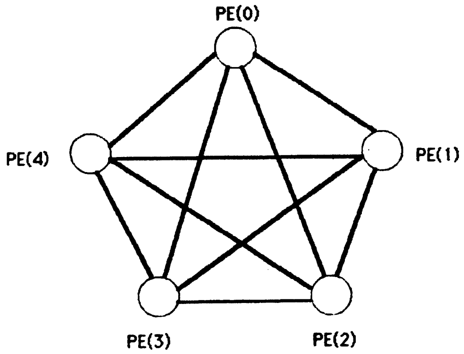
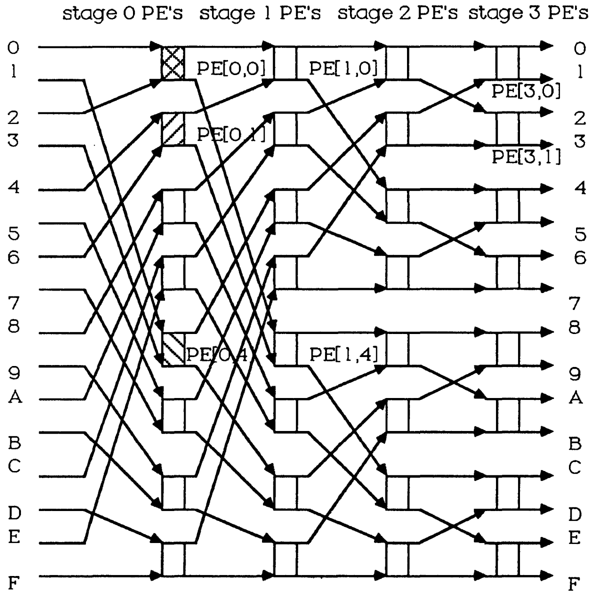


Fig. 1.b Undirected Test Graph for a 5-processor System



☒ = Faulty PE's which block the communication paths of their testers

▨= Faulty PE's which block the communication paths of their testees

Fig. 2 Two Cases in Which Diagnosis System Fails to Yield Diagnosable Results

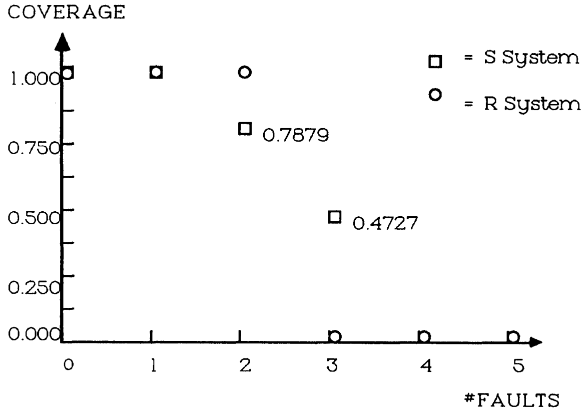


Fig. 3.a Comparison of Coverage as a Function of Number of Faults in a 3-stage MIN System

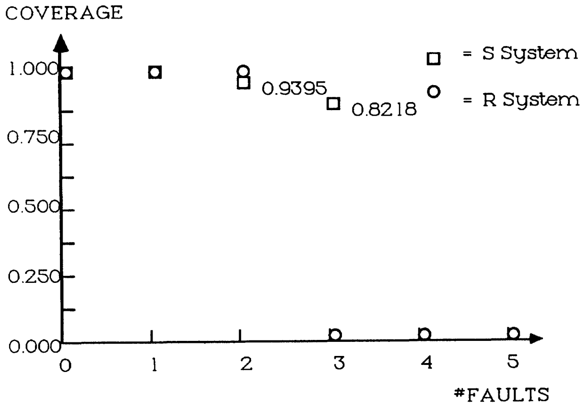


Fig. 3.b Comparison of Coverage as a Function of Number of Faults in a 4-stage MIN System

# THE IMPACT OF HARDWARE GATHER/SCATTER ON SPARSE GAUSSIAN ELIMINATION.

John G. Lewis and Horst D. Simon

Boeing Computer Services
Engineering Technology Applications
565 Andover Park West
Tukwila, WA 98188

Abstract -- Recent vector supercomputers provide vector memory access to "randomly" indexed vectors, whereas early vector supercomputers required contiguously or regularly indexed vectors. This additional capability, known as "hardware gather/scatter", can be used to great effect in general sparse Gaussian elimination. In this note we present some examples that show the impact of this change in hardware on the choice of algorithms for sparse Gaussian elimination. Common folk wisdom holds that general sparse Gaussian elimination algorithms do not perform well on vector computers. Our numerical results demonstrate that hardware gather/scatter allows general sparse elimination algorithms to outperform algorithms based on a band, envelope, or block structure on such computers.

## Gaussian Elimination on Vector Computers

Early experience with sparse Gaussian elimination on vector computers /3/ showed none of the dramatic improvements in speed-up encountered in other linear algebra computations. This is due to the fact that Gaussian elimination with a sparse data structure requires access to irregularly spaced data. Early vector computers, such as the CRAY-1 and the CYBER 205 computers, allow vector transfers to and from memory only for contiguously or regularly spaced vectors. Most sparse Gaussian elimination algorithms spend the vast majority of the factorization execution time in a loop of the following type:

```
          INTEGER I, N, M
          INTEGER INDEX(M)
          REAL A, X(M), Y(N)
               .
               .
          DO 10 I = 1, M
             Y(INDEX(I))=A*X(I) + Y(INDEX(I))
  10      CONTINUE
```

The difficulty this loop presents for vector computers is the indexing or indirect addressing for the vector Y. This loop is often referred to as an sparse or indexed SAXPY. The efficiency of the implementation of this loop determines the performance of the algorithm. Because of the importance of this loop or kernel a subroutine called SAXPYI, which follows the spirit and the notation of the BLAS /6/, has been proposed as a facility in extensions of the BLAS /2/.

The use of the SAXPYI loop as presented above, in FORTRAN, would result in no use of the vector hardware of early supercomputers. The loop would be executed using scalar instructions only, producing no speedup at all over equivalent scalar computers. The indexed SAXPY loop can be rewritten to allow some use of of the vector hardware. Dembart and Neves /1/ analyzed seven different formulations of this loop on a CDC STAR 100, and determined that there were combinations of vector length and vector density for which each of the formulations was fastest. Similar analyses by these authors for the CYBER 203 and 205 showed equivalent results, althought the ratios changed. For reasonable combinations of vector length and vector density the most important of the six alternatives to the original scalar code was the same on all three machines. This alternative is first to gather all components on which operations are to be performed in a temporary dense vector, then to perform a dense SAXPY on this short vector, and finally to scatter the results back to the appropriate memory locations.

Although this looks far more complicated than the original loop, it permits the use of the vector arithmetic units for the SAXPY loop. The non-vectorizable memory transfers are isolated to separate loops that could be made more efficient by using assembly language subroutines (albeit in scalar mode). This formulation of the indexed SAXPY is known as a GATHER-SAXPY-SCATTER (GSS) implementation, for the operations performed in turn by the three loops.

On a CRAY-1 computer, the original indexed SAXPY loop written in FORTRAN executes at a maximum rate of about 4 megaflops, much less than the maximum rate of 155 megaflops this machine can achieve for other operations. Woo and Levesque /9/ analyzed the G-S-S formulation and showed that its maximum rate in assembly language was around 8 megaflops. Alternatively, a good assembly language implementation of the original loop using only scalar hardware, performs asymptotically at 13 megaflops (see /8/). This implementation is never slower than the G-S-S formulation, demonstrating that this is essentially a scalar computation for the CRAY-1.

In contrast, the standard implementation of banded or variable banded Gaussian elimination algorithms use dense dot products or dense SAXPY operations as their fundamental inner-loops. Such implementations can achieve vector speeds on vector computers. However, they usually do not approach the asymptotic speeds of these machines because the vector lengths are limited by the bandwidth, which should not become very large. The possibility of using the vector hardware for these schemes and the inherent performance limitation of the indexed SAXPY loop has led to the conventional folk wisdom that (variable) banded factorization schemes will usually outperform general sparse Gaussian elimination on vector computers.

The vector supercomputers being produced currently,

0190-3918/86/0000/0366 $01.00 © 1986 IEEE

in particular the CRAY X-MP/4 and the more recent models of the CRAY X-MP/2, are equipped with new facilities that permit memory access according to an index vector in hardware. That is, these machines permit the GATHER and SCATTER loops to be performed using vector memory transfers to and from a hardware vector register. This gather/scatter hardware leads to a much faster implementation of SAXPYI, using the G-S-S formulation. The assembly language coded implementation in VectorPak reaches 78 megaflops asymptotically. Some detailed SAXPYI timings are given in Table 1 below /8/:

**Table 1.** SAXPYI Speed on the CRAY X-MP/24 (1 CPU, rates given in megaflops).

| | M = 10 | M = infinity |
|---|---|---|
| (ignoring the hardware for gather/scatter) | | |
| CFT 1.13 | 5.0 | 5.7 |
| VectorPak | 6.3 | 14.5 |
| (using the hardware gather/scatter) | | |
| CFT 1.14 | 16.1 | 54.6 |
| VectorPak | 16.1 | 78.6 |

The older CRAY Fortran Compiler CFT 1.13 does not make use of the hardware for gather/scatter. A corresponding VectorPak implementation of SAXPYI has been developed for CRAY X-MP's without hardware gather/scatter. Both exhibit the scalar performance characteristic of the CRAY-1. In contrast the utilization of hardware gather/scatter either via CFT 1.14 or with VectorPak shows a dramatic improvement.

Numerical Results.

Our first series of numerical results demonstrates the speed-up which can be obtained using hardware/gather in a general sparse elimination algorithm on very large real life applications. We use a modified minimum degree (MD) algorithm by Liu /7/ to solve the following seven problems taken from the sparse matrix collection /4/. Below a short problem description and some ordering statistics are given. All problems with the exception of LRGPWR are finite element models of large three dimensional structures. All matrices are symmetric and positive definite.

**Table 2.** Problem Description.

| Problem | Description |
|---|---|
| STK3562 | Finite element model of Sports Arena<br>N=3562, $A_{nz}$=78174, $L_{nz}$=275360 |
| STK3948 | Finite element model of oil platform<br>N=3948, $A_{nz}$=56934, $L_{nz}$=647274 |
| STK4884 | Corps of Engineers model of dam<br>N=4884, $A_{nz}$=142747, $L^{nz=736294}$ |
| LRGPWR | Electric power network of U.S.<br>N=5300, $A_{nz}$=8271, $L_{nz}$=22764 |
| ST10974 | Elevated pressure vessel<br>N=10974, $A_{nz}$208838, $L_{nz}$=994885 |
| ST11948 | Nuclear power station<br>N=11948, $A_{nz}$=68571, $L_{nz}$=650777 |
| ST15439 | 76 story skyscraper<br>N=15439, $A_{nz}$=118401, $L_{nz}$=1401129 |

where,
N order of the matrix,
$A_{nz}$ nonzeros in lower triangle of the original matrix,
$L_{nz}$ nonzeros in the Cholesky factor of A,

The numbers in Table 2 show that with the exception of the power network problem LRGPWR all matrices considered have a substantial number of nonzeros per row. Thus we expect to see the theoretical speedups due to hardware gather/scatter realized on some practical applications.

All problems were solved using four different implementations of the key SAXPYI loop. The original source code was modified by inserting a few compiler directives to affect the vectorization of the key loops. Even though CFT1.14 can vectorize the access of random elements according to an index vector as in the SAXPYI-loop, a compiler directive must be inserted in the original code, in order to instruct the compiler to do so. The Fortran code with inserted compiler directives was then compiled twice under CFT1.14. Using a compiler option, code was generated both for a Cray X-MP with and without hardware gather/scatter.

Then the code was modified to call an optimized implementation of SAXPYI from VectorPak /8/. This modifed code was also compiled under CFT1.14, and then executed twice. First a VectorPak library for Cray X-MP's without gather/scatter was used, and then the corresponding library for machines with gather/scatter. Table 3 and 4 below present the execution times obtained for factorization and solution for the four implementations. All execution times are listed relative to the VectorPak with gather/scatter time, which is normalized to one. The actual execution time in seconds for the VectorPak with gather/scatter implementation is given in Table 5, together with the execution times for an envelope factorization based on the reverse Cuthill-McKee (RCM) algorithm from /5/.

Tables 3 and 4 show clearly the direct benefits of the hardware gather/scatter feature for general sparse elimination schemes. Using this feature factorization and solution are in some cases almost up to 8 times faster. Very sparse problems such as LRGPWR, however, do not benefit from this speedup, since the number of nonzeros per row is too small. Note that even the factored matrix LRGPWR has only about 9 nonzeros per row, too few to obtain the fast asymptotic rates in Table 1. On the other hand, the finite element problems do have a large number of nonzeros per row, and thus almost realize the speedup potential of the asymptotic rates given in Table 1.

The numbers for VectorPak demonstrate two facts. Obviously it pays more in terms of speedup to use a carefully assembly coded subroutine on a machine without hardware gather/scatter. And in spite of advances in the CRAY compilers, VectorPak still offers a 20-25% improvement in the factorization and a 30% improvement in the solution as compared to straight forward CFT1.14.

**Table 3.** Execution Times for Sparse Matrix Factorization.
(normalized so that VectorPak with g/s = 1.00)

| Problem | CFT1.14<br>no g/s | CFT1.14<br>with g/s | VectorPak<br>no g/s |
|---|---|---|---|
| STK3562 | 6.33 | 1.14 | 2.54 |
| STK3948 | 9.66 | 1.24 | 3.25 |
| STK4884 | 8.70 | 1.20 | 3.14 |
| LRGPWR | 1.25 | 0.93 | 1.17 |
| ST10974 | 7.24 | 1.16 | 2.80 |
| ST11948 | 8.75 | 1.22 | 3.12 |
| ST15439 | 8.78 | 1.25 | 3.15 |

**Table 4.** Execution Times for Sparse Matrix Solution.

| Problem | CFT1.14<br>no g/s | CFT1.14<br>with g/s | VectorPak<br>no g/s |
|---|---|---|---|
| STK3562 | 6.94 | 1.48 | 2.55 |
| STK3948 | 9.22 | 1.47 | 3.13 |
| STK4884 | 8.77 | 1.39 | 3.00 |
| LRGPWR | 1.96 | 1.57 | 1.11 |
| ST10974 | 7.45 | 1.48 | 2.68 |
| ST11948 | 5.94 | 1.50 | 2.23 |
| ST15439 | 7.54 | 1.49 | 2.68 |

**Table 5.** Execution Times (sec) for Factorization and Solution Routines.

| Problem | Factorization<br>RCM | <br>MD | Solution<br>RCM | <br>MD |
|---|---|---|---|---|
| STK3562 | 3.421 | 1.276 | 0.047 | 0.033 |
| STK3948 | 7.487 | 4.074 | 0.064 | 0.055 |
| STK4884 | 4.071 | 4.129 | 0.065 | 0.066 |
| LRGPWR | 3.642 | 0.102 | 0.061 | 0.028 |
| ST10974 | - | 4.891 | - | 0.108 |
| ST11948 | - | 3.871 | - | 0.094 |
| ST15439 | 17.440 | 7.791 | 0.213 | 0.152 |

The comparison in Table 5 shows that general sparse methods outperform envelope solvers on vector computers with hardware gather/scatter. Because of the natural vectorization of envelope methods and the essentially scalar performance of general sparse methods on earlier vector computers, general sparse methods were generally thought of non-competitive on vector computers. Table 5 disproves this assertion, in particular since some of the problems could not even be solved by RCM within the 4Mword available on the Cray X-MP/24 at Boeing Computer Services.

A final point worth making concerns a software issue. Sparse Gaussian elilimination software from /5/ had been modified with calls to computational kernels replacing some inner loops several years ago, when it was installed on the CRAY-1S. As soon as the CRAY X-MP arrived some of the above test examples were run again without any code modifications. By using computational kernels from a kernel library such as VectorPak the application programmer therefore can reap directly the benefits of hardware improvements, without concerning himself with the often subtle details of a new implementation, or without waiting for the compiler writer to catch up with the hardware architect. Thus the success in using hardware gather/scatter in the context of sparse Gaussian elimination also validates the concept of computational kernels as a tool for combining portability and optimality on advanced architectures.

## References.

/1/ B. Dembart and K. Neves, "Sparse Triangular Factorization on Vector Computers", in Exploring Applications of Parallel Processing, Electric Power Research Institute, EL-566-QR, Palo Alto, California (1977), pp. 22-25.

/2/ D.S. Dodson and J.G. Lewis, "Proposed Sparse Extensions to the Basic Linear Algebra Subprograms", SIGNUM Newsletter, Vol. 20, (1985), pp. 22-25.

/3/ I.S. Duff, "The Solution of Sparse Linear Equations on the CRAY-1.", CRAY Channels, Vol. 4, (1982), pp. 4-9.

/4/ I.S. Duff, R. Grimes, J. Lewis, and W. Poole, "Sparse Matrix Test Problems", SIGNUM Newsletter, (1982), p. 22.

/5/ A. George and J. Liu, Computer Solution of Large Sparse Positive Definite Linear Systems, Prentice Hall, Englewood Cliffs, (1982).

/6/ C. Lawson, R.Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", ACM Trans. on Mathematical Software, Vol. 5, (1978), pp. 308-323.

/7/ J.W.H. Liu, "Modification of the Minimum Degree Algorithm by Multiple Elimination", ACM Trans. on Mathematical Software, Vol. 11, (1985), pp. 141-153.

/8/ VectorPak Subroutine Library User's Manual, Document 20460-0501, Boeing Computer Services, (1986).

/9/ P.T. Woo and J.M. Leveque, "Benchmarking a Sparse Elimination Routine on the Cyber 205 and the CRAY-1", Proceedings of the Sixth SPE Symposium on Reservoir Simulation, (1982).

# AN ALGORITHM FOR SOLVING SPARSE SETS OF LINEAR EQUATIONS WITH AN ALMOST TRI-DIAGONAL STRUCTURE ON SIMD COMPUTERS[a]

Torstein Opsahl

Science Applications Research
Lanham, Maryland 20706

Dennis Parkinson

DAP Support Unit
Queen Mary College
University of London
London, England

## Abstract

In this paper, an algorithm for solving sparse sets of equations with an almost tri-diagonal structure on Single-Instruction-Multiple-Data Stream (SIMD) computers, like the ICL Distributed Array Processor (DAP) and the Goodyear Aerospace Massively Parallel Processor (MPP), is described. Central to this algorithm is the partitioning of the original sparse set of equations into a number of smaller tri-diagonal sets that can be solved in parallel using cyclic reduction. A DAP implementation of the algorithm, which demonstrates how the nearest neighbor connection network can be used to keep track of the couplings between unknowns, as these (the couplings) are modified by the solution procedure, is detailed. The paper concludes with a discussion of the algorithm's performance on an actual application.

## 1. Introduction

The numerical solution of one-dimensional differential equations often requires the inversion of matrices with a sparsity structure that we will refer to as almost tri-diagonal. Such problems can, for instance, be found in the field of nuclear reactor hydrodynamics and other branches of fluid dynamics. In this paper, we describe an algorithm for solving almost tri-diagonal problems on Single-Instruction-Multiple-Data Stream (SIMD) computers like the ICL Distributed Array Processor (DAP) and the Goodyear Aerospace Massively Parallel Processor (MPP).

These machines derive their high performance from the lockstep operation of a large number (4,096 and 16,384 in the case of the DAP and MPP, respectively) of simple, one-bit Processing Elements (PEs), rather than from a few, powerful processors as in the case of vector computers such as the CRAY-XMP and CYBER 205. Thus, algorithms cannot exploit the full processing power of the DAP and MPP unless they allow most, or all, of the PEs to be engaged in productive work[b] most, or all, of the time. Parallel cyclic reduction, which is a standard technique for solving single tri-diagonal sets of equations, satisfies this criterion. In our algorithm for inverting almost tri-diagonal problems, we take advantage of this by partitioning the original sparse matrix into a number of smaller matrices with tri-diagonal structures. As will be shown (see Section 2.2.2), these can all be inverted in parallel using cyclic reduction, provided the total number of equations in the almost tri-diagonal set does not exceed the number of PEs of the computer.

The full algorithm for solving almost tri-diagonal problems has been successfully implemented on the DAP. Following the description of the algorithm, details of this implementation are given. The paper concludes with a discussion of the algorithm's performance on an actual application.

## 2. An Algorithm for Solving Almost Tri-Diagonal Sets of Equations

Consider undirected graphs or networks of N nodes that satisfy the following requirements (see Figure 1):

(i) The majority of nodes [O(N)] have only two neighbors.

(ii) No nodes have more than three neighbors, and there are at most m such nodes ($0 \leq m << N$); these will be referred to as junctions.

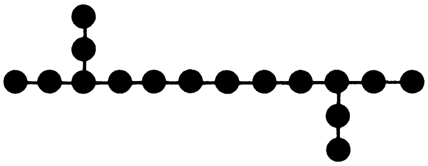


Figure 1. Sample Network

Networks like these are frequently used to numerically solve one-dimensional differential equations.

[a] This work was supported by a grant from the United Kingdom Atomic Energy Establishment, Winfrith.

[b] We here define productive work as an operation that contributes toward completing the task of an algorithm.

In that case, some quantity x is to be computed at each node. Typically, this involves solving a set of simultaneous, linear equations which, given conditions (i) and (ii), can be expressed by:

$$a_i x_r + b_i x_i + c_i x_\ell + e_i x_b = d_i \qquad (1)$$

where

$r, \ell, b, i = 1..N$

$$\begin{array}{l} r = i-1 \\ \\ \ell = i+1 \end{array} \quad ; \; j_k + 1 < i < j_{k'}-1 \qquad (2)$$

$j_k, j_{k'}$ = numbers of junctions; $k, k' = 1..m$

From (2) it follows that for each sub-graph or sub-network, $j_k+1 \le i \le j_{k'}-1$, (1) may be rewritten as three equations:

$$\begin{array}{lll} a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i ; & j_k+1<i<j_{k'}-1 & \\ a_i x_{j_k} + b_i x_i + c_i x_{i+1} = d_i ; & i = j_k+1 & (3) \\ a_i x_{i-1} + b_i x_i + c_i x_{j_{k'}} = d_i ; & i = j_{k'}-1 & \end{array}$$

Furthermore, if $a_i x_{j_k}$ and $c_i x_{j_{k'}}$ are treated as right-hand-side terms, (3) can be compressed to a single equation:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i + \alpha_i x_{j_k} + \beta_i x_{j_{k'}} \qquad (4)$$

where

$$\begin{array}{ll} a_i x_{i-1} = 0; & i = j_k+1 \\ c_i x_{i+1} = 0; & i = j_{k'}-1 \\ \alpha_i = 0; & j_k+1<i\le j_{k'}-1 \\ \alpha_i = -a_i; & i = j_k+1 \\ \beta_i = 0; & j_k+1\le i<j_{k'}-1 \\ \beta_i = -c_i; & i = j_{k'}-1 \end{array}$$

It can be seen that (4) is of tri-diagonal form although it has three right-hand-side terms. The m junctions separate the original network (graph) into S [$O(2^m)$] sub-networks. Hence, (1) can be partitioned into S tri-diagonal sets of equations like (4). These can be solved separately to yield $x_i (i \ne j_1 ... j_m)$ as a linear function of $x_{j_k}$ and $x_{j_{k'}}$ :[c]

$$x_i = p_i + q_i x_{j_k} + r_i x_{j_{k'}} \qquad (5)$$

where

$k, k' = 1..m$
$i = 1..N \; (i \ne j_1 ... j_m)$

To complete the solution of the original sparse set of equations, (1), the m equations for the junctions ($e_i \ne 0$; $i = j_1 ... j_m$) must now be used. By substituting the expressions for $x_r$, $x_\ell$, and $x_b$ obtained from (5) in these junction equations, a closed matrix problem (size mxm) is created, which can easily be solved to yield values for $x_i$, $i = j_1 ... j_m$. Back-substitution in (5) then gives the values for the remaining N-m unknowns.

An additional step is needed in the above algorithm when the network (graph) represents a closed loop. Unless the nodes can be numbered so that the loop closes back on itself ("wraps around") at a junction--this is, of course, impossible if a network does not contain junctions--(5) will have a slightly different form for the two sub-networks, s' and s", that close the loop:

$$x_i = p_i + q_i x_{s''_n} + r_i x_{j_{k'}} ; \qquad s'_1 \le i \le s'_n \qquad (6)$$

$$x_i = p_i + q_i x_{j_k} + r_i x_{s'_1} ; \qquad s''_1 \le i \le s''_n \qquad (7)$$

where

$s'_1$ = lowest numbered node of sub-network s'

$s''_1$ = lowest numbered node of sub-network s"

$s'_n$ = highest numbered node of sub-network s'

$s''_n$ = highest numbered node of sub-network s"

Before proceeding to substitute in the equations for the m junctions, the terms $q_i x_{s''_n}$ and $r_i x_{s'_1}$ must be eliminated. It will be seen that this is easily accomplished using the equations obtained by writing down (6) and (7) for $i=s'_1$ and $i=s''_n$, respectively.

A fundamental problem in implementing the algorithm described in this section on SIMD computers like the DAP and MPP is how to keep track of the $(x_{j_k}, x_{j_{k'}})$-pair to which each $x_i$ is coupled. The method for accomplishing this on the DAP is described in Section 2.3.3. A brief overview of this machine is now given.

[c] Obviously, the pair (k, k') is different for each sub-network.

### 2.1 Brief Overview of the DAP[d] and its High-Level Language (HOL)

The ICL DAP is an SIMD computer consisting of $64^2$ (4096) one-bit Processing Elements (PEs)--all under control of a Master Control Unit (MCU). Communication lines between PEs are provided by a nearest-neighbor connection network, and each PE is equipped with 16 kbits of memory for a total storage capacity of 8 Mbytes. All of this can be used as a memory module of the DAP's front-end, an ICL 2980 mainframe computer.

The DAP can operate in two different parallel processing modes. The most powerful of these performs sequential operations on bit slices from 4096 different memory words and is referred to as matrix mode. In the less powerful vector mode processing, the PEs are made to carry out operations on 64 words simultaneously with all bits of each word processed in parallel.

A special, non-portable HOL, called DAP FORTRAN, has been developed for the DAP. It differs from ordinary FORTRAN mainly in that it has facilities for expressing the hardware parallelism of the DAP. More specifically, DAP FORTRAN allows an operation to be specified for all elements of a vector or matrix simultaneously. There is, however, one proviso: the dimensions of a vector and matrix declared in DAP FORTRAN must match those of the PE array; thus, a vector can only have 64 elements and a matrix 4096, arranged as a two-dimensional $64^2$ array. A matrix can also be reduced to a one-dimensional structure by treating it as a "long" vector consisting of 4096 linearly arranged elements. Operations on data arrays with more than two dimensions must be broken up into sets of either vector or matrix operations depending on the array declarations.

To complement the parallel data structures, the indexing facilities of ordinary FORTRAN have been extended in DAP FORTRAN. Most notably, logical expressions, or masks, can be used to select data items from vectors and matrices.

### 2.2 DAP Implementation Details of the Algorithm for Solving Almost Tri-Diagonal Sets of Equations

2.2.1 Memory Mapping. For algorithms to efficiently exploit the processing power of the DAP, they must engage on the order of 4,096 PEs simultaneously in productive work. It is demonstrated in the following that the algorithm described in Section 2 can achieve this, provided the networks (graphs) are mapped onto the PE array so that all information associated with node i is stored in the local memory of PE i. This information consists of the coefficients for equation i and the numbers of the nodes to which this equation couples node i. Without the latter, neither the substitutions into the m junction equations nor the back-substitution step can be performed.

2.2.2 Extending the Use of Parallel Cyclic Reduction to Multiple Sets of Tri-Diagonal Equations. Parallel cyclic reduction is a standard technique for solving tri-diagonal equations on SIMD machines like the DAP and the MPP. Formally, for a single system of equations, $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = k_i$, the technique is expressed by the following recursive formulae:

Left-Hand Side

$$a_i^{(\ell)} = -a_i^{(\ell-1)} a_{i-2^{(\ell-1)}}^{(\ell-1)} / b_{i-2^{(\ell-1)}}^{(\ell-1)} \qquad (8)$$

$$c_i^{(\ell)} = -c_i^{(\ell-1)} c_{i+2^{(\ell-1)}}^{(\ell-1)} / b_{i+2^{(\ell-1)}}^{(\ell-1)} \qquad (9)$$

$$b_i^{(\ell)} = b_i^{(\ell-1)} - a_i^{(\ell-1)} c_{i-2^{(\ell-1)}}^{(\ell-1)} / b_{i-2^{(\ell-1)}}^{(\ell-1)} - c_i^{(\ell-1)} a_{i+2^{(\ell-1)}}^{(\ell-1)} / b_{i+2^{(\ell-1)}}^{(\ell-1)} \qquad (10)$$

Right-Hand Side

$$k_i^{(\ell)} = k_i^{(\ell-1)} - a_i^{(\ell-1)} k_{i-2^{(\ell-1)}}^{(\ell-1)} / b_{i-2^{(\ell-1)}}^{(\ell-1)} - c_i^{(\ell-1)} k_{i+2^{(\ell-1)}}^{(\ell-1)} / b_{i+2^{(\ell-1)}}^{(\ell-1)} \qquad (11)$$

where

$\ell$ = step number

$a_i^{(\ell-1)} = 0;\ i \leq 2^{(\ell-1)}$

$c_i^{(\ell-1)} = 0;\ i \geq n-2^{(\ell-1)}$

After $\log_2 n$ steps, where n is the number of equations in a set, all coefficients $a_i$ and $c_i$ will have been eliminated, and the values of the unknowns are given by (11). It will now be shown how parallel cyclic reduction can be used to solve S sets of tri-diagonal equations like (4) simultaneously on the DAP.

[d] The description is confined to the installation at Queen Mary College (University of London), and the reader should be aware that there are DAP systems that differ from this one in a number of respects.

By applying (11) to the right-hand-side terms of equation (4), it follows that:

$$\alpha_i^{(\ell)} = \alpha_i^{(\ell-1)}; \qquad j_k \leq i < j_k + 2^{(\ell-1)} \tag{12}$$

$$\alpha_i^{(\ell)} = -a_i^{(\ell-1)} \alpha^{(\ell-1)}_{i-2^{(\ell-1)}} / b^{(\ell-1)}_{i-2^{(\ell-1)}}; \; j_{k'} \geq i \geq j_k + 2^{(\ell-1)}$$

$$\beta_i^{(\ell)} = \beta_i^{(\ell-1)}; \qquad j_{k'} - 2^{(\ell-1)} < j \leq j_{k'} \tag{13}$$

$$\beta_i^{(\ell)} = -c_i^{(\ell-1)} \beta^{(\ell-1)}_{i+2^{(\ell-1)}} / b^{(\ell-1)}_{i+2^{(\ell-1)}}; \; j_k \leq i \leq j_{k'} - 2^{(\ell-1)}$$

Comparing equations (12) and (13) with equations (8) and (9), respectively, we observe that:

(A) $a_i^{(\ell)} \neq 0$ for those values of i that $\alpha_i^{(\ell)} = 0$.

(B) The recursive formula for $\alpha_i^{(\ell)}$ when $j_{k'} \geq i \geq j_k + 2^{(\ell-1)}$ is identical to equation (8).

(C) $c_i^{(\ell)} \neq 0$ for those values of i that $\beta_i^{(\ell)} = 0$.

(D) The recursive formula for $\beta_i^{(\ell)}$ when $i_k \leq i \leq i_{k'} - 2^{(\ell-1)}$ is identical to equation (10).

(A) and (C) show that the coefficients $a_i^{(\ell)} \neq 0$ and $\alpha_i^{(\ell)} \neq 0$ and $c_i^{(\ell)} \neq 0$ and $\beta_i^{(\ell)} \neq 0$, respectively, can be merged into two vectors, each of length N. If these are declared as "long" vectors in DAP FORTRAN--in keeping with the previously specified memory mapping--it follows from (B) and (C) that their elements, and hence all S sets of equations, can be operated on in parallel on the DAP as long as $N \leq 4,096$. Furthermore, no extra arithmetic is required to compute $\alpha_i^{(\ell)}$ and $\beta_i^{(\ell)}$ compared with that which would be needed to solve a tri-diagonal problem with a single right-hand-side term.

2.3.3 Details of the DAP FORTRAN Code. Figure 2 is a DAP FORTRAN listing of the main subprogram in the implementation of our algorithm for solving almost tri-diagonal problems. SUBROUTINE NET_TRID_SOLVE first inverts S sets of equations of form (4) using parallel cyclic reduction as described in the preceding section. The vectors whose elements are $a_i^{(\ell)} \neq 0$ and $\alpha_i^{(\ell)} \neq 0$ and $c_i^{(\ell)} \neq 0$ and $\beta_i^{(\ell)} \neq 0$, respectively, correspond to LDC and UDC of Figure 2. To perform left-hand-side indexing operations on these vectors, three Boolean "long" vectors, or masks, MSK_JCT, RIGHT, and LEFT are used. Their effect is to disable writing into memory in those PEs whose logical positions in the PE array correspond to the positions of elements with value FALSE in the masks. For instance, the statement LDC(RIGHT) = -LDC... computes the coefficients $a_i^{(\ell)} \neq 0$ and $\alpha_i^{(\ell)} \neq 0$ with RIGHT inhibiting overwriting of the elements $\alpha_i^{(\ell-1)} \neq 0$: these are needed in the back-substitution step of the algorithm for solving sparse problems of form (1). The coefficients $\beta_i^{(\ell-1)} \neq 0$ are also needed in this step. Hence, in the statement UDC(LEFT) = -UDC..., which calculates $c_i^{(\ell)} \neq 0$ and $\beta_i^{(\ell)} \neq 0$, LEFT inhibits overwriting of these coefficients. Finally, MSK_JCT disables processing in those PEs that hold the coefficients of the m junction equations.[e] Thus, the parallelism achieved during the cyclic reduction of S sets of equations is (N-m)[O(N)]. After $n = O[\log_2 \max(j_{k'} - j_k + 1)]$ steps, all solutions of form (5) have been obtained, and the first stage of the algorithm is complete.

Now, the number of elements $\alpha_i$ and $\beta_i$ that have values different from zero doubles at each step in the cyclic reduction phase. Thus, at step $\ell$, $x_i$ is coupled to $x_{j_k}$ and $x_{j_{k'}}$ for $j_k + 1 \leq i < j_k + 2^{\ell}$ and $j_{k'} - 2^{\ell} < i \leq j_{k'} - 1$, respectively. If the values of $j_k$ and $j_{k'}$ are initially stored in the elements $(j_k+1)$ and $(j_{k'}-1)$, respectively, of two separate "long" vectors, rc and lc, then the elements of these vectors must be updated according to:

$$rc_i^{(\ell)} = j_k; \qquad j_k + 1 \leq i < j_k + 1 + 2^{\ell} \tag{14}$$

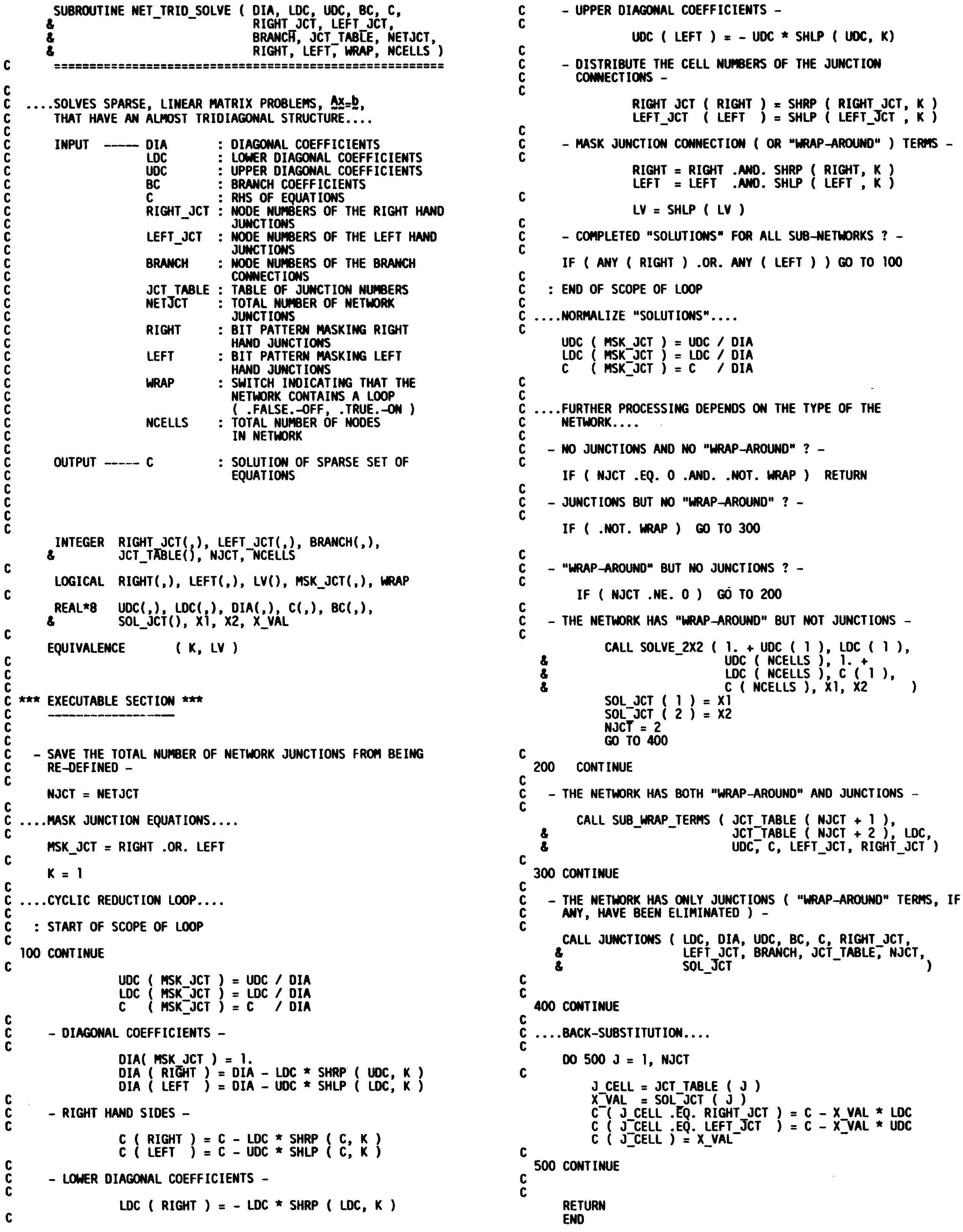
$$lc_i^{(\ell)} = j_{k'}; \qquad j_{k'} - 1 \geq i > j_{k'} - 1 - 2^{\ell} \tag{15}$$

where

k, k' = 1..m;
i = 1..N;

In Figure 2, RIGHT_JCT and LEFT_JCT correspond to rc and lc, respectively. The modification of the information stored in these "long" vectors, as specified by equations (14) and (15), is performed by using DAP FORTRAN built-in shift functions: SHRP propagates data from a PE to its "right" neighbor, and SHLP performs the same task in the "left" direction.

[e] The reader is reminded that these equations are set aside during the first stage of the algorithm described in Section 2.



```
      SUBROUTINE NET_TRID_SOLVE ( DIA, LDC, UDC, BC, C,
     &                            RIGHT_JCT, LEFT_JCT,
     &                            BRANCH, JCT_TABLE, NETJCT,
     &                            RIGHT, LEFT, WRAP, NCELLS )
C     ======================================================
C
C
C ....SOLVES SPARSE, LINEAR MATRIX PROBLEMS, Ax=b,
C     THAT HAVE AN ALMOST TRIDIAGONAL STRUCTURE....
C
C     INPUT  ----- DIA       : DIAGONAL COEFFICIENTS
C                  LDC       : LOWER DIAGONAL COEFFICIENTS
C                  UDC       : UPPER DIAGONAL COEFFICIENTS
C                  BC        : BRANCH COEFFICIENTS
C                  C         : RHS OF EQUATIONS
C                  RIGHT_JCT : NODE NUMBERS OF THE RIGHT HAND
C                              JUNCTIONS
C                  LEFT_JCT  : NODE NUMBERS OF THE LEFT HAND
C                              JUNCTIONS
C                  BRANCH    : NODE NUMBERS OF THE BRANCH
C                              CONNECTIONS
C                  JCT_TABLE : TABLE OF JUNCTION NUMBERS
C                  NETJCT    : TOTAL NUMBER OF NETWORK
C                              JUNCTIONS
C                  RIGHT     : BIT PATTERN MASKING RIGHT
C                              HAND JUNCTIONS
C                  LEFT      : BIT PATTERN MASKING LEFT
C                              HAND JUNCTIONS
C                  WRAP      : SWITCH INDICATING THAT THE
C                              NETWORK CONTAINS A LOOP
C                              ( .FALSE.-OFF, .TRUE.-ON )
C                  NCELLS    : TOTAL NUMBER OF NODES
C                              IN NETWORK
C
C     OUTPUT ----- C         : SOLUTION OF SPARSE SET OF
C                              EQUATIONS
C
C
C
C
      INTEGER  RIGHT_JCT(,), LEFT_JCT(,), BRANCH(,),
     &         JCT_TABLE(), NJCT, NCELLS
C
      LOGICAL  RIGHT(,), LEFT(,), LV(), MSK_JCT(,), WRAP
C
      REAL*8   UDC(,), LDC(,), DIA(,), C(,), BC(,),
     &         SOL_JCT(), X1, X2, X_VAL
C
      EQUIVALENCE       ( K, LV )
C
C
C
C *** EXECUTABLE SECTION ***
C     ------------------
C
C
C   - SAVE THE TOTAL NUMBER OF NETWORK JUNCTIONS FROM BEING
C     RE-DEFINED -
C
      NJCT = NETJCT
C
C ....MASK JUNCTION EQUATIONS....
C
      MSK_JCT = RIGHT .OR. LEFT
C
      K = 1
C
C ....CYCLIC REDUCTION LOOP....
C
C   : START OF SCOPE OF LOOP
C
  100 CONTINUE
C
              UDC ( MSK_JCT ) = UDC / DIA
              LDC ( MSK_JCT ) = LDC / DIA
              C   ( MSK_JCT ) = C   / DIA
C
C     - DIAGONAL COEFFICIENTS -
C
              DIA( MSK_JCT ) = 1.
              DIA ( RIGHT ) = DIA - LDC * SHRP ( UDC, K )
              DIA ( LEFT  ) = DIA - UDC * SHLP ( LDC, K )
C
C     - RIGHT HAND SIDES -
C
              C ( RIGHT ) = C - LDC * SHRP ( C, K )
              C ( LEFT  ) = C - UDC * SHLP ( C, K )
C
C     - LOWER DIAGONAL COEFFICIENTS -
C
              LDC ( RIGHT ) = - LDC * SHRP ( LDC, K )
C
```

```
C     - UPPER DIAGONAL COEFFICIENTS -
C
              UDC ( LEFT ) = - UDC * SHLP ( UDC, K)
C
C     - DISTRIBUTE THE CELL NUMBERS OF THE JUNCTION
C       CONNECTIONS -
C
              RIGHT_JCT ( RIGHT ) = SHRP ( RIGHT_JCT, K )
              LEFT_JCT  ( LEFT  ) = SHLP ( LEFT_JCT , K )
C
C     - MASK JUNCTION CONNECTION ( OR "WRAP-AROUND" ) TERMS -
C
              RIGHT = RIGHT .AND. SHRP ( RIGHT, K )
              LEFT  = LEFT  .AND. SHLP ( LEFT , K )
C
              LV = SHLP ( LV )
C
C     - COMPLETED "SOLUTIONS" FOR ALL SUB-NETWORKS ? -
C
      IF ( ANY ( RIGHT ) .OR. ANY ( LEFT ) ) GO TO 100
C
C   : END OF SCOPE OF LOOP
C
C ....NORMALIZE "SOLUTIONS"....
C
      UDC ( MSK_JCT ) = UDC / DIA
      LDC ( MSK_JCT ) = LDC / DIA
      C   ( MSK_JCT ) = C   / DIA
C
C
C ....FURTHER PROCESSING DEPENDS ON THE TYPE OF THE
C     NETWORK....
C
C   - NO JUNCTIONS AND NO "WRAP-AROUND" ? -
C
      IF ( NJCT .EQ. 0 .AND. .NOT. WRAP )  RETURN
C
C   - JUNCTIONS BUT NO "WRAP-AROUND" ? -
C
      IF ( .NOT. WRAP )  GO TO 300
C
C   - "WRAP-AROUND" BUT NO JUNCTIONS ? -
C
        IF ( NJCT .NE. 0 )  GO TO 200
C
C   - THE NETWORK HAS "WRAP-AROUND" BUT NOT JUNCTIONS -
C
            CALL SOLVE_2X2 ( 1. + UDC ( 1 ), LDC ( 1 ),
     &                       UDC ( NCELLS ), 1. +
     &                       LDC ( NCELLS ), C ( 1 ),
     &                       C ( NCELLS ), X1, X2      )
            SOL_JCT ( 1 ) = X1
            SOL_JCT ( 2 ) = X2
            NJCT = 2
            GO TO 400
C
  200 CONTINUE
C
C   - THE NETWORK HAS BOTH "WRAP-AROUND" AND JUNCTIONS -
C
        CALL SUB_WRAP_TERMS ( JCT_TABLE ( NJCT + 1 ),
     &                        JCT_TABLE ( NJCT + 2 ), LDC,
     &                        UDC, C, LEFT_JCT, RIGHT_JCT )
C
  300 CONTINUE
C
C   - THE NETWORK HAS ONLY JUNCTIONS ( "WRAP-AROUND" TERMS, IF
C     ANY, HAVE BEEN ELIMINATED ) -
C
      CALL JUNCTIONS ( LDC, DIA, UDC, BC, C, RIGHT_JCT,
     &                 LEFT_JCT, BRANCH, JCT_TABLE, NJCT,
     &                 SOL_JCT                            )
C
C
  400 CONTINUE
C
C ....BACK-SUBSTITUTION....
C
      DO 500 J = 1, NJCT
C
         J_CELL = JCT_TABLE ( J )
         X_VAL  = SOL_JCT ( J )
         C ( J_CELL .EQ. RIGHT_JCT ) = C - X_VAL * LDC
         C ( J_CELL .EQ. LEFT_JCT  ) = C - X_VAL * UDC
         C ( J_CELL ) = X_VAL
C
  500 CONTINUE
C
C
      RETURN
      END
```

Figure 2. DAP FORTRAN Listing

At the end of the cyclic reduction process, the coefficients $q_i$, $r_i$, and $p_i$ are held in the "long" vectors LDC, UDC, and C, respectively. In the case of a single tri-diagonal system of equations, $q_i$ and $r_i$ are all zero. Thus, C represents the solution of the system and no further processing need take place in NET_TRID_ SOLVE (see Figure 2).

Sub-programs SOLVE_2X2 and SUB_WRAP_TERMS perform the special eliminations that are required if a network represents a closed loop, while the substitutions from (5) into the m junction equations take place in SUBROUTINE JUNCTIONS. Both of these are sequential tasks, and no use of the DAP's parallelism can be made in performing them. It is thus vital to the efficiency of our algorithm for solving almost tri-diagonal problems that the number of junctions, m, in a network, as well as being very much smaller than the total number of nodes, N, also is small in absolute terms.

The subprogram that solves the closed mxm matrix problem for $x_{j_k}$ (k=1..m) is invoked by SUBROUTINE JUNCTIONS. On return from JUNCTIONS, the values of x are stored in the DAP FORTRAN vector SOL_JCT with element k being the value of $x_{j_k}$.

Finally, the back-substitutions for $x_{j_k}$ and $x_{j_{k'}}$ in (5) are performed by the DO 500 loop. At the start of each pass through this loop, the values of $j_k$ and $x_{j_k}$ are placed in J_CELL and X_VAL, respectively.

This is followed by the broadcasting of X_VAL to those PEs for which J_CELL is equal to $j_k$ or $j_{k'}$: the broadcasting is effected by the expressions J_CELL .EQ. RIGHT_JCT and J_CELL .EQ. LEFT_JCT. At the end of each pass through the DO 500 loop, the value of $x_{j_k}$ is placed in the $j_k^{th}$ element of C so that on return from NET_TRID_ SOLVE, the complete solution to a sparse matrix problem of form (1) is stored in this "long" vector.

## 3. Applications of the Algorithm for Solving Almost Tri-Diagonal Sets of Equations

The algorithm described here has been used to solve some fairly simple one-dimensional problems in nuclear reactor hydrodynamics on the DAP. In one case, a grid depicting a loop with two branches connected to it (see Figure 1) was used with the total number of nodes being 256. The different steps of the algorithm were timed, and the results, together with estimates for a 1000-node case, are summarized in Table 1. It will be seen that the scalar operations (SUBROUTINE SUB_WRAP_TERMS, JUNCTIONS, and SOLVE_2X2) consume less than 10 percent of the total processing time. This demonstrates that the algorithm uses the DAP's parallel processing power efficiently, provided the requirement m<<N is satisfied. Ideally, of course, N should be 4,096, since this allows the full parallelism of the DAP to be exploited.

Table 1. Breakdown of Processing Time Spent in Different Parts of Algorithm for Solving Sparse Linear Problems With an Almost Tri-Diagonal Structure

| | Number of Network Nodes | |
|---|---|---|
| Code Section | 256 | 1000* |
| SUBROUTINE NET_TRID_SOLVE | 118 ms | 141 ms |
| SUBROUTINE SUB_WRAP_TERMS | 11 ms | 11 ms |
| SUBROUTINE JUNCTIONS | 5 ms | 5 ms |
| SUBROUTINE SOLVE_2X2 | 1 ms | 1 ms |

ms = milliseconds

* Figures are estimates

## 4. Conclusions

An algorithm for solving almost tri-diagonal sets of equations on SIMD computers like the DAP and MPP has been described. The algorithm exploits the sparsity structure of this class of problems, thereby allowing parallel cyclic reduction to be used as the main step in the solution procedure. It has been demonstrated that this results in efficient use of the DAP's processing power in solving problems with an almost tri-diagonal structure, provided the number of equations is comparable to the parallelism of the machine (4,096).

## Acknowledgements

The authors would like to thank Professor David C. Leslie of Queen Mary College, University of London, for his helpful suggestions and advice on the subject of this paper and Diane Elam for typing the manuscript.

## References

[1] R.W. Hockney, and C.R. Jesshope, "Parallel Computers," Adam Hilger Ltd., (1981), 280 pp.

[2] T. Opsahl, "DAP-TRAC: A Practical Application of Parallel Processing to a Large Engineering Code," Ph.D. Thesis, University of London, (1984).

# BLOCK-ORIENTED, LOCAL-MEMORY-BASED LINEAR EQUATION SOLUTION ON THE CRAY-2: UNIPROCESSOR ALGORITHMS

D. A. CALAHAN

Dept. of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI, 48109-1109

## ABSTRACT

Experience with the CRAY-2 on the effects of common memory speed and loading on performance indicate that local-memory-based algorithms have potentially a large advantage. The performance of a number of common- and local-memory algorithms are compared for the LU factorization of a dense system of equations on the CRAY-2. A two-level blocked algorithm is introduced to allow both pivoting and high asymptotic performance. Measurements show as much as a 6:1 speedup between blocked assembly-language versus (traditional) vectorized Gauss Fortran implementations; the contributions of both the algorithm and the language to this speedup are evaluated.

## Introduction

### CRAY-2 Architecture and Algorithm Implications

The CRAY-2 architecture (Figure 1) has several features relevant to this algorithm study.

(a) Common memory features.The massive common memory (CM) trades size for access time, so that a considerable delay is usually encountered in reading from CM. Also, only one data path connects common memory to each processor's functional units.

(b) Local memory. The above speed disadvantages are compensated by a local memory (LM), which serves as backup vector and scalar storage for the functional unit's register storage.

(c) Chaining. The CRAY-2 does not have hardware chaining; this must be achieved by software and/or algorithm means.

The implications of distributed memory (including hierarchies such as CM and LM) on linear algebra algorithm organization has been studied since the existence of paged memory systems [1][2][3][4]. In general, computations must be arranged so that the number of floating-point operations on data at the low memory levels is sufficient to warrant data transfers to these levels. This implies, for example, that a matrix-vector multiply - which involves only two operations for each matrix data element - may perform less efficiently than a matrix-matrix multiply.

### Review of Vector Linear Algebra Algorithms

The asymptotic execution rate (MFLOPS) of a factorization algorithm is equal that of the kernel that performs the add-multiplies associated with reducing rows and columns. Three substantially different such algorithms deserve consideration.

Gauss vector-scalar multiply. This requires that, in reducing the rth row , successive operations on preceding rows must be performed serially since a partial result from each row-operation is used as an operand in the next one (the reader is assumed familiar with this procedure). In rows with lengths longer than the vector functional unit length, this dependency can usually be avoided by assembly coding; it is then termed the GAXPY kernel. It has the advantage of yielding the largest average vector length of any of the following kernels and so is a serious consideration when the matrix size is not significantly larger than the maximum allowable vector length, and assembly coding is allowed. This procedure does not lend itself to partitioning when large matrices are involved, but is a potentially useful subalgorithm in such cases.

Matrix-vector multiply. Early experience with the CRAY-1 indicated that block-oriented algorithm organization had at least a pedagogical advantage for large problems[5][6]. Unfortunately, the necessary {vector - (matrix*vector)} kernel was not made a part of the CRAY scientific library and consequently the kernel was not syntactically distinguished from the rest of CAL-coded factorization algorithms. The organizational concept of basing factorization on matrix-vector multiply subroutines was developed in [7][8], where it was illustrated how unrolling Fortran loops could be used to achieve a high performance completely from a high-level language. This emphasized portability and maintainability. More recently, these kernels - known as second-level BLAS - have been proposed as the basis for other common linear algebra algorithms[9].

Matrix-matrix multiply. Although it has always been clear that factorization can be accomplished by a matrix-matrix multiply kernel, the CRAY-1 memory hierarchy was not sufficiently distinctive to achieve a significant advantage over the above two kernels[3]. The additional memory paths of the X-MP made this even less attractive, and partially reduced the advantage of the matrix-vector multiply above. The disadvantages of basing factorization on matrix multiplies are the necessity for other matrix-level kernels to perform reciprocations and substitutions and, most important, the difficulty of partial pivoting.

The memory distribution must be quite distinctive to warrant the programming effort of matrix*matrix-based factorization. This paper documents this case for the CRAY-2, while laying the groundwork for a multiprocessor implementation.

## Pivoting Algorithms

### M*V-based Algorithms

Given a set of equations*

$$\mathbf{A}X=B$$

where **A** is an nxn matrix and X and B are vectors, the factorized solution proceeds by forming lower and upper triangular factors **L** and **U**, viz

$$\mathbf{A} = \mathbf{LU} \tag{1}$$

and then solving for Y and X in substitution steps

$$\mathbf{L}\,Y = B, \qquad \mathbf{U}\,X = Y$$

The complexity of the factorization step of Eq. (1) is $O(n^3)$. The substitutions have complexity $O(n^2)$, with only one add-multiply for each **L** and **U** element; consequently no algorithmic speedup results from transferring **L** and **U** to local memory, so that these substituion steps will not be studied.

*Matrices are in bold, vectors are in upper case, and scalars are in lower case type.

0190-3918/86/0000/0375 $01.00 © 1986 IEEE

For algorithms based on a matrix-vector multiply (abbr. M*V-based), the columns of L and rows of U are indicated as in Figure 2. Here the diagonal element, the row to its right, and the column below it are denoted $a_{22}$, $A_{12}$, and $A_{21}$ respectively. Ignoring pivoting for the moment, the steps to perform the factorization are then

$$\left|\begin{matrix} a_{22} \\ A_{32} \end{matrix}\right| \longleftarrow \left|\begin{matrix} a_{22} \\ A_{32} \end{matrix}\right| - \left|\begin{matrix} A_{21} \\ A_{31} \end{matrix}\right| A_{12} \qquad (2)$$

$$A_{23} \longleftarrow A_{23} - A_{21} A_{13} \qquad (3)$$

$$a_{22} \longleftarrow 1/a_{22} \qquad (4)$$

$$A_{32} \longleftarrow a_{22} A_{32} \qquad (5)$$

The performance of the matrix*vector kernel of Eqs. (2) and (3) depends on the implementation; even Fortran codings have radically different performance (see Appendix).

## M*M-based Algorithms

### Multiply kernel (level 1)

Another matrix partition permits the factorization to be performed on submatrices (Figure 3). The equations equivalent to (2) - (3) are

$$\left|\begin{matrix} A_{22} \\ A_{32} \end{matrix}\right| \longleftarrow \left|\begin{matrix} A_{22} \\ A_{32} \end{matrix}\right| - \left|\begin{matrix} A_{21} \\ A_{31} \end{matrix}\right| A_{12} \qquad (6)$$

$$A_{23} \longleftarrow A_{23} - A_{21} A_{13} \qquad (7)$$

where $A_{22}$ is an $n_d x n_d$ matrix. This multiply kernel has an asymptotic execution rate of approximately 400 MFLOPS when written in assembly language (CAL) and executing from LM with $n_d = 64$. This includes time to transfer to/from CM from/to LM under a daytime memory load condition.

### Pivoting and block reduction (level 2)

On a vector machine such as the CRAY-2, partial column pivoting has two components: (1) the search for the maximum element of a column, and (2) exchange of two complete rows of the matrix. The latter is usually preferred over maintenance of an index pointer in order to avoid relatively slow indirect addressing. These two functions are denoted

$$a \longleftarrow \text{piv} \{ s, V \}$$

where $a$ is the element of maximum absolute value of scalar s and the elements of vector V.

In M*V-based factorization this search is routinely performed after Eq. (2) or (3) by the step

$$a_{22} \longleftarrow \text{piv} \{ a_{22}, A_{32} \} \qquad (3a)$$

However, in the M*M-based version, the granularity of the algorithm does not recognize individual matrix elements and columns. The problem then becomes to preserve the high performance of the matrix-matrix multiply by performing the majority of computations at the block-level, yet to occasionally expose individual columns to permit pivoting. The solution is the following 2-level algorithm (Figure 3).

In level 1, Equations (6) and (7) are first carried out as an $O((n/n_d)^3)$ process. The columns of the resulting $A_x = [ A_{22}^T \ A_{32}^T ]^T$ block-column matrix are at this point partially reduced, with all the accumulations from the columns of $A_{31}$ performed but without contributions from the internal columns of $A_x$. Level 2 begins by reducing $A_x$ using either a GAXPY kernel or the M*V method of Eqs. (2)-(5), viz (an underline represents components of this second reduction level)

$$\left|\begin{matrix} \underline{a}_{22} \\ \underline{A}_{32} \end{matrix}\right| \longleftarrow \left|\begin{matrix} \underline{a}_{22} \\ \underline{A}_{32} \end{matrix}\right| - \left|\begin{matrix} \underline{A}_{21} \\ \underline{A}_{31} \end{matrix}\right| \underline{A}_{12} \qquad (8)$$

$$\underline{A}_{23} \longleftarrow \underline{A}_{23} - \underline{A}_{21} \underline{A}_{13} \qquad (9)$$

$$\underline{a}_{22} \longleftarrow \text{piv} \{ \underline{a}_{22} , \underline{A}_{32} \} \qquad (10)$$

$$\underline{a}_{22} \longleftarrow 1/\underline{a}_{22} \qquad (11)$$

$$\underline{A}_{32} \longleftarrow \underline{a}_{22} \underline{A}_{32} \qquad (12)$$

The computations of Eq. (8-12) are performed from CM and will so be slowed by memory access delays. The execution rate of this step is approximately 125 MFLOPS, using a CAL-coded pivot search.

Eqs. (8)-(12) have the effect of performing the block factorization and substitution steps

$$A_{22} \longleftarrow L_{22} U_{22} \qquad (13)$$

$$A_{33} \longleftarrow A_{32} U_{22}^{-1} \qquad (14)$$

Level 2 is then completed by the block substitution

$$A_{32} \longleftarrow A_{32} L_{22}^{-1} \qquad (15)$$

This can be carried out in local memory at a speed somewhat less than 200 MFLOPS.

With $n_d$ fixed, the complexity of level 2 is readily shown to be $O(n^2)$, whereas the M*M kernel complexity remains $O((n/n_d)^3)$. For large n, the execution rate should therefore approach that of the multiply kernel or approximately 400 MFLOPS.

## Performance

Both the M*V- and the M*M-based pivoting algorithms were run on the MFECC CRAY-2 in November, 1985, using the CIVIC Fortran compiler and CRAY-2 assembly langusge (CAL). Figure 4 presents the results of a number of algorithms and implementations.

The poor performance of the conventional vectorized Fortran Gauss algorithm is due to the lack of chaining and the long, single path to CM. A significant advantage accrues from use of the M*V algorithm, with or without use of CAL. The superior performance of the M*M-based solution eventually becomes evident for $n \geq 1000$; indeed the latter is the best algorithm for all but the smallest n shown. However, it would likely not be worth the programming effort for matrices in the order of n = 100.

Although such large matrices can rarely be factored without pivoting, it is interesting to observe the degradation due to introduction of the 2-level algorithm. The CRAY-2 implementation the piv{ s, V } function of Eq.

(3a) requires a fixed overhead dependent only on the length of V and independent of the matrix element values. Consequently, it is possible to delineate between the pivoting speedown due to piv{ s, V } and that due to the 2-level nature of the algorithm. These are presented in Figure 5 for the M*M algorithm. For n $\geq$ 256, the larger degradation is the result of the piv { s, V } function. Since the latter cannot be avoided, the algorithmic speedown from the introduction of a second level does not appear significant.

### Parallel Implementations

In general, the partitioning of an algorithm into larger computational tasks favors a parallel implementation, since fewer task startups are involved. Thus, an M*M-based algorithm seems advisable, with $n_d$ large. However, in a CRAY-2 system dedicated to an equation solution (a somewhat unlikely event), equalizing the workload among the processors (load-leveling) also becomes an issue; this favors smaller tasks associated with M*V-based factorization or else a smaller $n_d$. These issues are currently under investigation.

## APPENDIX

Fortran statement complexity has a significant impact on the performance of linear algebra and other scientific codes. This is illustrated in Figure 6, where the M*V Fortran kernel is unrolled and the resulting performance plotted. Performance continues to increase significantly for unrollings as large as 32, in contrast to the X-MP, where a 4-way unrolling is considered satisfactory. This phenonenom is considered to be the result of above-mentioned architectural featires. A 16-way unrolling is used in the M*V kernels cited in Figure 4.

## ACKNOWLEDGEMENT

High-performance CAL kernels used in the equation solvers were developed by Geoffrey Carpenter, U. M. Fayyad, and Jimmy Hsiao with a CRAY-2 instruction-level timing simulator developed by K. B. Elliott, as part of a CRAY-2 scientific library sponsored in part by NASA Ames Research Center [11 ].

The research was sponsored by the Mathematical and Information Sciences Directorate of the Air Force Office of Scientific Research under Grant AF-AFOSR 84-0096. CRAY-2 time was provided by MFECC, Lawrence Livermore National Laboratory.

## REFERENCES

[1] McKellar, A. C., and E. G. Coffman, "Organizing Matrices and Matrix Operations for Paged Memory Systems," CACM, vol. 12, no. 3, March, 1969, pp153-155.

[2] Von Fuchs, G., J. R. Roy, and E. Schrem, "Hypermatrix Solution of Large Sets of Symmetric Positive Definite Linear Equations," Comput. Math Appl. Mech. Engring., vol. 1, 1972, pp197-216.

[3] Calahan, D. A., "A Block-Oriented Sparse Equation Solver for the CRAY-1," Proc. 1979 Intl. Conf. on Parallel Processing, Bellaire, MI, pp234-239.

[4] Liu, P. S., and T. Y. Young, "VLSI Array Design Under Constraint of Limited I/O Bandwidth," Trans. IEEE, vol. C-32, no. 12, December, 1983, pp1160-1170.

[5] Calahan, D. A., "Preliminary Report on Results of Matrix Benchmarks on Vector Processors," Report #96, Systems Engineering Laboratory, University of Michigan, May, 1976.

[6] Fong, K. and T. Jordan, "Some Linear Algebraic Algorithms and Their Performance on the CRAY-1," Report LA-7664, Los Alamos Scientific Laboratory, June, 1977.

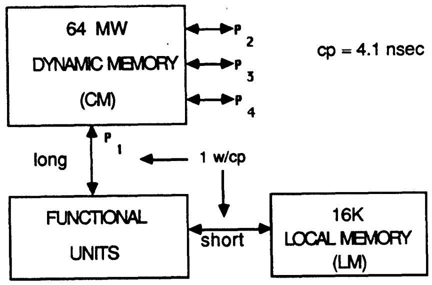
[7] Dongarra, J. J., and S. C. Eisenstat, "Squeezing the Most out of an Algorithm in CRAY Fortran," Report ANL/MCS-TM-9, Mathematics and Computer Science Division, Argonne National Laboratory, May, 1983; also in ACM Trans. on Mathematical Software, vol. 10, no. 3, pp221-230, 1984.

[8] Dongarra, J. J., F.G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," SIAM Review, vol. 26, pp91-112, 1984.

[9] Dongarra, J. J., J. DU Croz, S. Hammarling, and R. J. Hanson, "A Proposal for an Extended Set of Fortran Basic Linear Algebra Subprograms," Report ANL/MCS-TM-41, Mathematics and Computer Science Division, Argonne National Laboratory, December, 1984.

[10] Saad, Youcef, "Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors," Report YALEU/DCS/RR-348, Department of Computer Science, Yale University, January, 1985.

[11] Calahan, D.A., P.L. Berry, G.C. Carpenter, K.B. Elliott, U.M. Fayyad and C.M. Hsiao. "MICHPAK: A Scientific Library for the CRAY-2," Report SARL #8, Supercomputer Algorithm Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, December 1, 1985.



488 MFLOPS peak / processor

459 MFLOPS attainable / processor

Figure 1. CRAY-2 architecture

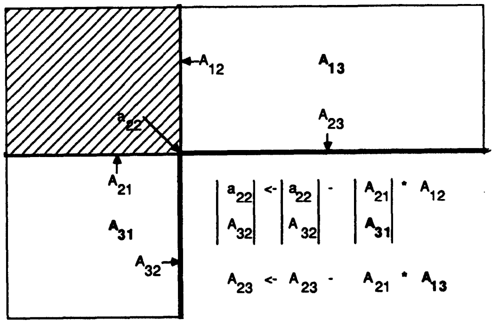


Figure 2. M*V-based factorization algorithm

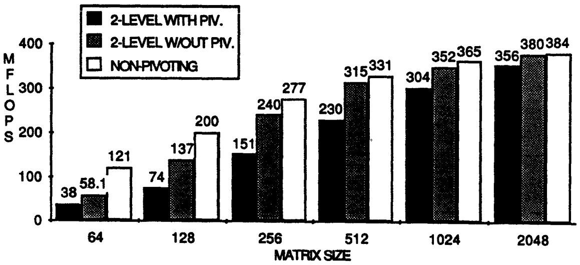


Figure 5. Influence of pivoting, 2-level algorithm

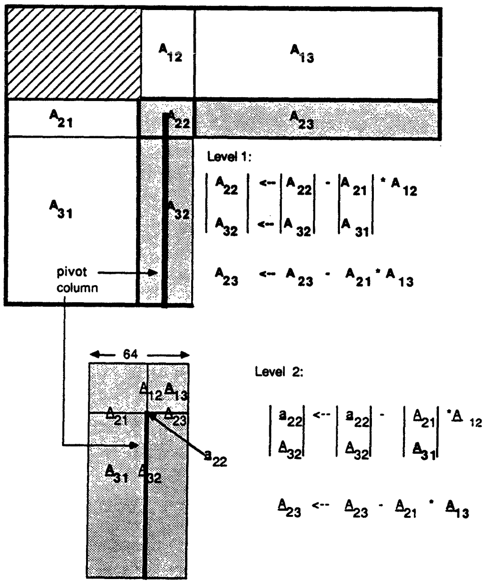


Figure 3. 2-level M*M-based factorization algorithm

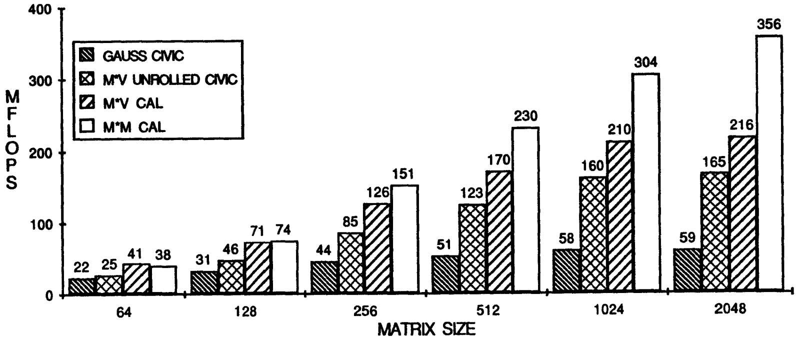


Figure 4. Performance of factorization algorithms (with pivoting)

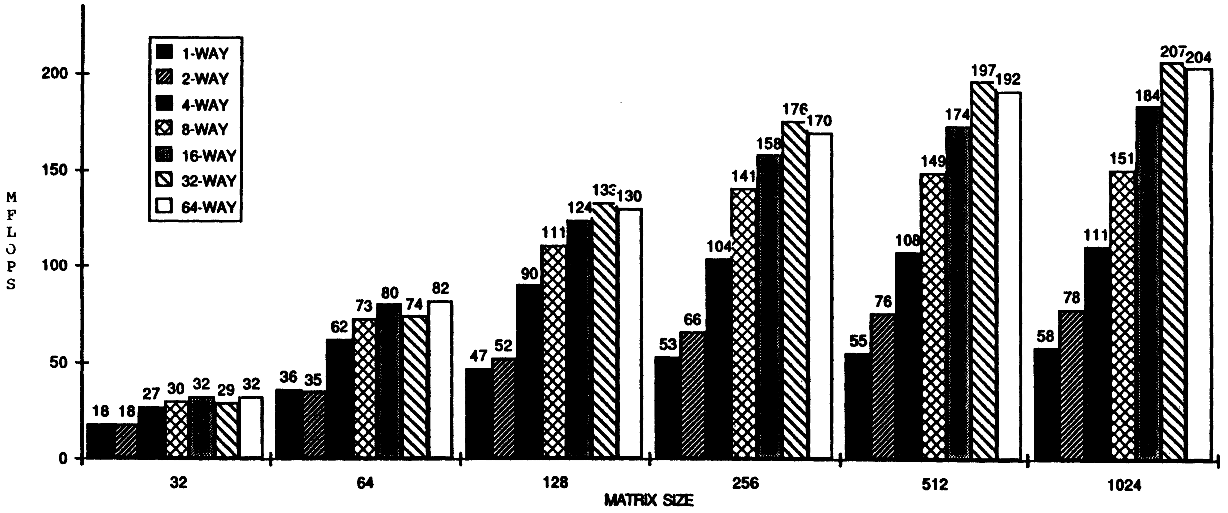


Figure 6. Effects of unrolling Fortran M*V kernel

# VLSI TIME/SPACE COMPLEXITY OF AN ASSOCIATIVE PARALLEL JOIN MODULE

A. R. Hurson
Department of Electrical Engineering and Computer Engineering
The Pennsylvania State University
University Park, PA 16802

## ABSTRACT

Recent interests in the so called 5th generation machines and languages coupled with the inefficiency of conventional machines in handling large volumes of non-numeric data, have given a new dimension to the design and implementation of data base machines. The majority of these architectures are based on the relational model, because of its simplicity and the mathematical foundation on which this model is based. Among the primitive relational operations, literature has placed the greatest emphasis on join. This is because of its complexity and the practical applications of the join operation in combining relations with common domain(s). This paper introduces an associative parallel join algorithm and its hardware design. In addition, it addresses the VLSI time/space complexity of the proposed model, as well as its performance evaluation.

## 1. INTRODUCTION

Continuing interests in data base systems coupled with advances in technology have motivated special purpose architectures. These architectures are designed to remove the shortcomings of the conventional von-Neumann design in the efficient handling of large volumes of non-numeric operations. In addition, recent discussions about knowledge base machines [16], and logic data bases have opened a new door for further research in this area. Finally, the definition of 5th generation languages such as PROLOG [13] calls for the practical/efficient implementation of these languages which necessitates the design and implementation of data base architectures.

The architecture of these special purpose systems is based on constraints which are imposed by the data base environment, i.e., associating name space with the information space at the high level, and performing a sequence of simple and repetitive non-numeric operations on a massive amount of data at the low level. Data base machines have met these conditions by:

i) searching and selecting the data at or near secondary storage in an associative fashion, and

ii) hardware implementation of suitable parallel algorithms for non-numeric operations.

In contrast to conventional architectures, such a direction eliminates the existing address mapping resolution and reduces the data transportation and semantic gap. The underlying structure of the majority of these architectures are based on the relational model [4]. This is due to the following reasons:

i) The simplicity of the relational model,

ii) The strong mathematical foundation on which the relational model is based, and,

iii) The one-to-one mapping between the representation of data and primitive operations in logic programming and the relational model [4, 6].

In practice, the performance of a relational data base system has been associated with the join operation. Such an association is based on the practical applications of the join in the queries and the complexity of the operation. This complexity directly affects the time and space efficiency of the join operation. Different hardware/software algorithms have been defined and implemented to improve the efficiency of such operation. This paper addresses a hardware module for the join operation along with its performance evaluation. Section 2 overviews some of the proposed join algorithms. Section 3 introduces the overall organization and the flow of data and control in the proposed architecture. Section 4 discusses the VLSI time/space complexities of the proposed model, and finally, section 5 evaluates the performance of the model.

0190-3918/86/0000/0379 $01.00 © 1986 IEEE

## 2. BACKGROUND

Join has been defined as a binary operator which combines two relations over their common attribute(s). Formally, let r(R) and s(S) be two relations and let $A \varepsilon R$ and $B \varepsilon S$ be $\theta$ compatible, where, $\theta \varepsilon \{=, \neq, <, >, \leq, \geq\}$, then $r[A\theta B]s$ is defined as:

$\{t| t_r(A)\theta t_s(B)$ and $t(R)=t_r$ and $t(S) = t_s\}$.

In a uniprocessor environment, three classes of join algorithms have been studied, namely nested loops, sort-merge, and hashing, with respective execution times of $0(n^2)$, 0(nlgn) and O(n). However, in a thorough evaluation of these algorithms, parameters such as communication, simplicity, regularity, space efficiency... etc., should also be taken into consideration. In the data base architecture where a higher performance and throughput is cited, the discussion about join should center around a parallel version of these algorithms. Although there are some designs which have ignored such a discussion[2,19], there is no reason to accept that they would not be able to perform this function. The assumption is that either a front-end machine or a dedicated join module will do the job.

Literature has studied different algorithms for join in both uniprocessor and multiprocessor environments[3,21]. In the following discussion, some of these algorithms which are proposed for systems on the exhaustive search policy will be overviewed:

CAFS[1] utilizes a hashing scheme. It can perform the semi-join operation by marking an array of bits based on the tuples in r and then using this array to select proper tuples in s. However, the architecture has not addressed the implementation of the Θ-join. Based on the hardware capability of the design, such an operation is expensive since it requires several passes over the data file.

RAP[17] has studied two types of join namely implicit (e.g. semi-join) and physical (e.g. Θ-join); the latter can be carried out as a sequence of implicit joins. The algorithm is based on the concept of nested loops, with some degree of parallelism, which is determined according to the number of read/write heads and the hardware capability of each read/write head.

DIRECT[5] is capable of performing a block oriented version of nested loops with some degree of parallelism (e.g. number of processors). During each iteration, a block (page) of the outer relation (say (r)) will be joined with entire blocks of the inner relation(s) which are distributed among different processors. Within each processor a simple sort-merge algorithm will be utilized to join two blocks together.

DELTA[7] has defined a parallel version of the sort-merge algorithm in its design: 12 sorters in parallel sort the relations in sequence based on the join attribute, and then a single merger is utilized to merge the sorted relations.

The performance of these algorithms has been formulated and compared against each other. Bitton et. al.[3] have shown that in a uniprocessor environment, the sort-merge algorithm has a better performance than nested loops. However, in a multiprocessor environment such as DIRECT, the nested loops algorithm outperforms sort-merge. This is due to the higher degree of parallelism in the operation, and better processor utilization made by the nested loops algorithm.

Valduriez and Gardarin[21] have compared the performance of a block oriented version of nested loops, parallel sort-merge, and parallel hashing schemes against each other. They have shown that in general, if the number of accesses to the hashed file is low, hashing algorithms offer a better performance than the other two techniques. Moreover, for relations of the same size, the sort-merge algorithm has a better performance. Finally, if the ratio between the relation sizes is different from 1, nested loops algorithm is the superior method.

The evaluation process as above[3,21] partially represents the relative behavior of these algorithms. A thorough evaluation and comparison should address:

i) The merit of each approach based on the environment, and

ii) Equally important parameters such as, space and processor utilization, ease of implementation from hardware/software point of view, processor-memory intercommunication and finally, practical overhead and practical limitations.

## 3. PROPOSED ARCHITECTURE

Associative memory has been defined as: "a collection or assemblage of elements having data storage capabilities, and which are accessed simultaneously and in parallel on the basis of data content rather than by specific address or locations,"[14]. Content addressability of data implies that each basic cell of memory should have the hardware capability to perform read, write, and search operations. Such a capability eliminates name resolution problem[9] and reduces the bottle neck between memory and CPU.

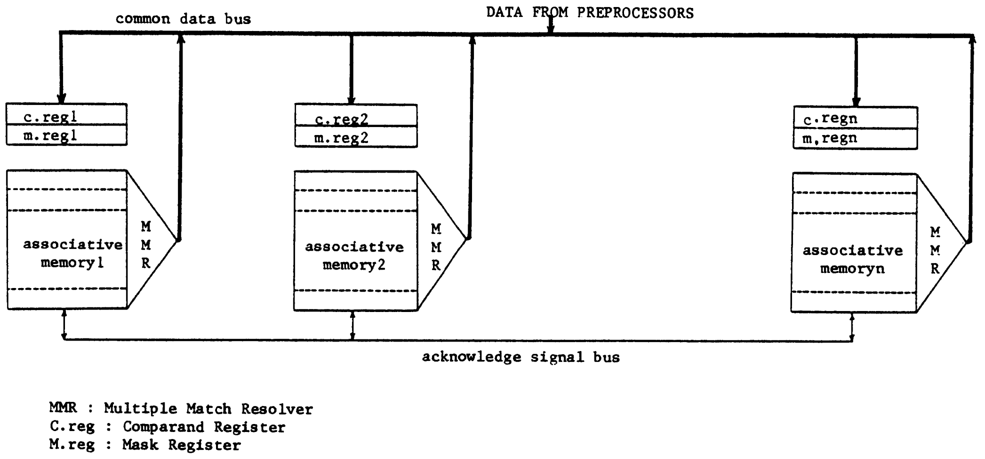


Figure 1 : Overall Organization of the Proposed Join Module.

Associative memory provides a suitable and fast medium for representation of the relational model and implementation of the basic relational operations. This stems from the fact that the representation of data in the associative memory is similar to the tabular representation of the data in the relational model. In addition, attributes in the relational data model are equally available to be searched, as in the associative memory where search can be performed on any defined field. Finally, there is a one to one mapping between the set operations and the associative operations.

The proposed join module is a collection of n identical and independent modules of associative memory (Figure 1). Associative memories can transfer data among each other under the control of a master controller, via a common data bus. In addition, they can pass control signals through a chain of acknowledge bus. These modules can be linked together to make a memory of size (k*w)*d ($1\leq k\leq n$, w is the word length and d is the depth) capable of holding d tuples of size k*w each, or they could be linked to form a memory of size w*(k*d) capable of holding k*d tuples of size w. This facility enables the system to adjust the available memorybased on the length of the tuples and the cardinality of the relation. The independence of the associative memories from each other enhances the modularity of the system and as such the fault tolerance of the system.

With respect to a query, domains in a relation (say r with relation scheme R) can be grouped into three classes:

i) those which are defined as a member of the search argument part (i.e., SA = $\{R_i | R_i \in R$ and $R_i \in$ search argument$\}$),

ii) those which are the member of the output set part (i.e., OS = $\{R_i | R_i \in R$ and $R_i \in$ output set$\}$), and

iii) those which are not the members of SA or OS (i.e., $\{R_i | R_i \in R$ and $R_i \notin (OS \cup SA)\}$).

Our assumption is that the relations are preprocessed before being routed to the join module, and then the selected tuples are passed to this module. In other words, a subset of the relation ($r_s$) with a relation scheme ($R_s$)will be explicitly stored in the join module:

$R_s \subseteq R$ and $\forall\, T_s \in r_s \Rightarrow T \in r$ such that

$T_s = T[OS]$ and $T[SA] \equiv$ search argument

This assumption is in accordance with the 90-10% rule.

The proposed join module performs a concurrent version of the nested- loops algorithm. Concurrency is achieved as a result of:

i) embedded parallelism in the associative operations, and

ii) the independence of associative memories from each other.

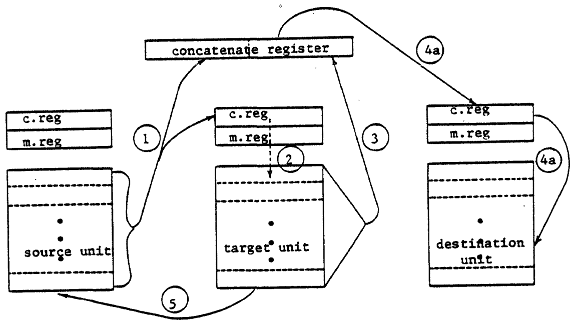
Therefore, we have parallelism within each associative memory and overlapping across the memories. For a join operation, three memory modules will be tightly coupled together (Figure 2). In our discussion we call them, source, target and destination modules respectively. Interestingly, one can utilize random access memory modules to house source and destination relations. However, for the sake of the uniformity and generality of the proposed join module, in the rest of this paper we assume three associative memory modules to house source, target and destination relations. Tuples from the source module (source-tuples) are routed to the concatenate-register and target module (1) one at a time. A source-tuple is searched against the contents of the target module (2). In case of a match, the selected tuples are routed to the concatenate-register in pipeline fashion (3). The result of concatenation is then sent to the destination module (4). When, the selected tuples in the target module are about to be exhausted, an acknowledge signal is sent to the source module informing the need for a new tuple (5).

The order of this algorithm with respect to the number of searches is O(n), as opposed to $O(n^2)$ in the uniprocessor systems. Moreover, because of the interaction between the source and target memory modules (acknowledge signal bus), operations in the source memory can be initiated and overlapped with operations in the target memory. From this discussion, one can conclude that the steps ① , ④ , and ⑤ are overlapped with step ③ (Figure 2). Algorithm 1 represents the sequence of operations according to the notations and the organization which have been adopted in this paper. The generality and adaptability of the associative operations increases the versatility of the proposed model in performing different variations of the join operations (e.g. natural join, semi-join) without additional hardware overhead.

The proposed model can be extended over a parallel execution of a sequence of interrelated joins. Such a concept is of special interest in a PROLOG type logic programming environment[13].

```
FOR ALL $t_r \in r$ DO;
    parallel search $t_r$ against s
    while there is a $t_r$ such that $t_r(A) \theta t_s(B)$
        $q = t_r || t_s$
    end;
end;
```

ALGORITHM 1: AN ASSOCIATIVE JOIN



① source tuple to concatenate-reg & target module

② associate search

③ target tuple to concatenate-reg

(4a) destination tuple to destination unit

(4b) store

⑤ acknowledge signal

Figure 2: Flow of Data in a Join Operation.

## 4. VLSI TIME AND SPACE COMPLEXITIES OF THE PROPOSED MODEL

In the design and development of a topology for current technology, one should remember the practical constraints which are imposed by the VLSI technology. The idea is to reduce the execution time as well as the communication through the replication of a few basic blocks in time or space. A fully parallel associative memory bears these conditions and hence is suitable for VLSI implemention. This is not the intension of this article to discuss the organization of associative memory, the interested reader is referred to [14, 20, 22]. In this section we will calculate the VLSI time and space complexities of the proposed model according to the design rules discussed in [15]. Our associative module, should be able to perform read, write and different variations of search operations with resepct to the contents of the mask and word select registers. In our design each cell of the mask, comparand, and memory is a dynamic RAM cell refreshing itself during the second phase of a 2-phase clock system. Figure (3) depicts the stick diagram of the interaction between one bit of mask and comparand registers according to the following interpretation:

$$\begin{array}{lll} One1_j=1 & \text{iff} & M_j=1 \text{ and } Comp_j=1 \\ Z1_j=1 & \text{otherwise} & \end{array}$$

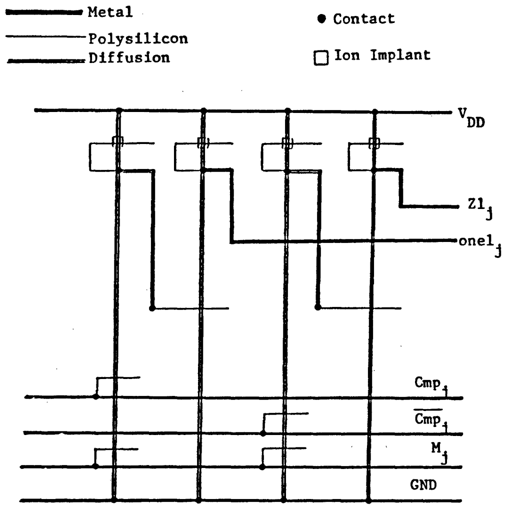


Figure 3: interconnection between mask and comparand registers.

the $Zl_j$ and $Onel_j$ are run in metal and shared by the $j^{th}$ bit of each memory word. The associated circuitry for read/write operations and selection of a selected word has been discussed in [23]. In order to simplify the memory cell and hence to reduce the size of the module: i) control signals (i.e., word select, search) are used at the word level rather than the bit level. In other words, the search is carried out for all the words, and then, the tag bit is set according to the control lines and the word select signal. ii) the searches on equality, in equality, less than equal, greater than equal are performed based on searches on less than and greater than.

Figure (4) represents one bit of the memory cell and the circuitry associated with it, where $C_{i,j}$ ($\overline{C}_{i,j}$) is the content of the $j^{th}$ bit of the $i^{th}$ word, $L_{i,j-1}$, $\overline{L}_{i,j-1}$, $G_{i,j-1}$, and $\overline{G}_{i,j-1}$ are less than and greater than signals generated by the left most bit (e.g. j-1) and finally $L_{i,j}$, $\overline{L}_{i,j}$, $G_{i,j}$, and $\overline{G}_{i,j}$ are the less than and greater than signals generated by the $j^{th}$ bit of word i. Figure 5 depicts the circuitry around the tag bit according to the above discussion, LT, GT, and EQ are the less than, greater than, and equality control lines respectively.

Assuming the $\Delta\tau$ be the average delay time of an inverter, then, the search time is calculated based on: i) 4 parallel delay for a 3-input NAND gate and an inversion, II) (n-1) $2\Delta\tau$ serial operation of (n-1) NOR gates and an inversion, and iii) $9\Delta\tau$ delay for setting the tag bit. Resulting in a total delay time of:

$13\Delta\tau + 2(n-1)\Delta\tau$ (n is the word length)

i) The geometry area of a cell is estimated at $200\lambda * 100\lambda$ including the area needed for routing the signals among the cells. Thus, an associative memory of m words, each n bits long requires an area of: $200n\lambda * 100m\lambda$ For m=512, n=256, $\Delta\tau=1$ ηsec. and $\lambda=1$ μm we will have a search time of less than 1000ηsec for an environment of $2^{17}$ bits (including the expected complexity due to the read/write circuitry around each cell). In addition, the geometry area for the above configuration is estimated to about 5 cm * 5cm.

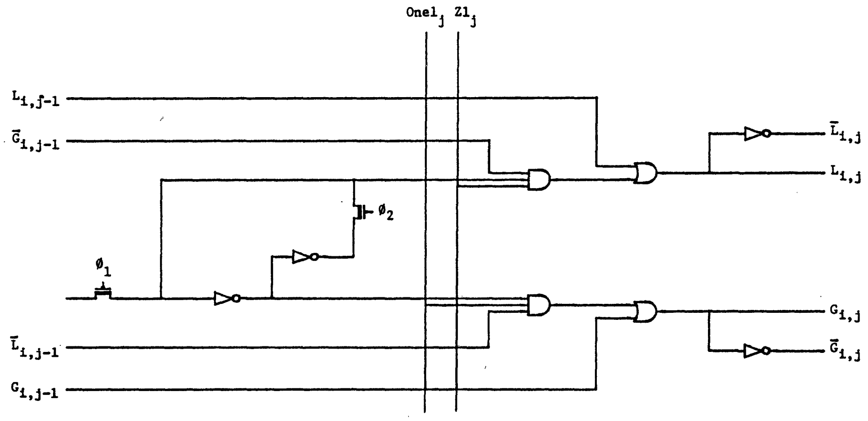


Figure 4: A Memory Cell.

## 5. PERFORMANCE EVALUATION OF THE PROPOSED MODEL

The objective of this seciton is two fold: First, the timing analysis of the proposed model is discussed and then its performance is calculated and compared against some of the available models in the literature.

### Execution time analysis

Our timing analysis is based on the sequence of operations discussed in Algorithm 1 and the following notations.

i) No ordering is imposed on the storage policy - i.e., unordered data file.

ii) According to 90-10% rule, the source and target relations are preprocessed before being transferred to the join module.

iii) Parallel preprocessors share a common cache memory with a hit ratio (H) of 85%. The organization of such intermediate memory between the secondary memory and array of preprocessors is similar to the model in [3,21].

iv) Size of associative memories (e.g. associative blocks) are different from the page sizes in secondary storage. This is due to the reconfigurability of the associative memories (section 3) and the technology.

v) n,m: Number of pages in the source and target relations respectively.

$C_r$: Time to read and pass one page from the cache memory to the preprocessors. It is defined as:

$$C_r=H(R_m)+(1-H)(R_m+R_c) \qquad (1)$$

where $R_m$ and $R_c$ are the page read time of secondary storage and cache memory respectively, and H is the hit ratio.

$C_{select}$: Time to select and project tuples in a page. This is proportional to the number of tuples in a page.

$$C_{select}=K*(T_{select} + T_{project}) \qquad (2)$$

$C_f$: Compaction factor defined by 90-10% rule [9].

P: Number of preprocessors.

$\delta$: The ratio between the associative memory block size and cache memory page size - i.e.,$\delta$=T/K where T and K are the number of words in an associative memory and cache page respectively.

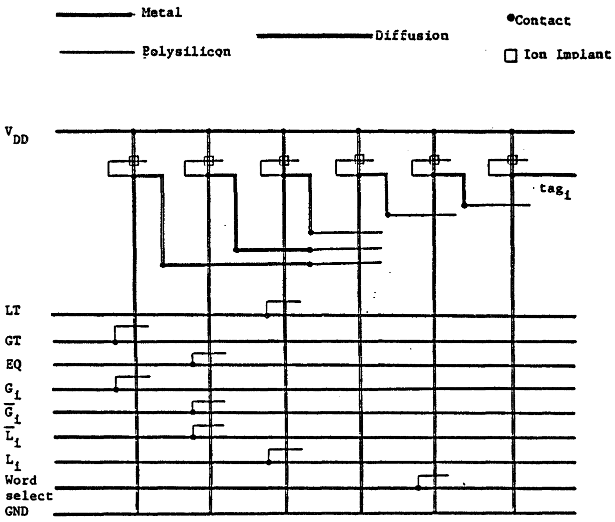


Figure 5: Circuitry around a tag bit.

$C_{PA}$: Time to fill out one associative memory by the preprocessors.

$C_{join}$: Time to join two associative memories together (equation 4).

$C_W$: Time to pass the contents of an associative memory to the front-end processor.

S: The join sensitivity factor. It is defined as the average number of tuples in the target relation which are joined with a tuple in the source relation. It should be noted that due to the preprocessing operation, the data in the associatve memories are more sensitive to the join operation. Therefore, the join sensitivity factor in our design should be higher than the one proposed in [3].

According to these parameters, the execution time of the operation is calculated as:

$$T_{NL} = (\frac{n+m}{P})\ (C_r + C_{select}) + \left\lceil\frac{n*C_f}{\delta}\right\rceil \Big(C_{PA} + \left\lceil\frac{m*C_f}{\delta}\right\rceil (C_{PA}) + C_{join} + S(C_W)\Big) \qquad (3)$$

In which:

$$C_{join} = C_{initial} + C_{read} + T\Big((C_{search}) + S(C_{read})\Big) + C_{ter}. \qquad (4)$$

where:

$C_{initial}$: represents the initial time which one has to spend before initialization of join operation (i.e., setting the mask registers).

$C_{read}$: is the time needed to read out a record from an associative memory module and pass it to the proper destination.

$C_{search}$: is the time needed to search the contents of the comparand register against the contents of the memory in associative fashion. This is a function of the word length and isindependent from the depth of the associative memory.

According to our implementation (Section 4), the search time is unique for equality and greater than/less than operations.

$C_{search}$: $C_{search} = aQ+b$ where Q is the word length and a,b are constant values which are defined by technology.

$C_{ter.}$: is the overhead time which is needed to terminate the operations (i.e., passing the last acknowledge signal to controller, etc.).

Performance Evaluation

Our evaluation and comparison are based on the time analysis of the basic operations as are dictated by the current technology. A detaileddiscussion for the time analysis of the basic operations is given in [24].

Table 1 depicts the execution time of the join module for different word sizes. In addition, Table 1 shows the estimated size of the source and target relations with respect to the 90-10% rule. As can be seen, the join operation between two relations of 256K and 2048K sizes regardless of the input/output operations is about 18 msec.

Table 2 shows the execution time of the proposed model against the models in [3, 21]. All the models in table 2 are based on the nested loops policy. The choice of the algorithm regardless of the uniformity among all algorithms is due to the fact that the nested loops algorithm in a parallel environment offers a better performance and resource utilization (Section 2), than the algorithms based on sort-merge and hashing.

## 6. CONCLUSION AND FURTHER DISCUSSION

An associative nested loops algorithm for join operation has been discussed. It has been shown that in a parallel environment a join operation based on the nested loops policy offers a better performance than algorithms based on the hashing or sort-merge policies. However, through the time analysis we have shown that the proposed algorithm has a better performance than the proposed models in [3, 21] by a factor of 7. In addition, the generality of the associative operations on one hand, and the one to one mapping between the relational operations and associative operations on the other hand can provide a general purpose module for relational operations. This increases the resource utilization of our model in comparison to other proposed models, which are utilized just by the join operation. Finally, our model is capable of handling a sequence of the interrelated join operations in parallel. A feature which is missing in all other proposed models.

TABLE 1: EXECUTION TIME OF THE PROPOSED JOIN MODEL.

| WORD SIZE (Byte) | EXECUTION TIME(msec.) | SOURCE RELATION (Byte) | TARGET RELATION (Byte) |
|---|---|---|---|
| 64 | 4.44 | 64K | 512k |
| 128 | 8.8 | 128K | 1024K |
| 256 | 17.54 | 256K | 2048K |

TABLE 2: EXECUTION TIME OF THE DIFFERENT JOIN MODELS.

| MODEL | EXECUTION TIME(msec.) |
|---|---|
| $T_{NL}$ (proposed model) | 926.93 |
| $T_{NL}$ (Bitton et.al.) | 6593. |
| $T_{NL}$ (Valduriez et.al.) | 6610.44 |

The regularity and, modularity of associative memory, and simplicity of each basic cell has provided a suitable ground for VLSI implementation of the proposed architecture. Our VLSI time and space analysis have shown that the proposed model is well within the range of the current technology.

In the past, due to the high cost of hardware, a wide range applications of associative processing were not feasible. It has been estimated that a basic associative cell is about 2-3 times more complex than a basic cell of random access memory. In addition, an associative module requires a special I/O connections for fast processing. However, because of its preprocessing capability the proposed model utilizes the same I/O connections as the models in [3, 6, 21]. Moreover, improvement in technology and hence cost reduction of the hardware modules, have made it feasible to increase the utilization of associative processing.

## LIST OF REFERENCES

1) Babb, E., "Implementing a relational data base by means of specialized hardware," ACM Transactions on data base systems, Vol. 4, No. 1, March 1979, pp. 1-29.

2) Berra, B., and Oliver, E., "The role of associative array processor in data base machine architecture," IEEE Computer, Vol. 12, No. 3, March 1979, pp. 53-61.

3) Bitton, D. et. al., "Parallel algorithms for the execution of relational data base operations," ACM Transactions on data base systems, Vol. 8, No. 3, September 1983, pp. 324-353.

4) Codd, E. F., "A relational model of data for large shared data systems," CACM, Vol. 13, No. 6, June 1970, pp. 377-387.

5) Dewitt, D. J., "Direct - A multiprocessor organization for supporting relational data base management systems," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 395-406.

6) Gallaire, H., and Minker, J. (eds) Logic and data bases, Plenum Press, 1978.

7) Hiroshi, S., et. al., "Design and implementation of the relational data base engine," proceedings of the international conference of fifth generation computer systems, 1984, pp. 419-426.

8) Hong, Y. C., "Efficient computing of relational algebric primitives in a data base machine architecture," IEEE Transactions on Computers, Vol. C-34, No. 7, July 1985, pp. 588-595.

9) Hsiao, D. K., Data Base Computers, in advance in computers edited by Yovits, M. C., Academic Press, 1980, pp. 1-64.

10) Hurson, A. R., "An associative backend machine", sixth workshop on computer architecture for non-numeric processing, Hyeres, France, June 1981.

11) Hurson, A. R., "An associative backend machine for data base management," IEEE workshop on computer architecture for pattern analysis and image data base management, Virginia, November 1981, pp. 225-230.

12) Ibaraki, T., and Kameda, T., "On the optimal nesting order for computing N relational joins," ACM Transactions on data base systems, Vol. 9, No. 3, September 1984, pp. 482-502.

13) Kluzniak, F., and Szpakowicz, S., Prolog for Programmers, Academic Press, New York, 1985.

14) Kohonen, T., Content Addressable Memories, Springer Verlag, New York, 1980.

15) Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, New York, 1980.

16) Moto-Oka, T. (edi), Fifth Generation Computer Systems, North-Holland, New York, 1983.

17) Schuster et. al., "RAP-2 an associative processor for data base and its applications," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 446-458.

18) Shaw, E. "Relational operator", workshop on data base machine, Minnowbrooke Syracuse, New York, 1980.

19) Lin, C. S. et. al, "The design of a rotating associative memory for relational data base applications," ACM Transactions on data base systems, Vol. 1, No. 1, March 1979, pp. 53-65.

20) Thurber, K. J., and Wald, L. D., "Associative and parallel processors," ACM computing surveys, Vol. 7, No. 4, December 1975, pp. 215-255.

21) Valduriez, P., and Gardarin, G. "Join and semijoin algorithms for a multiprocessor data base machine," ACM Transactions on data base systems, Vol. 9, No. 1, March 1984, pp. 133-161.

22) Yau, S. S., and Fung, H. S., "Associative processor architecture- A survey," ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 3-28.

23) Hurson, A. R., "A VLSI design for the parallel finite state automation and its performance evaluation as a hardware scanner," International Journal of Computer and Information Sciences, Vol. 13, No. 6, December 1984, pp. 491-508.

24) Hurson, A. R., "Timing analysis of a parallel associative join module," Technical Report, Department of Electrical Engineering, The Pennsylvania State University, University Park, PA, December 1985.

# A New VLSI System For Adaptive Recursive Filtering*

**Kam Hoi Cheng and Sartaj Sahni**
Computer Science Department
University of Minnesota
Minneapolis, MN 55455

## ABSTRACT

We develop an efficient bidirectional chain VLSI system for the adaptive recursive filtering problem. Our design is an improvement over previous designs. It matches the performance of a broadcast chain but does not use the broadcast capability.

**Keywords and Phrases**
VLSI architectures, systolic systems, adaptive recursive filtering

## 1. Introduction

VLSI architectures for a variety of problems have been proposed by several authors. A bibliography of over 150 research papers dealing with this subject appears in [KUNG83]. In this paper, we are concerned solely with the adaptive recursive filtering problem. The input to this problem is an $n \times w$ matrix $A$ of weighting coefficients and an $1 \times w$ vector $(x_{1-w}, ..., x_0)$. The output is a $1 \times n$ vector $(x_1, ..., x_n)$ where:

$$x_i = \sum_{j=1}^{w} a_{ij} \; x_{i+j-w-1} \qquad i = 1, 2, ..., n \qquad (1)$$

In evaluating a VLSI design, we assume that the VLSI system will be attached to the host processor using a bus. The evaluation of a VLSI design should take the following into account:

1. Processors --- how many processors are used in the VLSI system? This figure is denoted by $P$.
2. Bus bandwidth --- the maximum amount of data to be transmitted between the host and the VLSI system in any cycle. This figure is denoted by $B$.
3. Speed --- how much time does the VLSI system need to complete its task? This time may be decomposed into the times $T_C$ (time for computations) and $T_D$ (time for data transmissions both within the VLSI system and between the host and the VLSI system).

Let $C$ denote the time spent for computation by a single processor algorithm and $D$ denote the total amount of data that needs to be transmitted between the host and VLSI system. For the adaptive recursive filtering problem, $C = nw$ and $D = nw + n + w$.

The ratio

$$R_D = B * T_D / D$$

measures the effectiveness with which the bandwidth $B$ has been used. Clearly, $R_D \geq 1$ for every VLSI design.

The ratio

$$R_C = P * T_C / C$$

measures the effectiveness of processor utilization. Once again, we see that $R_C \geq 1$ for every VLSI design.

Finally, we may combine the two efficiency ratios $R_C$ and $R_D$ into the single ratio $R = R_C * R_D$. A design that makes effective use of the available bandwidth and processors will have $R$ close to 1.

The efficiency measure $R$ as defined here is the same as that used in [CHEN84a,b and 85] to evaluate VLSI designs for matrix multiplication and back substitution. This measure is also quite similar to that proposed in [HUAN82]. In fact, the two measures become identical when $T_C = T_D$.

In comparing different architectures for the same problem, one must be wary about over emphasizing the importance of $R_C$, $R_D$ and $R$. Clearly, by using $P = 1$ and $B = 1$, we get $R_C = R_D = R = 1$ but no speed up at all. So, we are really interested in minimizing $T_C$ and $T_D$ while keeping $R$ close to 1.

VLSI architectures for the adaptive recursive filtering problem have been proposed earlier in [KUNG78 and 84], [LEIS83], [HUAN82] and [ROBE84]. The design of [HUAN82] uses a broadcast chain and has $P = w$, $B = w + 2$, $T_C = n + w - 1$, $T_D = n + w$, $R_C \sim 1 + w/n \sim 1$, $R_D \sim 1 + 1/w + w/n \sim 1$ and $R \sim 1$. The design of [KUNG78] uses a bidirectional chain of processors. An improved version is described in [LEIS83]. For this, $P = \lceil w/2 \rceil$, $B = \lceil w/2 \rceil + 2$, $T_C = 2n + w - 2$, $T_D = 2(n + w - 1)$, $R_C \sim 1 + w/(2n) \sim 1$ $R_D \sim 1 + 3/w + w/n \sim 1$ and $R \sim 1$. The design of [KUNG84] uses a systolic ring architecture to solve the simple recurrence problem. It can be easily extended to solve the adaptive recursive filtering problem. This extension has $P = \lceil w/2 \rceil$, $B = \lceil w/2 \rceil + 1$, $T_C = 2(n - 1) + w$, $T_D = 2(n + w - 1) + 1$, $R_C \sim 1 + w/(2n) \sim 1$, $R_D \sim 1 + 1/w + w/n \sim 1$ and $R \sim 1$.

While all the above designs have an $R \sim 1$, the broadcast chain of [HUAN82] has a $T_C$ and $T_D$ that is about half that of the other designs. In this paper, we develop a bidirectional chain VLSI system that has the same (actually slightly smaller) $T_D$ and $T_C$ as the broadcast chain of [HUAN82]. For our design, $P = w$, $B = w + 1$, $T_C = n + \lceil w/2 \rceil$, $T_D = n + w + 1$, $R_C \sim 1 + w/(2n) \sim 1$, $R_D \sim 1 + w/n \sim 1$ and $R \sim 1$. Our design shows that a broadcast chain is not required to obtain this $T_C$ and $T_D$ performance.

## 2. $o(n)$ Throughput Bidirectional Chain

An $o(n)$ throughput bidirectional chain for the adaptive recursive filtering problem can be obtained by extending the systolic design of [ROBE84] for the nonadaptive recursive filtering problem. This extension requires us to recast (1) into the following form:

*This research was supported in part by the National Science Foundation under grant MCS-83-05567.

0190-3918/86/0000/0387 $01.00 © 1986 IEEE

$$\begin{aligned} x_i &= \sum_{j=1}^{w} a_{ij}\ x_{i+j-w-1} \\ &= \sum_{j=1}^{w-1} a_{ij}\ x_{i+j-w-1} + a_{iw} \sum_{j=1}^{w} a_{i-1,j}\ x_{i+j-w-2} \\ &= \sum_{j=1}^{w-1} a_{ij}\ x_{i+j-w-1} + a_{iw} \sum_{j=0}^{w-1} a_{i-1,j+1}\ x_{i+j-w-1} \\ &= \sum_{j=0}^{w-1} b_{ij}\ x_{i+j-w-1}, \qquad i \geq 1 \qquad (2) \end{aligned}$$

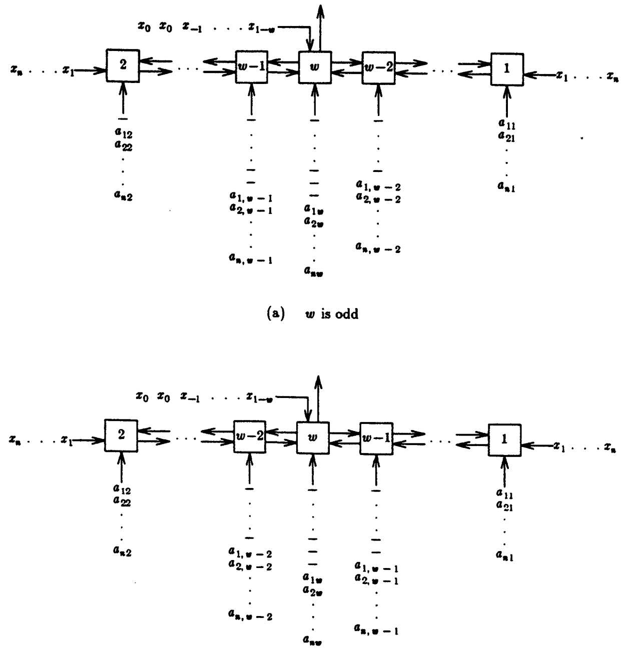
where

$$b_{i0} = a_{iw}\, a_{i-1,1},\ b_{ij} = a_{ij} + a_{iw}\, a_{i-1,j+1}, \qquad 1 \leq j \leq w-1 \qquad (3)$$

$$a_{01} = 1,\ a_{0j} = 0,\ 2 \leq j \leq w, \qquad x_{-w} = x_0 \qquad (4)$$

To calculate the $b_{ij}$'s of (3) dynamically, $w$ PEs in addition to the $w+1$ PEs used in [ROBE84] are needed. The performance figures of the resulting VLSI system are $P = 2w + 1$, $B = 2w + 3$, $T_C \sim n + \lceil w/2 \rceil$, $T_D \sim n + w$, $R_C \sim 2 + 1/w + w/n \sim 2$, $R_D \sim 2 + 1/w + w/n \sim 2$ and $R \sim 4$.

Improved performance can be obtained by using the bidirectional chain architecture of Figure 2.1. All the even numbered PEs are on the left, while all the odd numbered PEs are on the right. The output is generated from the middle PE, PE($w$). The PEs to the left of PE($w$) compute all terms involving even columns of $A$, while PEs on the right compute all terms involving odd columns of $A$. The case when $w$ is odd is shown in Figure 2.1(a). The case when $w$ is even is shown in Figure 2.1(b).



(a) $w$ is odd

(b) $w$ is even

Figure 2.1

The middle processor, PE($w$), has the five registers: $A$, $V$, $X$, $Y$ and $Z$. The remaining PEs have three registers ($A$, $X$ and $Y$) each. We use the notation $R(i)$ to denote register $R$, $R \in \{A, V, X, Y, Z\}$, of PE($i$). The $A$ register of each PE is used to hold an input value from the $A$ matrix. PE($i$) receives input from column $i$ of $A$ only, $1 \leq i \leq w$. The $X$ register of each PE holds an $x_i$ value while the $Y$ registers hold partial sums in the computation of an $x_i$. In each cycle, the $X(i)$s move one step away from the center PE, PE($w$), while the $Y(i)$s move one step towards this PE.

The working of the VLSI system is described formally in Algorithm 2.1. The first **for** loop sets up the initial configuration. The three steps in the **parallel do** are executed simultaneously. When this **for** loop terminates, PE($w$) contains $x_p$ for $p = \lceil (w-1)/2 \rceil - w = \lceil -(w+1)/2 \rceil$ in its $X$ register. The $X$ register of a PE that is $a$ units away from PE($w$) contains $x_{p-a}$. The second **for** loop contains two sets of concurrently executed statements. In the first set, i.e. first **parallel do**, essentially five concurrent activities are performed in each iteration of this loop:

(1) PE($w$) either inputs an $x_i$, $i \leq 0$ or outputs a newly computed $x_i$, $i > 0$.

(2) All $X$ values move one PE away from the middle PE.

(3) Each PE inputs an $A$ value. Note that we assume $a_{ij} = 0$ for $i \leq 0$.

(4) All $Y$ values move one PE towards the middle PE. However, the $Y$ value from PE($w-1$) is moved to the $Z$ register of PE($w$) rather than to its $Y$ register (this latter register receives the $Y$ value from PE($w-2$)). The boundary PEs (1 and 2) reset their $Y$ registers to zero.

(5) From the data patterns of Figure 2.1(a) and (b), we observe that if the $Y$ value in PE($w-1$) is a partial sum for $x_i$, then that in PE($w-2$) is a partial sum for $x_{i+1}$. Hence, $Y(w)$ and $Z(w)$ contain incompatible partial sums. The partial sum in $Y(w)$ is to be used in the next iteration. $V(w)$ is used to save the previous value of $Y(w)$. Consequently, $V(w)$ and $Z(w)$ contain partial sums for the same $x_i$.

In the second **parallel do** set of statements, either a new term is added to a partial sum $Y(i)$ or a new $x_i$ is computed. PE($w$) computes a new $x_i$ by computing $(V(w) + Z(w))$ and $A(w) * X(w)$ in parallel. The two results are then added (the operations may also be pipelined). Assuming that the time for an addition is no more than that for a multiply, the computation performed in PE($w$) takes the same time as that performed in the other PEs.

Figure 2.2 is a timing diagram for the case $w = 5$ where $j$ refers to the **for** loop index of Algorithm 2.1. For each PE, the contents of its $X$ and $Y$ registers following the execution of the **for** loop for that $j$ value are shown. The notation $[i, p]$ denotes $\sum_{\substack{j=1 \\ j\ odd}}^{p} a_{ij}\ x_{i+j-w-1}$ for PEs on the right of PE($w$) and $\sum_{\substack{j=1 \\ j\ even}}^{p} a_{ij}\ x_{i+j-w-1}$ for PEs on the left. $V(w)$ contains the sum of odd terms (as $w$ is odd), while $Z(w)$ contains the sum of even terms (as $w$ is odd).

The performance figures of this design are $P = w$, $B = w + 1$, $T_C = n + \lceil w/2 \rceil$, $T_D = n + w + 1$, $R_C \sim 1 + w/(2n) \sim 1$, $R_D \sim 1 + w/n \sim 1$ and $R \sim 1$.

```
for $j \leftarrow 1$ to $\lceil (w - 1)/2 \rceil$ do
  do in parallel
    $X(w) \leftarrow x_{j-w}$
    $X(w-1) \leftarrow X(w)$
    $X(i) \leftarrow X(i+2), \quad 1 \le i \le w-2$
  end
end
for $j \leftarrow \lceil (w + 1)/2 \rceil$ to $n + w$ do
  do in parallel
    case
      $j < w+1 : X(w) \leftarrow x_{j-w}$
      $j = w+1 : X(w) \leftarrow x_0$
      $j > w+1$ : output $X(w)$ { output $x_{j-w-1}$ }
    endcase
    $X(w-1) \leftarrow X(w)$
    $X(i) \leftarrow X(i+2), 1 \le i \le w-2$
    $A(w) \leftarrow a_{j-w,w}$
    $A(i) \leftarrow a_{j+\lfloor (w-i)/2 \rfloor + 1 - w, i}, 1 \le i \le w-1$
    $Y(1) \leftarrow Y(2) \leftarrow 0$
    $Y(i) \leftarrow Y(i-2), 3 \le i \le w$
    $V(w) \leftarrow Y(w)$
    $Z(w) \leftarrow Y(w-1)$
  end
  do in parallel
    $Y(i) \leftarrow Y(i) + A(i) * X(i) \; 1 \le i \le w-1$
    $X(w) \leftarrow (V(w)+Z(w))+A(w)*X(w) \quad j \ge w+1$
  end
end
output $X(w)$ { output $x_n$ }
```

**Algorithm 2.1**

| j | PE | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | | 4 | | 5 | | | | 3 | | 1 | |
| | $X$ | $Y$ | $X$ | $Y$ | $X$ | $Y$ | $V$ | $Z$ | $X$ | $Y$ | $X$ | $Y$ |
| 1 | – | – | – | – | $x_{-4}$ | – | – | – | – | – | – | – |
| 2 | – | – | $x_{-4}$ | – | $x_{-3}$ | – | – | – | $x_{-4}$ | – | – | – |
| 3 | $x_{-4}$ | – | $x_{-3}$ | – | $x_{-2}$ | – | – | – | $x_{-3}$ | – | $x_{-4}$ | [1,1] |
| 4 | $x_{-3}$ | [1,2] | $x_{-2}$ | – | $x_{-1}$ | – | – | – | $x_{-2}$ | [1,3] | $x_{-3}$ | [2,1] |
| 5 | $x_{-2}$ | [2,2] | $x_{-1}$ | [1,4] | $x_0$ | [1,3] | – | – | $x_{-1}$ | [2,3] | $x_{-2}$ | [3,1] |
| 6 | $x_{-1}$ | [3,2] | $x_0$ | [2,4] | $x_1$ | [2,3] | [1,3] | [1,4] | $x_0$ | [3,3] | $x_{-1}$ | [4,1] |
| 7 | $x_0$ | [4,2] | $x_1$ | [3,4] | $x_2$ | [3,3] | [2,3] | [2,4] | $x_1$ | [4,3] | $x_0$ | [5,1] |
| 8 | $x_1$ | [5,2] | $x_2$ | [4,4] | $x_3$ | [4,3] | [3,3] | [3,4] | $x_2$ | [5,3] | $x_1$ | [6,1] |
| 9 | $x_2$ | [6,2] | $x_3$ | [5,4] | $x_4$ | [5,3] | [4,3] | [4,4] | $x_3$ | [6,3] | $x_2$ | [7,1] |
| 10 | $x_3$ | [7,2] | $x_4$ | [6,4] | $x_5$ | [6,3] | [5,3] | [5,4] | $x_4$ | [7,3] | $x_3$ | [8,1] |

Figure 2.2 $w = 5$

## 3. Conclusions

We have developed a VLSI system for the adaptive recursive filtering problem that has $T_C$ and $T_D$ that is $o(n)$ and also has $R \sim 1$. Previously, this had been done only for the case of VLSI systems using the broadcast capability. Our design does not employ this capability. The performance characteristics for various VLSI systems for the adaptive recursive filtering problem are summarized in Table 3.1.

| Perf | Architecture | | | | |
|---|---|---|---|---|---|
| | Bidirectional | | | | Systolic |
| | Broadcast Chain | Chain | | | Ring |
| | [HUAN82] | [KUNG78] | [ROBE84] | Our | [KUNG84] |
| $P$ | $w$ | $\lceil w/2 \rceil$ | $2w$ | $w$ | $\lceil w/2 \rceil$ |
| $B$ | $w + 2$ | $\lceil w/2 \rceil + 2$ | $2w$ | $w + 1$ | $\lceil w/2 \rceil + 1$ |
| $T_C$ | $n + w$ | $2n + w$ | $n + \lceil w/2 \rceil$ | $n + \lceil w/2 \rceil$ | $2(n - 1) + w$ |
| $T_D$ | $n + w$ | $2(n + w - 1)$ | $n + w$ | $n + w$ | $2(n + w - 1)$ |
| $R_C$ | 1 | 1 | 2 | 1 | 1 |
| $R_D$ | 1 | 1 | 2 | 1 | 1 |
| $R$ | 1 | 1 | 4 | 1 | 1 |

$C = nw, D = nw + n + w$

Table 3.1

Further improvement in throughput (at the expense of design complexity) is possible. However, this cannot be obtained using recurrence (1) as in order to compute $x_i$, we need to know $x_{i-1}$. Hence $x_i$ can be computed, at best, one cycle after $x_{i-1}$ has been computed. However, we can bring both $T_C$ and $T_D$ down to $o(n/2)$ by computing two $x_i$s each cycle using recurrence (2) and formulae (3) and (4). The idea is the same as used in the back substitution problem in [CHEN84b]. The VLSI system that incorporates this uses more hardware and is quite a bit more complex. The method may be extended to get a $T_C$ and $T_D$ of $o(n/k)$ for any fixed $k$.

## 4. References

[CHEN84a]K.H. Cheng and S. Sahni, *VLSI Systems For Matrix Multiplication,* Department of Computer Science, University of Minnesota, **August 1984.**

[CHEN84b]K.H. Cheng and S. Sahni, *VLSI Architectures For Back Substitution,* Department of Computer Science, University of Minnesota, **November 1984.**

[CHEN85]K.H. Cheng and S. Sahni, *VLSI Architectures For LU Decomposition,* Department of Computer Science, University of Minnesota, **January 1985.**

[HUAN82]K.H. Huang and J.A. Abraham, *Efficient parallel algorithms for processor arrays,* IEEE International Conference On Parallel Processing, **1982,** pp. 271-279.

[KUNG78]H.T. Kung and C.E. Leiserson, *Systolic arrays for VLSI,* Department of Computer Science, Carnegie-Mellon University, **April 1978.**

[KUNG83]H.T. Kung, *A Listing of Systolic Papers,* Department of Computer Science, Carnegie-Mellon University, **May 1984.**

[KUNG84]H.T. Kung and M. Lam, *Wafer scale integration and two level pipelined implementations of systolic arrays,* Journal of Parallel and Distributed Processing, Vol. 1, #1, **1984.**

[LEIS83]C.E. Leiserson, *Area-Efficient VLSI Computation,* MIT Press, **1983.**

[ROBE84]Y. Robert and M. Tchuente, *Designing Efficient Systolic Algorithms,* Laboratoire IMAG, BP 68, F38402 Saint Martin D'Heres Cedex, **1984.**

# RECBAR: A RECONFIGURABLE MASSIVELY PARALLEL PROCESSING ARCHITECTURE

*Vijay Balasubramanian and Prithviraj Banerjee*

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL-61801

## ABSTRACT

This paper presents two massively parallel processing architectures suitable for solving a wide variety of divide-and-conquer type algorithms for problems such as the Discrete Fourier Transform, Production Systems, Design Automation and others. The first architecture, called the Chain-structured Butterfly ARchitecture (CBAR), consists of a two-dimensional array of $N = L.(log_2(L) + 1)$ processing elements (PE) organized as $L$ levels of $log_2(L) + 1$ stages, and which has the butterfly connection between PEs in consecutive stages with straight-through feedback between PEs in the last and first stages. This connection system has the desirable property of allowing thousands of PEs to be connected with $O(N)$ connection cost, $O(log_2(\frac{N}{log_2 N}))$ communication paths and a small number (=4) of I/O ports per PE. However, this architecture is not fault tolerant. We, therefore, propose a second architecture, called the REconfigurable Chain-structured Butterfly ARchitecture (RECBAR), which is a modified version of the CBAR. The RECBAR possesses all the desirable features of the CBAR, with the number of I/O ports per PE increased to six, and uses $O(\frac{log_2 N}{N})$ overhead in PEs and approximately 50% overhead in links to achieve single-level fault tolerance. Reliability improvements of the RECBAR over the CBAR are studied.

## 1. INTRODUCTION

Recent developments in technology have made it possible to interconnect a large number of processing elements in order to form an integrated system. Various network architectures have been proposed that are suitable for both multiprocessors and VLSI systems [1, 2]. In this paper, we present a massively parallel processing architecture suitable for solving a wide variety of divide-and-conquer type algorithms for problems in signal processing, production systems, design automation and others. This architecture, called the Chain-structured Butterfly ARchitecture (CBAR), has the desirable property of allowing thousands of PEs to be connected with $O(N)$ connection cost, $O(log(\frac{N}{log N}))^*$ communication paths and a small number (=4) of I/O ports per PE.

One attribute that is desirable in any complex parallel processing system but missing in the CBAR is fault-tolerance. Fault tolerant network architectures are emerging as an important area of study [3, 4, 5]. We, therefore, propose a second architecture, called the REconfigurable Chain-structured Butterfly ARchitecture (RECBAR), which is a modified version of the CBAR. The RECBAR possesses all the desirable features of the CBAR, with the number of I/O ports per PE increased to six, and uses $O(\frac{log N}{N})$ overhead in PEs and approximately 50% overhead in links to achieve fault tolerance.

## 2. CHAIN-STRUCTURED BUTTERFLY ARCHITECTURE

The chain-structured butterfly architecture (CBAR) is a novel parallel processing architecture which has processors in place of exchange switches in a regular butterfly interconnection network. In addition, the output ports of the processors in the last stage of the CBAR are fed back to the of the first stage. The network resembles the Loop Structured Switching Network proposed by Wong and Ito [6] except that there is no unshuffle while traversing the feedback path. The organization of a processing element (PE) is similar to that in the MAN-YO Architecture [7] ; each PE consists of the actual processor and a router cell, the latter managing packet transmission among different PEs. The router cell is basically a store and forward crossbar switch with three input and three output ports, one input- output pair being solely used for the processor, and the remaining two port pairs being network ports, used for connection among router cells.

### 2.1. Network Topology

The CBAR is a two-dimensional array of $N = L.(log\ (L) + 1)$ PEs arranged in $L$ levels and $S = log\ (L) + 1$ stages. The following description can be easily understood if the reader refers to Fig. 1, which shows the network for $N = 32$ and $L = 8$. The stages are labeled in a sequence from 0 to $log\ (L)$ with 0 for the leftmost stage and $log\ (L)$ for the rightmost stage. Similarly the PE levels are labeled in a sequence from 0 to $L - 1$. Each PE can be uniquely identified by the stage and level to which it belongs: $<i, j>$ represents a PE in the $i^{th}$ stage and $j^{th}$ level. The integers $i$ and $j$ can be represented by their binary equivalents with $s = log\ (S)$ and $l = log\ (L)$ bits, respectively. Hence the processor $<i, j>$ has a binary address $[q_{s-1} \cdots q_0, p_{l-1} \cdots p_0]$. An output link of a PE is referred to as the '0 link' if it connected to the upper output port, and a '1 link' if it is connected to the lower output port.

The topology describing rules of the network are defined as follows:
(1) For link 0: CBAR $<i, j> = <i + 1, j - 2^i .bit_i(j)>$,
(2) For link 1: CBAR $<i, j> = <i + 1, j + 2^i .[1 - bit_i(j)]>$,
for $0 \leqslant i \leqslant log\ (L) - 1, 0 \leqslant j \leqslant L - 1$,
where $bit_i(j)$ equals the bit with weight $2^i$ in the binary representation of $j$.
(3) For the feedback links, 0 and 1:
CBAR $<log\ (L), j> = <0, j>$, for $0 \leqslant j \leqslant L - 1$.

The labeling scheme and interconnection of the PEs can be verified in the example shown in Fig. 1.

**Acknowledgement :** This research was supported by the Semiconductor Research Corporation under contract SRC RSCH 84-06-049.

[*] In this paper, we denote *log* to be the logarithm base 2.

### 2.2. Properties of the CBAR

This section states some useful results concerning the behavior of the CBAR for a single packet being routed from a source PE to a destination PE. We define a 'step' in packet routing as the transfer of a packet from a (network) input port buffer or the processor output port buffer of a PE router cell to a (network) input port buffer of the next PE router cell. This involves setting of the router cell to use either the 0 or the 1 link of the PE. The process of choosing the 0 or the 1 link of a PE in the $i^{th}$ stage can be viewed as modifying the $i^{th}$ bit of the address of the level in which the packet currently resides. For the remaining part of this section we consider a CBAR which has $L$ levels and a packet which is generated at the PE $<i, p_{l-1} \cdots p_1 p_0>$ and destined for the PE $<i', p'_{l-1} \cdots p'_1 p'_0>$, where $l = log\ (L)$, and $0 \leqslant i \leqslant l$.

Given two bit strings, $STR1$ and $STR2$, the maximum length common suffix of $STR1$ and $STR2$ will be denoted as $MLCS(STR1, STR2)$. Given a bit string $STR$, a cyclic shift left by k bits followed by a bit reversal will be denoted by $CSLR_k(STR)$. The length of a string $STR$ is denoted by $|STR|$. The Augmented Source Bit String ($ASBS$) and the Augmented Destination Bit String ($ADBS$) are defined as $(p_l p_{l-1} \ldots p_0)$ and $(p'_l p'_{l-1} \ldots p'_0)$, respectively, where $p_l$ and $p'_l$ are extra bits that are added to model the feedback connection of the CBAR network. The extra bits are equal to each other and can be assigned either 0 or 1.

It can be shown that the packet will be routed to the destination PE in exactly $k + [(i' - i - k)\ modulo\ (l + 1)]$ steps after its generation at the source PE, where $k = l + 1 - |MLCS\ (CSLR_{l-i+1}(ASBS), CSLR_{l-i+1}(ASBD))|$. It follows that in a CBAR with $L$ levels, a packet will be delivered to its destination within $2.log\ (L) + 1$ steps of routing regardless of where it is generated. This allows us to conclude that the CBAR network has $O\ (log\ (\frac{N}{log\ N}))$ communication between PEs.

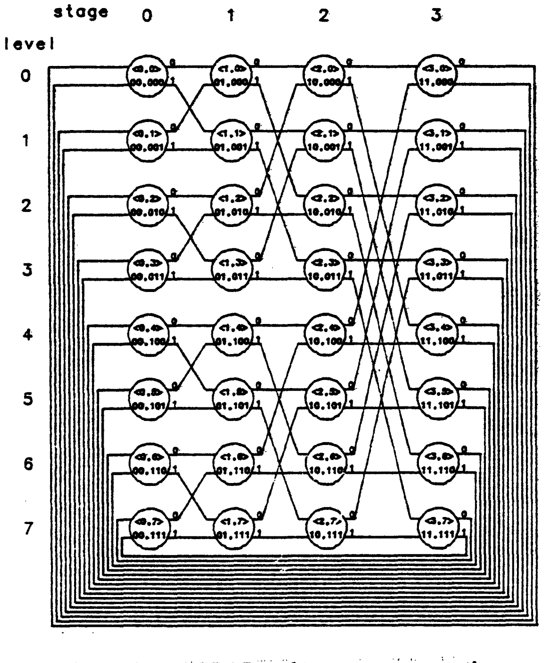


**Fig. 1.** *A 8×4 CBAR with name representation*

### 3. THE RECONFIGURABLE-CHAIN STRUCTURED BUTTERFLY ARCHITECTURE

We now propose a modified version of the CBAR which we call the reconfigurable chain-structured butterfly architecture (RECBAR).

### 3.1. Network Topology

The RECBAR can be formally described as follows. The network consists of $N = (L + 1).(log\ (L) + 1)$ PEs arranged in $L + 1$ levels and $log\ (L) + 1$ stages. The stages are numbered in a sequence from 0 to $log\ (L)$, with 0 for the leftmost stage and $log\ (L)$ for the rightmost stage. Similarly, the levels are labeled in a sequence from 0 to $L$. Each PE can be uniquely identified as $<i, j>$, where $i$ denotes the stage and $j$, the level to which the PE belongs. Each PE has three input and output ports except those in the first and last stages which have two input and three output ports, and three input and two output ports, respectively. An 'upper link' is attached to the upper output port of a PE, a 'middle link' to the middle output port and a'lower link' to the lower output port of the PE.

The topology describing rules of the network are as follows:

(1) For an upper link:
RECBAR $<i, j> = <i + 1, (j - 2^i)\ modulo\ (L + 1)>$,
(2) For a middle link: RECBAR $<i, j> = <i + 1, j>$,
(3) For a lower link:
RECBAR $<i, j> = <i + 1, (j + 2^i)\ modulo\ (L + 1)>$,
for $0 \leqslant i \leqslant log\ (L) - 1, 0 \leqslant j \leqslant L$.
(4) The feedback links:
RECBAR $<log\ (L), j> = <0, j>$, for $0 \leqslant j \leqslant L$

The reader may verify the PE interconnections in the example of Fig. 2. It should be noted at this point, that the RECBAR network resembles the Inverse Augmented Data Manipulator

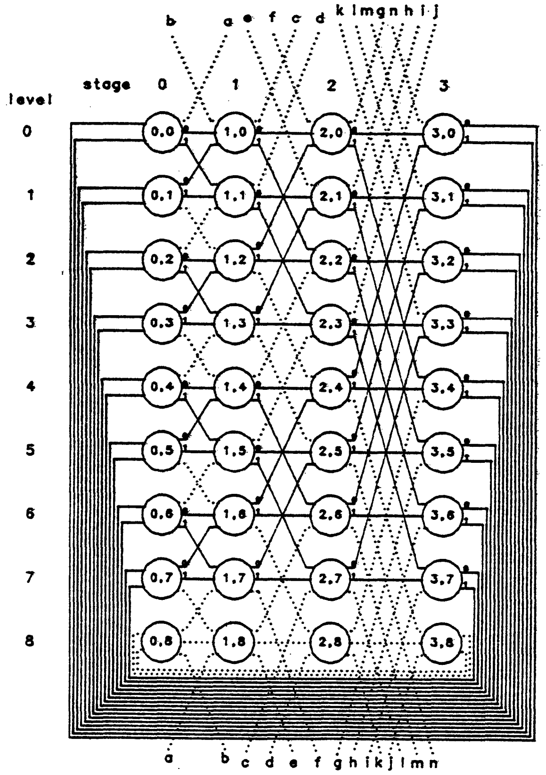


**Fig. 2.** *A 9×4 RECBAR with name representation*

Network proposed by McMillen and Siegel [8], with the following three differences. First, the exchange switches in the IADM are replaced by processing elements. Secondly, in the IADM there were $L$ levels, whereas we have $L+1$ levels in the RECBAR. Thirdly, we use only $L$ of the levels during operation, the extra level being included just for reconfiguration, whereas in the IADM, all the $L$ levels were used simultaneously and there were no ideas of reconfiguration.

### 3.2. The RECBAR operation under no fault

We will show that the RECBAR can emulate the CBAR under no fault conditions. The level of PEs labeled $L$ is not considered so that we have $L.(log\ (L)+1)$ PEs arranged in $L$ levels and $log\ (L)+1$ stages, as required by the CBAR. Now any PE $<i,j>$ $(i \neq log\ (L))$, in the CBAR is connected to PEs $<i+1, j>$, and $<i+1, j-2^i>$ *or* $<i+1, j+2^i>$ depending on the value of the $i^{th}$ bit of the binary word corresponding to $j$. In the RECBAR, we have the PE $<i,j>$ connected to PEs $<i+1, j>$, $<i+1, j-2^i>$ *and* $<i+1, j+2^i>$, the modulo operation coming into effect only if $j-2^i$ is negative or $j+2^i$ exceeds $L+1$, which would not be the case for the CBAR addressing scheme. The feedback links remain the same in both the networks. All one needs to do is to select the two proper output links out of the three available, in each PE (except for the last stage), and the RECBAR would operate as the CBAR.

The output link selection is based on the CBAR topology rules developed in Section 2.1. Address each PE $<i,j>$ $(i \neq log\ (L))$, by the binary representations of the stage and level to which it belongs. If the $i^{th}$ bit of the level address is a 0, choose the lower two output links, assigning a 0 to the middle link and a 1 to the lower link. If the $i^{th}$ bit is a 1, choose the upper two output links, assigning a 0 to the upper link and a 1 to the middle link. The significance of the 0 and the 1 links is the same as in the CBAR. Fig. 2. shows the RECBAR configured for operation under no fault. The continuous lines denote the links in use and the dotted lines denote the redundant links.

### 3.3. The RECBAR operation under a single fault

We will show that the RECBAR can emulate the CBAR in the presence of faults in a single PE level.

THEOREM 1: The RECBAR can be reconfigured to emulate the CBAR under failure of any single level.

PROOF: Suppose a level $r$ of PEs fails for some $0 \leqslant r \leqslant L-1$. We will show that it is possible to reconfigure the RECBAR such that the level $r$ is removed and the spare level $L$ brought instead. We will transform the addresses of the PEs by the following simple transformation: the PE whose old address was $<i,j>$ will be assigned the new address $(i, j-r-1)$, where the subtractions are performed *modulo* $(L+1)$. For example, the PE whose old address was $<i, r+1>$ is assigned the new address $(i,0)$ and the PE whose old address was $<i, r-1>$ is assigned the new address $(i, L-1)$, since $(r-1-r-1)$ *modulo* $(L+1)$ $=(-2)$ *modulo* $(L+1)=L-1$.

In order that the CBAR structure be realized, we need to have the following connections for the PEs in the new addressing scheme:

(1) $<i,j>$ to $<i+1, j>$,
(2) $<i,j>$ to $<i+1, j-2^i>$, if $bit_i(j)=1$,
(3) $<i,j>$ to $<i+1, j+2^i>$, if $bit_i(j)=0$,
(4) $<log\ (L), j>$ to $<0,j>$,
for $0 \leqslant i \leqslant log\ (L)-1$, $0 \leqslant j \leqslant L-1$.

This implies that the original network should have the following connections:

(1) $<i, (j+r+1)\ mod\ (L+1)>$
to $<i+1, (j+r+1)\ mod\ (L+1)>$,
(2) $<i, (j+r+1)\ mod\ (L+1)>$
to $<i+1, (j+r+1)\ mod\ (L+1) \pm 2^i>$.[*]
(3) $<log\ (L), (j+r+1)\ mod\ (L+1)>$
to $<0, (j+r+1)\ mod\ (L+1)>$,
for $0 \leqslant i \leqslant log\ (L)-1$, $0 \leqslant [(j+r+1)\ mod\ (L+1)] \leqslant L-1$.

From the definition of the RECBAR, these connections are present, where $j$ is replaced by $(j+r+1)\ mod\ (L+1)$. □

EXAMPLE 1: Consider a RECBAR with $L=8$. Let level 3 become faulty. We rename level 4 as 0, level 5 as 1,..., level 8 as 4, level 0 as 5,..., and level 2 as 7. Let us see whether we have proper connections for the PE <2,2> in the new addressing scheme. We need the connections <2,2> to <3,2> and <3,6>. These correspond to <2,6> to <3,6> and <3,1> in the original network, since $(j+r+1)\ mod\ (L+1)$ $=(j+4)\ mod\ 9$, which is equal to 6 for $j=2$ and is equal to 1 for $j=6$. We notice that these connections are present in the RECBAR of Fig. 2. Fig. 3 illustrates the reconfigured RECBAR for a fault in level 3.

The selection of the proper outputs ports in each PE is straightforward. Let level $r$ be the faulty level, for some $0 \leqslant r \leqslant L-1$. Compute $(j-r-1)\ mod\ (L+1)$ for each PE $<i,j>$ and represent the value obtained by its binary equivalent. Bit $i$ of this binary word then determines the ports to be selected, the selection process being similar to the one discussed in Section 3.2. The reader may verify the port selection on the example of Fig. 3.

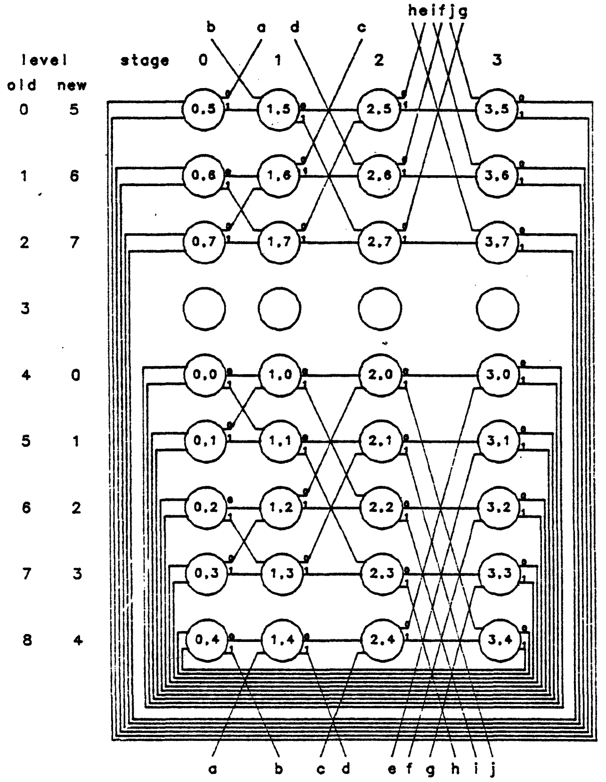


Fig. 3. *A reconfigured 9×4 RECBAR for a fault in level 3*

[*] The addition or subtraction of $2^i$ depends on $bit_i[(j+r+1)\ mod\ (L+1)]$.

### 3.4. Structure of the Router Cell

The router cell has three network input-output port pairs and one processor input-output port pair, as shown in Fig. 4. A 4×4 crossbar switch connects the input ports to the output ports. The output port set selection logic is shown in outline mode. It consists of two 1-to-2 demultiplexers controlled by a common port set selection line. The operation of the router cell is clear from the figure itself and need not be explained.

### 3.5. Overhead

Consider a CBAR with $L$ levels. It has $N = L.(log\ (L)+1)$ PEs and $2N$ links. The corresponding RECBAR has $N' = (L+1).(log\ (L)+1)$ PEs and $3N' - (L+1) = (L+1).(3.log\ (L)+2)$ links. The PE overhead ratio comes out to be equal to $\frac{1}{L}$ and the link overhead ratio comes out to be equal to $\frac{(L+3).log\ (L)+2}{2L.(log\ (L)+1)}$. In the limit as $L$ tends to a large value, the link overhead ratio becomes nearly 50%.

## 4. RELIABILITY ANALYSIS

We will now estimate the improvement in the reliability of the system topology. We assume that the failure rate of a PE is exponential with a failure rate of $\lambda_p$ and the failure rate of a link to be $\lambda_l$. In Fig. 5, we show the reliabilities of the CBAR and the RECBAR for $L = 8$, $\lambda_p = 0.1$ failures per unit time and $\lambda_l = 0.01$ failures per unit time. We assume that the failure rate of a link is much less than that of a PE because it is much less complex.

## 5. CONCLUSIONS

In this paper, we have presented a chain-structured butterfly architecture (CBAR) similar in organization to the architectures proposed by Wong and Ito [6] and Koike and Ohmori [7]. The CBAR has the desirable property of being able to connect an extremely large number of processors with $O(N)$ connection cost, $O\ (log\ (\frac{N}{log\ N}))$ communication paths, and a small number (=4) of I/O ports per PE. We have also discussed a reconfigurable version of the CBAR, the RECBAR, which can tolerate any number of failures of processing elements and links associated with a single level. This fault tolerance is achieved at a processor overhead ratio of $O(\frac{log\ N}{N})$ and a link overhead ratio of approximately 50%. The reliability analysis has shown that the RECBAR is much more reliable than the CBAR.

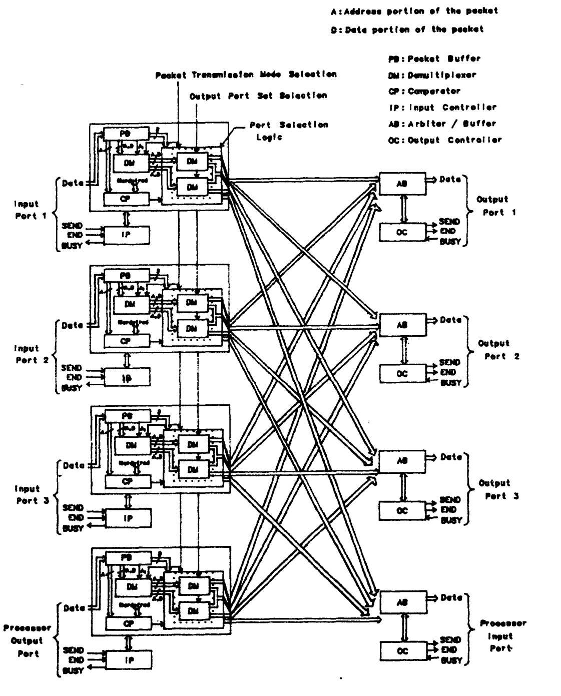


Fig. 4. *The RECBAR router cell block diagram*

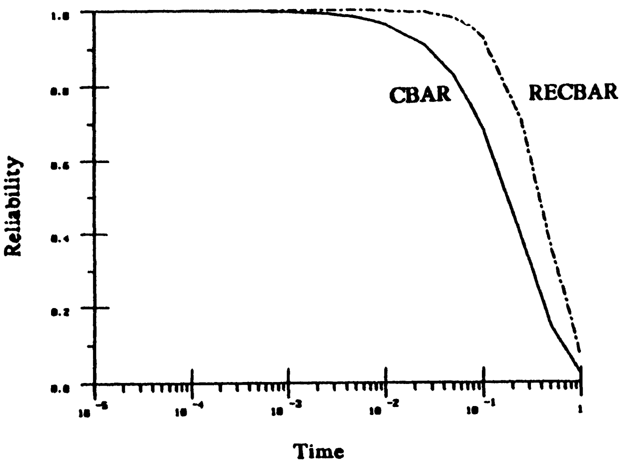


Fig. 5. *Reliability comparisons between CBAR and RECBAR*

## REFERENCES

[1] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," in *IEEE Tran. Computers*. pp. 458-473, May 1977.

[2] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network For Parallel Computation," *Commun. Ass. Comput. Mach.*, vol. 24, pp. 300-309, May 1981.

[3] I. Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," in *Proc. 8th Int. Symp. on Computer Architecture*. Minneapolis, Minnesota, pp. 425-442, May 1981.

[4] C. S. Raghavendra, A. Avizienis, and M. Ercegovac, "Fault-Tolerance in Binary Tree Architectures," in *Proc. 13th Int. Symp. on Fault-Tolerant Computing*. pp. 360-364, Jun. 1983.

[5] D. K. Pradhan, "Fault-Tolerant Multiprocessor Link and Bus Network Architectures," in *IEEE Trans. Computers*. pp. 33-45, Jan. 1985.

[6] F. S. Wong and M. R. Ito, "A Loop Structured Switching Network," *IEEE Trans. Computers*, vol. C-33, pp. 450-455, May 1984.

[7] N. Koike and K. Ohmori, "MAN-YO : A Special Purpose Parallel Machine for Logic Design Automation," *1985 Int. Conf. on Parallel Processing*, pp. 583-590, Aug. 1985.

[8] R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulator Network in an MIMD System," *IEEE Trans. Computers*, vol. C-31, pp. 184-196, Dec. 1982.

# SYNTHESIS AND MAPPING ALGORITHMS FOR A RECONFIGURABLE OPTICAL INTERCONNECTION NETWORK

*Insup Lee*
*Sam Goldwasser*
*David Smitley*

General Robotics and Active Sensory Processing Group
Department of Computer and Information Science
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, PA 19104

## Abstract

The performance of a parallel algorithm depends on the interconnection topology of the target parallel system. An interconnection network is called reconfigurable if its topology can be changed between different algorithm executions. Since communication patterns vary from one parallel algorithm to another, a reconfigurable network can effectively support algorithms with different communication requirements. This paper describes a reconfigurable optical network and explains how to generate network topologies which are optimized with respect to a given task. We describe an algorithm that takes as input a task graph and generates as output a topology that closely matches the given input graph. The best, worst and average case performance of the algorithm is analyzed and it is shown that on the average the optimum topology is generated.

## 1. Introduction

The performance of a parallel algorithm depends on how well its communication characteristics match the interconnection topology of the underlying parallel system. Since different algorithms exhibit different communication patterns, work in the area of general-purpose parallel system design centers on developing a network topology that is suitable for various communication requirements. A common approach is to statically connect processing elements in a regular pattern with certain properties such as small diameter, easy routing and expansion, low congestion, etc. General-purpose parallel systems with static interconnection networks however have several limitations due to their fixed nature. The first limitation is that a given algorithm, which may be viewed as a graph with nodes representing processes and edges representing potential communications, has to be mapped onto the parallel system. This mapping problem in general is computationally intractable even for parallel systems with homogeneous processing elements. The second limitation is that although one interconnection topology may be ideal for a set of algorithms, it may introduce unacceptable communication delays for other algorithms even with the best possible mapping. The third limitation is that developing parallel algorithms that closely match the interconnection topology of a target parallel system is a difficult task. A parallel system with a network whose interconnection topology can be configured to match the communication characteristics of an algorithm remedies these limitations.

An interconnection network is called *reconfigurable* if its topology can be altered between different algorithm executions or even between different phases of the same algorithm execution. Examples of architectures that can be configured into a limited number of interconnection patterns include the MPP [1] and the CHiP [2] architecture. In contrast to these networks which can only realize a limited number of topologies, this paper will investigate the use of a reconfigurable network which can realize any $r$-regular graph as a network topology. Such networks are referred as $r$-reconfigurable networks.

## 2. Reconfigurable Optical Network

Optics provides a way of implementing $r$-reconfigurable networks. We propose using an optical network that requires only $O(nr)$ components to implement a $r$-reconfigurable network for $n$ processors, where $n$ is ~ 100. The network consists of $nr$ optical transmitters (laser diodes), $nr$ deflectors (acousto-optic devices or mirrors) and $nr$ photo-sensitive receivers (photodiodes). The deflectors can be either mirrors mounted on servo motors or acousto-optic devices which deflect an incoming beam at an angle proportional to an applied frequency. To establish a communication channel between processors, the source processor directs a control signal to its deflector. The control signal's value determines the deflection angle of the incoming laser beam so that it impinges on the photodiode associated with the desired processor. The transmitting processor then modulates the laser beam with the information to be transmitted. The receiver detects this light beam, demodulates it, and processes the resultant data accordingly. Since laser beams do not interfere with each other,

0190-3918/86/0000/0394 $01.00 © 1986 IEEE

the network can realize any permutation. Other researchers have given alternate designs of optical reconfigurable networks [3].

In the design of our reconfigurable network, there is a tradeoff between the cost of the network and the time needed to reconfigure. For example, if mirrors are used as deflection elements, the network has a reconfiguration time of ~ 1 *msec*; whereas, if an acousto-optic deflector is used, the network has a reconfiguration time of ~ 1 *μsec*. The two networks differ in cost by at least an order of magnitude due to the cost of the acousto-optic deflector and related control electronics. In this paper, we consider the use of the network with the slower reconfiguration time. In particular, the time needed to make the transition from one configuration to another is long in comparison to the time between two communication events. Therefore, we assume that the topology changes only between different algorithm executions or between different phases of the same algorithm execution.

Given that the topology remains relatively static throughout the execution of an algorithm, the topology must be chosen so as to closely match the communication requirements of the algorithm. In the next section we describe an algorithm that takes as input a task graph and generates as output a topology that closely matches the given input graph. In section 4, we briefly analyze the best, worst and average case performance of the algorithm.

## 3. The Synthesis Problem

We now discuss the problem of generating an interconnection topology that matches closely the interconnection patterns of a given task graph. We define an optimum topology (and corresponding mapping) as one that maximizes the number of pairs of communicating tasks that fall on pairs of directly connected processors. The definition of the synthesis-mapping function can be stated as follows: Given a connected and undirected graph $T$, $|V(T)| = n$ such that each element of $V(T)$ is labeled $t_i$, $1 \le i \le n$, and a set $V(P)$ of $n$ nodes labeled $v_i$, $1 \le i \le n$, and a function $g: V(T) \rightarrow V(P)$, where $g(t_i) = v_i$, and an integer $r \ge 2$, then the problem is to find a function $c: V(P) \times V(P) \rightarrow \{0,1\}$ such that $P = \{V(P), E(P)\}$, where $E(P) = \{(u,v): c(u,v) = 1\}$, is a degree bounded connected graph with $degree(v_i) \le r$ for all $v_i \in V_{P,}$ $1 \le i \le n$, and *cardinality* =

$$|\{(u,v): (u,v) \in E(T) \; and \; (g(u), g(v)) \in E(P)\}|$$

is maximum.

$T$ is a task graph that models the algorithm to be executed on the system. The vertices in the graph correspond to individual tasks and an edge between two vertices signifies that communication occurs between these two tasks. No attempt has been made to quantize the amount or cost of communication between tasks. The processor system is also represented as a graph with vertices corresponding to processors and edges to communication links. The problem is to find the set of edges (communication links) to maximize the number of pairs of intercommunicating tasks that fall on pairs of directly connected processors. At the same time, because of the limited number of links each processor controls, the degree of each vertex must be bound. Also, the resultant topology must be connected since a message sent to a processor that is not directly connected to a source processor must be forwarded through intermediate processors. Graphs meeting these conditions will be referred to as degree constrained connected graphs (DCCG). The problem of finding such graphs will be referred to as the DCCG problem.

We have shown in [4] that the DCCG problem is NP-complete. Although the DCCG problem is NP-complete, the synthesis problem can be solved in polynomial time using graph matching methods if the condition that the resultant graph be connected is removed [5]. The outline of an algorithm [4] for finding sub-optimal DCCGs is as follows: First, use a matching technique to find a optimal disconnected topology (henceforth referred to as a maximal deficient $r$-factor or MDRF). Then, using heuristics, connect the disconnected components of the MDRF together. The connection of disconnected components may require that existing communication links interconnecting communicating tasks be broken. When links are broken, our algorithm ensures that further discontinuities are not introduced. The algorithm runs in the time complexity of $O(n^3)$.

## 4. Analysis of the Synthesis Algorithm

In the best case, our algorithm can generate topologies with a cardinality of $\lfloor nr/2 \rfloor$. In the worst case, we can bound the cardinality with respect to the optimum cardinality. To do this we note that the cardinality of the MDRF is at least as large as the cardinality of the optimum connected topology. Therefore, we can use the cardinality of the MDRF as a bound on the optimum cardinality. It is shown elsewhere [4] that use of our algorithm to connect the disconnected components of the MDRF reduces the cardinality by at most $\lfloor n/(r+1) \rfloor - 1$. Therefore, our algorithm always generates a connected topology that is within $\lfloor n/(r+1) \rfloor - 1$ of optimal. Furthermore, for any given $n$ and $r$, it is possible to construct a graph with a MDRF such that the connection of the MDRF by any connection algorithm results in a cardinality loss by the bound. That is, the performance of our connection algorithm equals that of an optimal algorithm in the worst case.

We use random graphs as processor graphs to analyze the average case. Random graphs are used since we want to determine the performance of our algorithm over a wide range of graphs. We define a random graph as a graph where the probability of an edge between two nodes is fixed. We say that almost every random graph has a given property if the probability of a random graph having that property approaches 1 as the number of nodes in the graph approaches infinity [6]. To apply well-known results from random graph theory in analyzing our algorithm for the average case, we assume that the topology synthesized by our algorithm can be modeled as a random $r$-regular graph. Since we are investigating the performance of our algorithm assuming a random graph input and since our algorithm generates a $r$-regular graph as output by deleting some of the edges of the input graph, this is a reasonable assumption to make.

To determine the average case performance, we need to determine the expected cardinality of a MDRF and also the expected reduction in cardinality due to the connection process. The cardinality of a MDRF has been characterized by Shamir and Upfal [7] who state that if almost every random graph $G$, with $nr$ even, has a minimal degree of $r$, then almost every graph $G$ has a MDRF with a cardinality of $nr/2$. Thus, for almost every random graph used as input with a minimal degree $\geq r$, a MDRF will be found with $nr/2$ edges.

Our algorithm diverges from optimality during the connection process. In particular, the algorithm does not connect the disconnected components so that the cardinality is reduced by a minimal amount. However, as we mentioned above, this reduction is $\leq \lfloor n/(r+1) \rfloor -1$. On the average, one would expect that the reduction would be smaller than $\lfloor n/(r+1) \rfloor -1$. Indeed, Wormwald [8] has shown that if $r \geq 3$ then almost every random $r$-regular graph is $r$-connected. Since almost every graph generated by the matching process is a $r$-regular graph, almost every graph generated by the graph factoring step of our algorithm will be $r$-connected and hence 1-connected. Therefore, almost every graph with minimal degree of $r \geq 3$ will have a cardinality of $\lfloor nr/2 \rfloor$ when mapped onto the topology generated by our algorithm.

To verify the average case prediction, we generated a range of random graphs with a varying number of nodes and edges. For each of the graphs we found a corresponding MDRF and then used our algorithm to connect the MDRF. It was observed that for almost all of the graphs the corresponding DCCG's had a cardinality of $\lfloor nr/2 \rfloor$. The only situation where the average case prediction failed occurred when the task graphs approached a completely connected graph. In this situation the matching algorithm that we were using to generate the MDRFs was producing MDRFs with relatively large numbers of disconnected components. This occurred since the algorithm used examines the edges adjacent to a node in sequential order in an effort to determine which ones to keep in the DCS. For example, let $r = 3$. The algorithm first examines node 1. It sees that there are edges (1,2),(1,3),(1,4) and adds them to the degree constrained subgraph (DCS). Similarly, it sees that there are edges (2,3), (2,4) and (3,4). Since it examines edges sequentially, it adds these edges to the DCS and thereby forms a component of nodes 1,2,3 and 4 to which no further edges can connect. This does not happen until the graph is almost completely connected since the probability of having all of the needed edges to form a component with consecutive nodes is low until the graph becomes almost completely connected. To avoid this problem the algorithm needs to be modified so that it examines the edges of each node in random as opposed to sequential order.

## 5. Summary

We have presented an optical reconfiguration network that requires only $O(nr)$ components. For this network to achieve better performance than that available from conventional static networks, the topology chosen must *match* that of the algorithm to be executed. We define what it means to *match* and mention an algorithm that takes as input an arbitrary task graph and generates as output a topology that closely matches the given input graph. We then analyze the best, worst and average case behavior of our algorithm. It is shown that on the average the algorithm almost always produces optimum topologies.

## Acknowledgements

This research was supported in part by NSF DCR 8501482, NSF MCS-8219196-CER, and ARO DAA6-29-84-k-0061.

## References

[1] K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, pp. 836-840 (Sept. 1980).

[2] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, pp. 47-56 (Jan. 1982).

[3] A.A. Sawchuk, B.K. Jenkins, C.S. Raghavendra, and A. Varma, "Optical Interconnection Networks," *Proc. 1985 Parallel Processing Conference*, pp. 388-392 (Aug. 1985).

[4] I. Lee and D. Smitley, "A Synthesis Algorithm for Reconfigurable Interconnection Networks," *Submitted to IEEE Trans. on Computers* (June 1986).

[5] J. Edmonds and E.L. Johnson, "Matching: A Well-Solved Class of Integer Linear Programs," *Combinatorical Structures and Their Applications, Proc. of the Calgary Inter. Conf. on Comb. Struct. and Their App.*, pp. 89-92, Gordon and Breach, Science Publishers (June, 1969).

[6] B. Bollobas, *Graph Theory, An Introductory Course*, Springer-Verlag, New York, Heidelberg, Berlin (1979).

[7] E. Shamir and E. Upfal, "On Factors in Random Graphs," *Israel Journal of Mathematics* **39**(4), pp. 297-302 (1981).

[8] N.C. Wormwald, "The Asymptotic Connectivity of Labelled Regular Graphs," *Journal of Combinatorial Theory* **Series B 31**, pp. 156-167 (1981).

# DESIGN OF VLSI ASYNCHRONOUS FIFO QUEUES FOR PACKET COMMUNICATION NETWORKS

Tam-Anh Chu
Department of EECS, M.I.T.
Cambridge MA 02139

Clement K.C. Leung
Department of CS, UT at Austin
Austin TX 78712

## Abstract

Throughput and latency in packet communication networks are determined to a large extent by throughput and latency in the first-in first-out (FIFO) queues used for packet buffering in these networks. We describe a design approach for self-timed FIFO queues with a novel organization which allows tradeoffs between area, throughput and latency in VLSI implementations. This flexibility is made possible by the use of asynchronous distributed control circuits. These circuits are synthesized directly from a graph model called Signal Transition Graphs, and are completely hazard-free. A number of nMOS test chips were fabricated and they worked at a 4 Mbytes/sec throughput rate.

## 1. Introduction

Multistage interconnection networks (MIN) are used to support communication among processing and storage modules in multiprocessor architectures [2]. To use a packet routing MIN, modules communicate by sending packets to each other. A packet consists of an address and data. The address is used to forward the packet along a path through the network from the sender to the destination module. Multistage interconnection networks are usually constructed out of $N \times N$ crossbar switches, where the number of ports $N$ is determined by performance requirements and packaging constraints. A switch forwards an input packet to an output port according to the destination address of the packet. Each switch may also provide buffering storage for packets whose forwarding paths are temporarily blocked by other network traffic. The buffering storage is usually managed by a first-in first-out discipline.

In this setting, the design of FIFO queues is an important consideration in the design of practical packet routing networks. We present an approach for designing FIFO queues in VLSI technology which allows tradeoffs between area, latency and throughput. At one end of the design spectrum, an area-efficient implementation with high throughput, but long latency, can be obtained. In this organization, the register stages are connected serially to allow data to ripple through; there is no global communication. At the other end of the spectrum is a queue with minimal latency, but somewhat lower throughput rate due to increased delay in the control operations and in the loading of global buses. On a VLSI chip, these buses are sets of wires carrying input and output data which are connected to the input and output of all register stages. An optimal design somewhere in the range of these extremes can be chosen depending on the application. A queue of $N$ stages can be partitioned such that only $M$ register stages load the buses; given that the permissible latency is $L$ stage delays, then $M = N/L$. Thus, this distributed organization also reduces the amount of global communication; this is particularly important for large queues, where the loading on control and data buses approaches the level existing in a typical memory array.

Our FIFO queue design makes use of distributed control structures and local communication. There are only a few types of modules in this design, with modules of each type replicated as necessary to construct complete FIFO queues. The distributed control structure allows the exploitation of concurrency. Concurrent read/write supports a higher throughput rate. The FIFO queue is also completely data driven, hence no potential read/write conflict exists and there is no need for any arbiter.

The distributed control organization of the FIFO lends itself naturally to a design using asynchronous, self-timed hardware circuits. Towards this end, a specification technique for asynchronous control structures based on a graph model called Signal Transition Graphs (STGs) have been proposed, and methods for direct and efficient synthesis of self-timed hardware circuits from such specifications have been developed. The STG model is especially appropriate for control modules which exhibit a high degree of asynchronous concurrency. A preliminary discussion of the STG model and its expressiveness and implementability are given in [5]. The use of STGs in the design and implementation of a self-timed $2 \times 2$ packet router is reported in [3].

The paper is organized as follows. Section 2 describes the functional behavior and alternate organizations of the FIFO queue. Section 3 introduces the STG model and discusses its use in the specification of self-timed circuits. In Section 4, STG specifications of the building block control modules for realizing the various FIFO queue organizations and their implementation are presented. Finally, Section 5 presents the results and some further discussions on the STG model.

## 2. Organization of the FIFO queue

This section discusses the one and two-dimensional organizations for the FIFO queue (Figs. 1a and b). The R-module is a control circuit designed to support pipelined operation of the register stages. A R-module has an input link from a previous stage, an output link to the next stage, and an output link to a register module in the same stage to control data loading into this register. A *link* is a pair of *ready/acknowledge* wires, depicted as an arc with the arrow pointing in the direction of the *ready* signal. When input data is available for loading into a register, a *ready* signal is sent to its R-module controller on the $I_r$ wire (Fig. 1a). The actual loading is performed when the R-module controller sends a *ready* signal to the register on the $L$ wire. Once data have been loaded into a register, an *acknowledge* signal is sent to its R-module on the $D$ wire. The R-module then returns an *acknowledge* signal on the $I_a$ wire of the input link and forwards a ready signal on the $O_r$ wire of its output link concurrently. The next data item will be loaded into its register only after the R-module has received an *acknowledge* signal on its $O_a$ wire and another *ready* signal on its $I_r$ wire. Thus, the operation of the R-module is pipelined. Data from one stage will be forwarded to the next unless the latter is full. The throughput rate of this queue is determined by the delays of the R-module and registers, whereas its latency is proportional to the number of stages in the queue.

A two-dimensional, or *ring* organization is shown in Fig. 1b. This queue consists of $M$ linear queues, each of $L$ stages, and two *token rings* for controlling input/output operation. The capacity of the queue is $M \times L$ and the latency is proportional to $L$. I-modules are connected together to form a *token ring* to control the writing of data into the queue. The ring is initialized such that only one I-module contains the token, marking the next available empty register stage. Since the *Write-request* signal, carried on wire $W_r$, is connected to all I-modules, the token should not be passed on to the next module in the ring if the *Write-request* signal is still active. This is an important timing restriction. The *Write-acknowledge* on wire $W_a$ is the output of an OR gate (shown as a heavy bar with a + sign) whose inputs are acknowledge wires from all I-modules. Similarly, reading from the FIFO is controlled by an *Output token ring*, formed by connecting O-modules together. Data written into the linear queues ripple to their output side, ready to be gated onto the output bus. The Output ring is initialized such that only one O-module contains the token. This module then controls the timing and signaling for gating of data to the output bus. The *Read-request* signal on wire $R_r$ is the output of an OR gate whose inputs are request wires from all O-modules. Another timing restriction exists for the Output token ring: since the *Read-acknowledge* signal, carried on wire $R_a$ is broadcast to all O-modules, the token should not be passed on to the next module while *Read-acknowledge* is still active.

0190-3918/86/0000/0397 $01.00 © 1986 IEEE

The Ring buffer which we fabricated is one with minimal latency ($L = 1$), with each of the linear queues containing exactly one stage. Registers in each stage have inputs connected to the input data bus, and outputs connected to the output data bus.

## 3. Signal Transition Graphs

In this section we introduce the STG model and its application to the specification of pipeline controllers. A more complete discussion of STG can be found in [6]. In short, STGs are a form of Petri nets restricted by a set of axioms, and their components (such as transitions and places) assigned attributes related to physical circuits. The result is a graph model which is much more amenable to analysis due to the reduced complexity, and still retains sufficient expressiveness for specifying most common behaviors of control circuits including concurrency, choices and conflicts.

A hardware circuit consists of an interconnection of *logic elements*, each having an output terminal and a number of input terminals. Every input terminal is connected either to an input terminal of the entire circuit, or to an output terminal of another logic element in the circuit. The set of all terminals of a circuit is called the set of *signals, M.* In order to describe the dynamics of a circuit, a set of *signal transitions* $T = M \times \{+, -\}$ is used to specify the rising and falling transitions of each signal in $M$. For each $i \in M$, its associated transitions are denoted by $i+$ and $i-$. It is often convenient to use the notations $t$ and $\bar{t}$ to denote pairs of transitions, such that if $t = i+$ then $\bar{t} = i-$ and vice versa.

For the purpose of this presentation, a STG is a directed graph represented as a triple $\langle T, \mathcal{R}, T_{E0} \rangle$ where $T$ is the set of signal transitions (defined over a signal set $M$), $T_{E0}$ the set of transitions which are enabled in the initial state of the circuit, and $\mathcal{R} \subseteq T \times T$ an *irreflexive, intransitive* relation over the set of transitions, called the *causal* relation. Graphically, $t_1 \mathcal{R} t_2$ is shown as an arc between two transitions: $t_1 \rightarrow t_2$. Let $\mathcal{R}^+ \subseteq T \times T$ denote the transitive closure of $\mathcal{R}$, $t_1 \mathcal{R}^+ t_2$ means that there exists a directed path from $t_1$ to $t_2$; this is shown graphically as $t_1 \rightarrow_p t_2$. The semantics of STG can be expressed in terms of *transition sequences* and their compositions; this has been carried out in [6] using trace theory [14]. Informally, $t_1 \mathcal{R} t_2$ means that the occurrence of transition $t_1$ causes that of transition $t_2$; this implies that if the circuit is in some state $s$ in which transition $t_1$ is enabled and eventually occurs, then the occurrence of $t_1$ brings the circuit to another state $s'$ say, in which $t_2$ is enabled and hence will eventually occur. This last statement hints that given an initial state of the circuit (in which transitions in $T_{E0}$ are enabled) and a STG expressing the causal relation between its signal transitions, one can generate a state transition graph from the STG. A circuit realization can then be obtained from the state graph. Furthermore, a constraint $t_1 \mathcal{R} t_2$ can be implemented as a logic element with $t_1$ as one of the inputs and $t_2$ as output. This important observation allows the decomposition of a STG specification into smaller subgraphs, each of which contains only transitions which are causally related. Thus, STG allows a very efficient and direct implementation based on this decomposition principle. This is a unique feature of STG compared to other approaches.

A multi-arc connects several tail transitions to one head transition, or one tail transition to several head transitions. We call these arc configurations *And Forks* and *Joins* (there are also *Or* constructs for specifying choices or conflicts in the complete STG model), and their diagrammatic notations are shown in Figure 2a. An And-fork is used to describe a situation in which the occurrence of a tail transition causes the occurrences of all of the head transitions. An And-join describes a situation in which all tail transitions in the relation have to occur to cause the occurrence of the head transition. These And constructs are used to describe concurrent operations in circuits.

**Liveness and Persistency.** A STG has a deadlock-free and hazard-free circuit realization only if it satisfies the properties of liveness and persistency. Since STGs are merely behavior specifications from which state graphs can be derived for implementation, these properties must ultimately be based on the latter type of graphs. However, there is a one-to-one correspondence between them, so that liveness and persistency will appear as *syntactic* constraints on STGs. We discuss briefly these properties and their STG syntactic ramifications.

The continual operation without deadlocking of control modules is a property called *liveness*. A STG is live iff its underlying state transition graph is strongly connected. The *necessary* condition for a STG to be live consists of (i) the STG is a strongly connected graph, and (ii) there is a *simple* cycle containing both $t$ and $\bar{t}$ for every $t \in T$. Since live STG are strongly connected, concurrency and ordering have to be characterized differently: two transitions can occur concurrently iff there is no simple cycle containing both of them in the STG; equivalently, the occurrence of two transitions are ordered iff there exists a simple cycle containing both of them.

Due to the similarity to a special class of Petri nets called *marked graphs* [7], it may appear that the form of STGs discussed here is always persistent. However this is not the case, as the underlying state graph may exhibit nonpersistency whenever two transitions are enabled in the same state and the occurrence of one removes the enabling condition of the other. A *persistency constraint* is an ordering constraint between two transitions used to eliminate nonpersistency, as illustrated in Fig. 2b. For a *live* STG, the condition $t\mathcal{R}^+\bar{t}$ always holds for every transition $t$. If $t\mathcal{R}u$ exists as shown and $u\mathcal{R}^+\bar{t}$ (depicted as an heavy arc in Fig. 2b) were not present, then transitions $\bar{t}$ and $u$ can occur concurrently. Suppose the course of action $t\mathcal{R}u$ is implemented by a hardware element with $t$ as one of its inputs and $u$ as its output. Concurrency between $\bar{t}$ and $u$ implies that while the hardware element is reacting to $t$ to cause $u$, $\bar{t}$ may be occurring simultaneously at the input of that hardware element. This is commonly known as a race condition in hardware circuits and can lead to malfunction. The approach to deal with this problem is to impose a persistency constraint on STG specifications, namely $u\mathcal{R}^+\bar{t}$, to eliminate this nonpersistent behavior. Hence, a STG specification is persistent if every transition $u$ caused by a transition $t$ precede $\bar{t}$, i.e. $\forall u \in T$, if $t\mathcal{R}u$ then $u\mathcal{R}^+\bar{t}$.

**Specification of Pipelined Circuits.** We can now develop a STG specification for pipelined control operations such that liveness and persistency are satisfied. Consider two cycles of transitions as shown in Fig. 2c. The left cycle (with $a$ and $\bar{a}$) represents the control sequence of the input portion of a pipelined circuit; the right one (with $b$ and $\bar{b}$) represents the control sequence of its output portion. The necessary condition for liveness is satisfied by each cycle if every transition in a cycle is paired with another in the same cycle. We want the two cycles to operate in parallel as much as possible, with the left one initiates control actions on the right one through arc $a\mathcal{R}b$. In order for the STG to be persistent, three additional arcs (in heavy line) are required. Because of the existence of arc $a\mathcal{R}b$, arc $b\mathcal{R}\bar{a}$ is required as a persistency constraint to prevent concurrent firing of $b$ and $\bar{a}$. The introduction of $b\mathcal{R}\bar{a}$ requires adding $\bar{a}\mathcal{R}\bar{b}$ to prevent concurrent firing of $\bar{a}$ and $\bar{b}$, and this in turns requires arc $\bar{b}\mathcal{R}a$. These four arcs allows the synchronization of two cycles of transitions in pipelined fashion such that the resulting STG is persistent. These constraints can be viewed from a different perspective by "unfolding" the cycles (Fig. 2d) into partial orders, in much the same fashion as occurrence nets [1]. In this type of nets, a node such as $a$ represents an *instance* of a transition $a$ in a cycle of operation. It can be seen that the persistency constraints appear as $a\mathcal{R}b\mathcal{R}\bar{a}\mathcal{R}\bar{b}\ldots$ and thus forces these transitions to occur in sequence. Otherwise, transitions belonging to other branches of the cycles can occur concurrently.

## 4. STG specification of control modules

In this section we apply the STG model to specify the I-module used in the FIFO organizations in Section 2. We will illustrate how to specify a STG such that it meets the liveness and persistency conditions set forth in Section 3. Once a STG satisfying these conditions is obtained, it can be translated directly into a circuit module using the synthesis steps discussed in [6]. We will give the hardware implementation obtained through this synthesis procedure for each STG specification, and discuss the procedure itself informally. The reader is referred to a complete version of the paper [4] for the discussion of O- and R-modules.

For all the control modules we will specify, event occurrences are signalled over control links, using the *reset signaling* handshake protocol [12]. Usually, an occurrence of an event is signalled by a positive transition on the ready wire of the control link; its acknowledgment is signalled by a positive transition on the acknowledge wire of the control link. The signals on these links are then reset through negative transitions before the occurrence of the next event can be signalled.

While liveness and persistency are considered to be fundamental properties of STG, there are other properties more related to the implementation of control circuits according to a certain design methodology. Two such constraints pertinent to the ensuing discussions called **R1** and **R2** are described.

**R1:** This constraint concerns the behavior in the initial state of a control circuit operating with the reset signaling protocol. Starting from the idle initial state, every control module used in the FIFO organizations alternates between an active phase consisting entirely of positive signal transitions, and a reset phase consisting entirely of negative signal transitions. In a circuit implementation, the signal state at each terminal is identified with a signal transition at that terminal: if the state of a signal $u$ is 1 (0), it implies that $u+$ ($u-$) has occurred. If the initial state of a circuit is all 0's then negative transition of the form $u-$ must have occurred in each of these signals in the immediate past. Thus, any positive transition in a STG, say $u+$ which is preceded only by negative transitions of the form $t-$ will always be activated in this initial state. When this is not desired, an artificial constraint from some other positive transition $r+$ to $u+$ must be added. Hence, it is required that *for every STG, the subgraph induced by the set of positive transitions is connected.*

**R2:** The second constraint results from the communication discipline imposed on a control circuit. Control circuits operating with the reset signaling protocol uses pairs of *ready/acknowledge* wires to communicate with the external world. A transition on the *acknowledge* wire can only occur in response to a transition on the *ready* wire and vice versa. For a pair of wires $\{I_r, I_a\}$ where $I_r$ is an input *ready* and $I_a$ an output *acknowledge*, this communication interface to the external world is specified in a STG by the pair of constraints $\{I_a- \mathcal{R} I_r+, I_a+ \mathcal{R} I_r-\}$. Similarly for a pair of wires $\{O_r, O_a\}$ where $O_r$ is an output *ready* and $O_a$ an input *acknowledge*, its corresponding set of constraints is $\{O_r+ \mathcal{R} O_a+, O_r- \mathcal{R} O_a-\}$. Thus, in a STG, *every transition of an input signal has exactly one transition which directly precedes it, and this transition must be that of an output signal.* Transitions of an input signals are underlined to distinguish it from transitions of "non-input" ones.

**STG Specification of the I-module**. An I-module controls the loading of input data into a linear queue (Fig. 1b). When its turn comes, a token is passed to the I-module from its immediate predecessor in the token ring, through a control link $P = \{P_r, P_a\}$. Upon receiving the token, the I-module responds to the next input request on its $W = \{W_r, W_a\}$ control link by sending a load request to the linear queue it controls through its $I = \{I_r, I_a\}$ control link. After loading a new data item into the linear queue, the I-module forwards the token it holds to the next I-module in the token ring through its $N = \{N_r, N_a\}$ link.

The STG description of I-module in Fig. 3 contains two main cycles, the left one coordinates the reception of the ring token with the reception of the next data item presented to the FIFO queue. The right one manages the forwarding of ring token to a successor I-module. The *-arc ($W_r- \mathcal{R} N_r+$) implements the timing constraint discussed earlier, such that $N_r$ does not go high (to pass a token to the next module) until after the input *Write-request* $W_r$ has gone low. These two cycles and the *-arc together provide the specification for proper event sequencing in the I-module. Other arcs are to be added to satisfied persistency and other constraints discussed above. First, arc $D_1$ ensures that all positive transitions form a connected subgraph according to constraint **R1**. Since $D_1$ is a constraint from transition $I_a+$ to $N_r+$, the pairs of transitions $\{I_a+, I_a-\}$ and $\{N_r+, N_r-\}$ could be used for implementing the persistency constraints. This set of arcs would include: $I_a+ \mathcal{R} N_r+$, $N_r+ \mathcal{R} I_a-$, $I_a- \mathcal{R} N_r-$, and $N_r- \mathcal{R} I_a+$. However, since $I_a+$ and $I_a-$ are transitions of an input signal, each can have no more than one incident arc according to constraint **R2**. To enforce these constraints, we change $N_r+ \mathcal{R} I_a-$ to $N_r+ \mathcal{R} I_r-$, and $N_r- \mathcal{R} I_a+$ to $N_r- \mathcal{R} I_r+$. These final constraints are shown as $D_1 - D_4$ in Fig. 3.

Liveness and persistency are satisfied by this STG. The synthesis procedure produces a state graph, from which the realization in Fig. 3 is obtained. The logic equation for $I_r$ is $I_r = W_r P_r \overline{N}_r + I_r(W_r + P_r + \overline{N}_r)$ and its implementation is a C-element with inputs $W_r, P_r$ and $\overline{N}_r$. The logic equation for $N_r$ is $N_r = \overline{N}_a \overline{W}_r I_a + N_r(\overline{N}_a + I_a)$ and its implementation is as shown in Fig. 3. The reader can readily verify that the circuit operates according to its STG specification.

## 5. Result and conclusion

A FIFO with 8 stages and 9-bit wide data path was designed, using a 4 micron nMOS technology. The chip size (including pads) is $3.15 \times 2.25 mm^2$. Six chips were received from MOSIS, they were tested and five were fully operational at a throughput rate of approximately 4 MBytes/sec. An nMOS circuit diagram for the chip is shown in Fig. 4, the lower portion is the control circuitry, with R-modules on top, I-ring in the middle and O-ring at the bottom. The control circuits take a relatively large amount of area in this chip. However, in a 2-dimensional organization with $L \geq 2$, the overhead due to I-modules and O-modules can be reduced significantly.

In this paper, Signal Transition Graphs have been used as a specification tool for asynchronous control modules. A STG specification can be viewed as an interpreted Petri net in which each transition is identified with a signal transition in a hardware circuit. In the synthesis approach proposed, a state transition graph is generated from a STG and then used to derive logic equations and hardware structures for the signals. A STG specification can thus also be viewed as a concise yet more abstract notation for specifying a class of state transition graphs.

In our specification and design examples, it has been shown how introducing additional constraints in a STG allows us to use level-sensitive hardware circuits instead of transition-sensitive hardware circuits in its implementation. These constraints are justified only informally. A more formal theory based on trace theory and state transition graphs are developed in [6].

The module descriptions used in this paper require only constructs for specifying sequencing and concurrency. There are other behaviors which exhibit conflict and data-dependent signal flow that would require additional STG constructs for their specification. These latter constructs are called OR-constructs, and the reader is referred to [5] for an introduction to their formulation and applications.

In [9] Martin described a design approach using constructs for non-deterministic programming to specify hardware modules whose behaviors exhibit only sequencing and arbitration requirements. This approach uses a subset of Dijkstra's guarded command language to specify each process; concurrent cooperating processes are described using notations similar to Hoare's CSP [10]. Heuristic procedures are used to "compile" a hardware implementation from a module specification into an interconnection of standard hardware templates such as And, Or, C-elements, etc.. During the compilation, the technique of reordering events in a sequence is made use to improve implementation efficiency. The complete STG model allows the specification of concurrency, sequencing and conflict in module behavior, and our implementation approach is aimed at automating the derivation of hardware structures from STG specifications. Recently there are works on the classification and synthesis of delay-insensitive circuits based on trace theory [13,14]. The relation of STG to trace theory is analogous to that of Petri nets model to its underlying sequence semantics. Thus, we believe that STG can serve as a high-level, more abstract specification than that of an approach directly based on trace theory.

There are also related works on verification of asynchronous hardware structures based on temporal logic [8]. Such techniques can be used fruitfully for correctness validation of self-timed circuits. The design of suitable translation techniques from high-level language to STG -in the same vein as those done by Martin and Rem [11]- is another area for further exploration.

**Acknowledgments.** The authors would like to thank Prof. Jack Dennis for his support and guidance, Prof. Lance Glasser and Dr. Bill Ackerman for helpful suggestions. This research was supported in part by grants 26-7603-3775 from Lockheed Corp., NSF-7915255 and a Hughes Fellowship.

## REFERENCES

[1] Best, E. "Concurrent Behaviour: Sequences, Processes and Axioms." *LNCS 197*, Springer-Verlag 1984.

[2] Chin, C.-Y. and K. Hwang, "Packet Switching Networks for Multiprocessors and Data Flow Computers." *IEEE Transactions on Computers*, C-33, No. 11, November 1984.

[3] Chu, T-A., C. Leung and T. Wanuga. "A design methodology for concurrent VLSI systems." *Proceedings of the ICCD-85*, IEEE, Oct. 1985.

[4] Chu, T-A. "Design of a self-timed ring buffer using Signal Transition Graphs." Computation Structure Group Memo. 247, MIT Lab for Computer Science, 1985.

[5] Chu, T-A. "On the models for designing VLSI asynchronous digital systems." To be published in *INTEGRATION, the VLSI Journal.* North-Holland, 1986.

[6] Chu, T-A. *Signal Transition Graphs and the Modeling of Self-timed Circuits.* Ph.D. thesis in preparation, Department of EECS, MIT, 1986.

[7] Commoner, et. al. "Marked directed graphs." *Journal of Computer and System Sciences.* No. 5, 1971.

[8] Dill, L.D. and E.M. Clarke, "Automatic verification of asynchronous circuits using temporal logic." *Proceedings of the 1985 Chapel Hill Conference on VLSI.* Computer Science Press, May 1985.

[9] Martin, A.J. "The design of a self-timed circuit for distributed mutual exclusion." *Proceedings of the 1985 Chapel Hill Conference on VLSI.* Computer Science Press, May 1985.

[10] Martin, A.J. "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits." 5210:TR:86, Dept. of Computer Science, CalTech 1986.

[11] Rem, M. "Concurrent Computations and VLSI Circuits." in *Formal Description of Programming Concepts - II.* D. Bjørner (ed.), North-Holland Pub. Co. *IFIP* 1983.

[12] Seitz, C.L. "System Timing." Chapter 7 of *Introduction to VLSI Systems*, Mead and Conway (Eds.), Addison Wesley 1981.

[13] Udding, J.T. *Classification and Composition of Delay-Insensitive Circuits.* Ph.D. thesis, Dept. of Mathematics and Computing Science, Eindhoven Univ. of Technology, 1984.

[14] van de Snepscheut, J.L.A. *Trace Theory and VLSI Design. LNCS 200*, Springer-Verlag 1985.

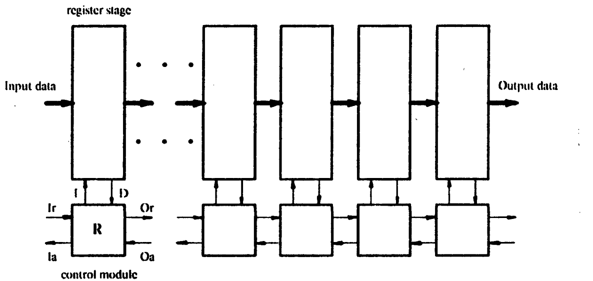


Fig. 1(a): A one-dimensional organization

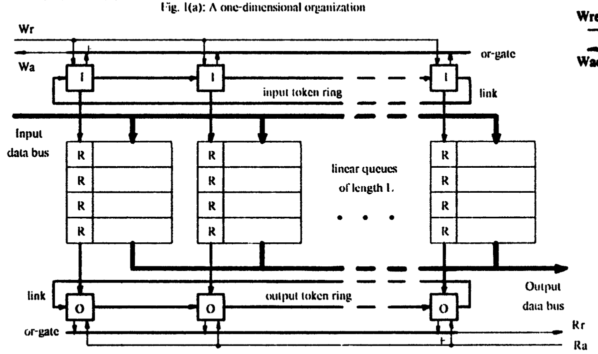


Fig. 1(b): A ring organization

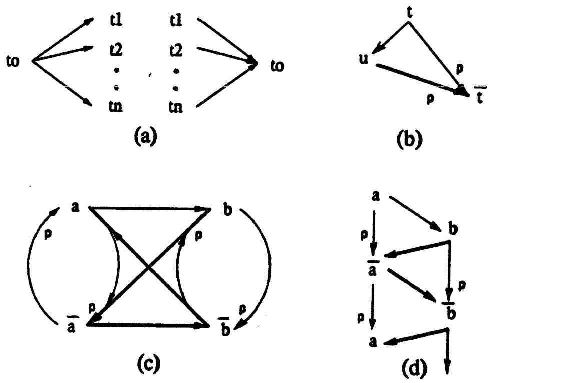


Figure 2: (a) Conjunctive Forks and Joins. (b) A persistency constraint. (c) A specification of pipeline operation. (d) The partial order resulted from "unfolding" of the STG in (c).

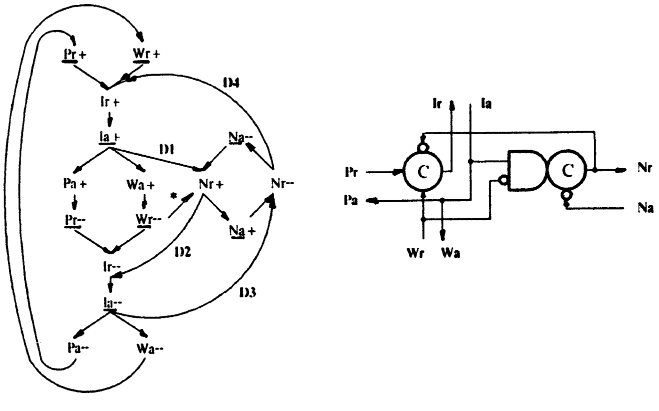


Figure 3: STG description and realization of the I-module

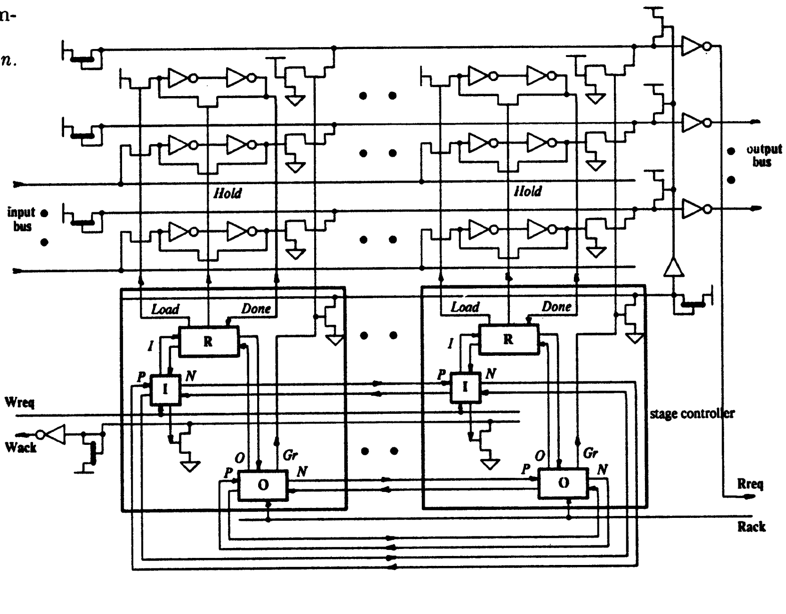


Figure 4: Circuit diagram of the test Ring Buffer.

# Optical Matrix-Vector Implementation of Crossbar Interconnection Networks*

A. A. Sawchuk, B. K. Jenkins, C. S. Raghavendra, and A. Varma

Signal and Image Processing Institute
and
Computer Research Institute

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089

## ABSTRACT

The use of optical techniques has been widely recognized as a solution for overcoming the fundamental problems in data communications. It is potentially feasible to build large optical crossbar networks that do not suffer from many of the limitations of their electronic counterparts. In this paper we discuss several optical matrix-vector implementations of crossbar networks and present a comparative study of these designs. These designs employ acousto-, electro-, or magneto-optic spatial light modulators for achieving the generalized crossbar functions. Some new optical systems suitable for implementation of crossbar networks are also described. They can also be used to construct multistage networks of larger size.

## 1. INTRODUCTION

Any multiprocessor system that uses several processors must be designed to allow efficient communication among processors and memory units, as this data communication contributes significantly to the overall performance of the system. An ideal interconnection network for this purpose is a high speed, high bandwidth crossbar network. A crossbar network will allow any processor to communicate with other processors or memories with simple control and small delay. Even though crossbars are ideal networks for multiprocessors, they are generally used in only relatively small systems, as the cost of large crossbars is prohibitively expensive with the VLSI technology. There have been several crossbar designs reported for up to several hundred inputs and outputs [BURG 83, DENN 82, BROO 84].

For large crossbar implementations other types of technology may offer a viable alternative. The use of optical techniques has been widely recognized as a solution to overcoming the fundamental problems in data communications [SAWC 84]. The parallel nature of optics and free-space propagation, together with their relative freedom from interference make them ideal for parallel communications. There exists many different optical devices which provide switching capability. It was shown in [SAWC 85] that any implementation of matrix-vector multiplier in optics results in an optical crossbar network.

Suppose that $N$ serial data input lines (each one-bit wide) are to be connected to a set of $N$ serial data output lines (each one-bit wide). We can think of the data bits on each line flowing synchronously, although, for some of the techniques we describe, this synchronism is not necessary. We can represent the data on the input lines at some instant of time by the column vector $\vec{b}$ of length $N$. The data on the output line of the interconnection network (which we refer to as a switch) is represented by the column vector $\vec{c}$ of length $N$, and both these vectors have only binary elements. The state of the switch is described by the $N \times N$ matrix $A$ whose elements are also 0's and 1's. Matrix $A$ is a generalized permutation matrix; an entry of 1 in row $i$ and column $j$ of $A$ means that input $j$ is connected to output line $i$.

Once the physical connection described by the matrix $A$ is made, the data transfer could be synchronous or asynchronous [SAWC 85]. This idea can be generalized for the interconnection of $N$ parallel input lines (each $M$ bits wide) to $N$ parallel output lines (each $M$ bits wide). Thus, systems capable of performing matrix-vector or matrix-matrix multiplications with binary entries may be suitable for use as a crossbar interconnection network. In this paper, several different optical implementation of crossbar networks are presented and the tradeoffs in their performance are discussed.

## 2. OPTICAL MATRIX-VECTOR SYSTEMS

There are many techniques available for matrix-vector or matrix-matrix multiplication using optics. In some of these systems, the physical switch may be analog and passive (a switchable reflective or transmissive element); thus bit synchronism is not required and the data bandwidth is limited only by the optical sources and detectors used (currently available sources and detectors can operate at > 1 Gb/s). In other systems, the matrix-vector multiplication is implemented by active electro-optical or acousto-optical components, implying the detection and regeneration of optical signals. In such systems the switch itself may limit the data bandwidth. In the remainder of this section we will discuss some of the optical systems we have studied for crossbar implementations.

### 2.1 $N^2$-Parallel Matrix-Vector Inner Product Processor

In this system, $N$ input lines drive an array of $N$ light emitting diodes (LEDs) or laser diodes with a binary signal, so that a binary 1 is represented by light of a fixed intensity, and a binary 0 is represented by a lower (or zero) intensity. An optical system to the right of the input vector spreads the light from each input source into a vertical column that illuminates the crossbar mask [SAWC 85]. Following the crossbar mask, the next set of optics collects the light transmitted by each row of the mask, and sums the mask output onto a vertical array of $N$ photodetectors corresponding to the $N$ output lines. Thus the system performs a parallel matrix-vector multiplication. Since it is a passive interconnection, once the mask is set, the data can flow through synchro-

* This research is supported by the DARPA/ARO Contract No. DAAG29-84-K-0066.

nously or asynchronously, can be analog or digital, and has a bandwidth limited only by the sources and detectors.

### 2.2 Systolic and Engagement Architectures

Systolic and engagement architectures have the advantage of providing for rapid reconfiguration of the network, i.e., they can be reconfigured at the bit rate of the data (neglecting any overhead in calculating the needed states of the crossbar matrix elements). In these systems, though, the input data must be configured appropriately for the matrix multiplication to proceed; this will require electronics at the input that can buffer the signals and send them into the switch in the appropriate sequence and at appropriate times. In some cases, only minimal buffering is required, e.g. when the data is originally time-division multiplexed bit by bit. More details on systolic architectures can be found in [SAWC 85].

An engagement architecture for matrix multiplication is shown in Fig. 2.1. The matrix elements of $A$ enter into a 2-D source array or a multichannel AO cell, skewed and formatted as shown. The data, or elements of $B$, enter a second, crossed multichannel AO cell from the top and propagate downwards. A 2-D stationary integrating detector array then accumulates the results and outputs them in parallel. Systolic architectures are similar to engagement architectures but use shift-and-add detectors instead of time-integrating detectors and data enters in a different way.

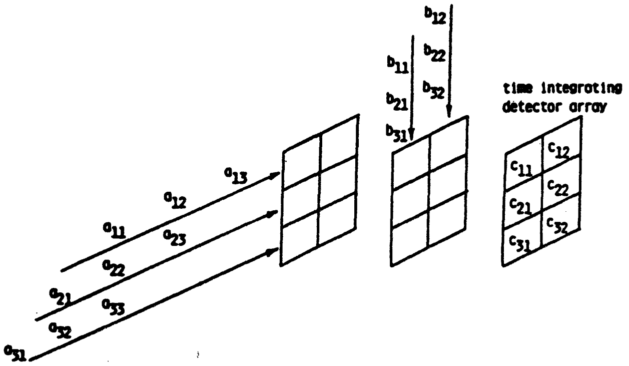


Fig. 2.1. Engagement architecture for matrix multiplication.

Compared to the inner product architecture above, systolic architectures yield crossbars that are more light efficient, have shorter reconfiguration times, but have lower bandwidth. For an $N$-parallel system, with sufficiently fast input and output devices, the AO cell will generally limit the bandwidth of each individual line to $\nu_0/2N$, where $\nu_0 \approx 1$ Gb/s. A number of these 1-D elements in the $N$-parallel multiplier can be placed in parallel however, permitting each line to be $M$ bits wide, thereby increasing the effective bandwidth by a factor of $M$ ($M \approx 100$ with current devices). If $M = N$, then the system is an $N^2$-parallel systolic matrix-vector multiplier. Because a new matrix is essentially read in for each new bit or word, the system can be reconfigured rapidly.

An $N$-parallel engagement matrix multiplier requires $N(3N-2)$ clock cycles to complete one matrix-matrix operation or one arbitrary switching operation on $N$ $N$-bit wide lines, approximately the same as the ($N$-parallel) systolic case. An $N^2$-parallel engagement arrangement, such as the RUBIC (Rapid Unbiased Bipolar Incoherent Calculator) cube [BOCK 84], takes $(3N-2)$ clock cycles. The input data formatting requirements are similar to the systolic case, but interleaved 0's are not required. The $N$-parallel case uses time multiplexing again. We can think of the width of each functional input line as being equal to the word length so that an $M$-bit wide line can transfer one word at an instant of time. In the $N^2$-parallel engagement system, there are $N$ physical input lines, and the switch operates on word slices: it is word serial and bit parallel except for a unit time shift from one physical line to the next. In other words, the first line has the first bit of each word, sequentially in time, etc. In the systolic case above, the physical lines are actually serial over diagonals of matrix $B$ rather than strictly bit-serial or word-serial. The detectors here are 1-D or 2-D time integrating detector arrays. As with systolic systems, reconfiguration is rapid but updating of the state of the switches is needed even when the state does not change from one multiply to the next.

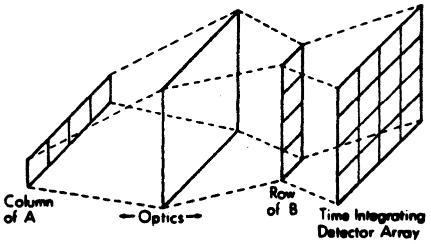


Fig. 2.2. $N^2$-parallel outer product processor.

### 2.3 $N^2$-Parallel Outer Product Processor

An $N^2$-parallel outer product processor such as that shown in Fig. 2.2 performs $C = AB$ by taking outer products of column $i$ of $A$ and row $i$ of $B$, and summing the results over $i$. It can perform a matrix-matrix operation in $N$ clock cycles. The requirement of a 2-D detector array will limit the bandwidth to much lower values; removing the data electronically from the detector array slows the system down. The data input is a row of B at a time, or bit parallel, word-serial. The detector is a 2-D array of stationary, time-integrating detectors. Since the $A$ matrix is read in for each matrix multiplication, reconfiguration is rapid, although the state of the switches needs to be updated for each matrix multiply.

### 2.4 $N^2$-Parallel Inner Product AO Deflector

While the above systems may be useful in many applications, they all share a common fault. Since the matrix $A$ is mostly 0's, most of the scalar multiplications performed by the above systems are multiplies by 0. This translates into either wasted time (thereby lowering the bandwidth) or wasted light (lowered light efficiency). One of the new architectures we have devised eliminates these problems which is the inner product AO deflector and is very similar to the system described in Section 2.1 except for the deflector. It is also a passive system and utilizes a simple 1-D detector array.

The input data to the crossbar enters through a 1-D array of sources. Each source is then transferred to the corresponding cell (or channel) of a multichannel AO device, used as a deflector. There is a separate channel for each data-input line, or each column of $A$. Rather than reading in the elements of $A$ into the AO device ($N$ elements into each channel), one single number is input into each channel, and the signal is frequency-modulated according to the amount of deflection desired for the corresponding input line. Broadcasting can be achieved, to a limited extent, by superimposing multiple frequencies onto the same channel of the AO cell, or by time multiplexing the different frequencies in the cell. Each channel of the AO cell in this system has just one signal on it; it is not divided into multiple (moving) resolution elements. Each channel then deflects the signal the appropriate amount in the y dimension, and the light is then focused down in the x dimension, yielding a one dimensional output array.

### 2.5 $N^2$-Parallel Inner Product/Engagement Processor

Another new type of matrix-matrix processor architecture that could be used as an optical crossbar is shown in Fig. 2.3. The system is a combination of an $N^2$-parallel inner product processor and an $N^2$-parallel engagement architecture. The $b$ 's to be multiplied in the system enter a parallel array of AO cells as shown, although the data for each row enters simultaneously instead of being staggered as in the $N^2$-parallel engagement processor. The input data also arrives in a time-staggered form and controls the illumination of LED's or laser diodes as shown. A set of optics similar to that in the $N^2$-parallel inner product processor spreads the light across the multichannel Bragg cell, and the final result accumulates on a time-integrating detector array. The results emerge in a time-offset fashion from a row of the detector array as shown. This system requires ($3N$−2) clock cycles to complete a matrix-matrix multiplication and requires a digital synchronous data format. Broadcast operation is possible with this system, and the overall light efficiency can be as high as $1/N$.

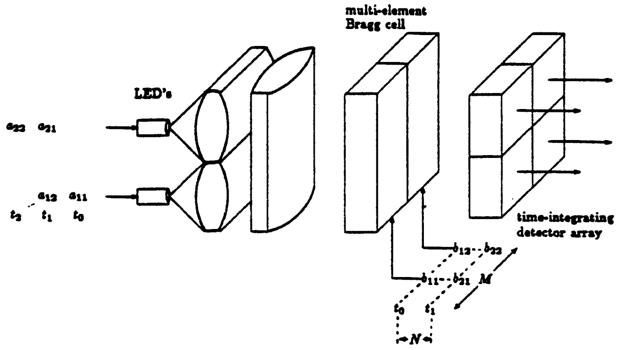


Fig. 2.3. $N^2$-parallel inner product/engagement processor.

## 3. PERFORMANCE COMPARISON

The optical systems introduced in the previous section differ widely among themselves in many characteristics which are of importance when used for interconnection. For example, some of the systems provide a bandwidth that is essentially independent of the size of the system, while for others, it is an inverse function of the system-size. In this section, we compare the different systems based on some parameters of interest in crossbar implementation.

*1. Number of Lines N:* Optical systems can implement moderately large crossbars (100×100 to 500×500) with relative ease.

*2. Bandwidth:* For passive optical networks, the bandwidth of each line is limited by sources and detectors, which can easily operate at 1 Gb/s rates or higher. Sources (laser diodes) and detectors are available at 20 GHz and 40 ps, respectively. 1-D arrays are presently limited to lower values; current detector arrays with parallel outputs operate at approximately 50 Mb/s. Higher bandwidths could be achieved by first coupling to fibers and routing the fibers to detectors; still higher bandwidths might be possible by avoiding conversion to electronics altogether. In contrast, electronic systems typically operate at 10 Mb/s for each line.

*3. Reconfiguration Time:* Reconfiguration of optical networks is limited to approximately 1 $\mu$s for moderate or large networks with current or near-future technology; in fact, most 2-D switch arrays have a frame time of 1 ms or greater. Acousto-optic devices have faster response times, but the electronic input (to each cell) is serial and limited to a few GHz; and the signal cannot be changed as it propagates down the cell. Most of the acousto-optic architectures permit fast reconfiguration at the expense of bandwidth.

*4. Broadcast Capability:* In electronic implementations, the provision of broadcasting involves higher control complexity, a larger number of pins, and/or slower operation. In some optical systems, the addition of broadcast capability involves only very minor increases in complexity.

*5. Data Format:* The switching process as well as the transfer of data can be performed either synchronously or asynchronously. Synchronous operation requires strobing or capture of data at precise instants of time in the system.

*6. Type of detector:* There are three major types of optical detectors, namely, real-time, shift/add, and time-integrating detectors. The inner-product processors need a vector array of detectors with separate channels and a real-time output. The data-bandwidths of these systems is generally limited by the response characteristics of the detector array. In some of these systems (e.g. those with real time detectors), the output may be directly coupled to a light guide or fiber; in this case the detectors, if any, are physically located some distance from the optical switching array. The engagement and outer-product systems with full $N^2$ parallelism require $N \times N$ time-integrating detectors which can be operated synchronously with the data. 2-D $N \times N$ detector arrays have either $N$ output lines or 1 output line. Thus the necessity of multiplexing at least $N$ detector signals onto each electronic output line creates a bottleneck and limits the bandwidth of the overall system. The systolic architectures require a shift/add detector array to perform the summation. $N \times N$ shift/add arrays necessarily have at most $N$ output lines; this and the use of electronics to perform the shift and add limits the detector speed, in turn lowering the overall bandwidth of the system.

Table 3.1 summarizes the above considerations for the six basic optical matrix-vector crossbar architectures given in Section 2.

## 4. CONCLUSIONS AND FUTURE WORK

In this paper, we described several possible optical systems for implementation of crossbar networks and studied the tradeoffs involved. Advantages of these systems include large amount of inherent parallelism, high data-bandwidth, small size and power requirements, and freedom from mutual interference of signals. We have found that moderately large crossbars (64×64 to 512×512) may be feasible using current or near-future optical technology. It is feasible to build optical crossbars that have a higher bandwidth and more data lines than electronic systems, although their reconfiguration speeds are much slower. The main difficulties that need to be surmounted are the slow reconfiguration, means for efficient conversion of electronic signals to optical signals and vice-

| Method of implementation | Number of lines | Bandwidth (BW) | Reconfiguration time | Broadcast | Data format | Type of detector |
|---|---|---|---|---|---|---|
| $N^2$-parallel inner product | $> 256$ | 1 Gb/s (passive) | $(N)1\ \mu$S | yes | async | real time $N \times 1$ element |
| $N^2$-parallel systolic | $> 100$ | 10 Mb/s* | $N$/BW | yes | sync | $N \times N$ shift/add |
| $N^2$-parallel engagement | $> 100$ | 10 Mb/s* - 100 Mb/s | $N$/BW | yes | sync | $N \times N$ time-integrating |
| $N^2$-parallel outer product | $>256$ | 10 Mb/s* - 1 Gb/s | max ($N$/BW, $N \cdot 10$ nS) | yes | sync | $N \times N$ time-integrating |
| $N^2$-parallel inner product AO deflector | $> 100$ | 1 Gb/s (passive) | 1 $\mu$S | limited | async | real time $N \times 1$ element |
| $N^2$-parallel inner product/ engagement | $>100$ | 10 Mb/s* - 100 Mb/s | $N$/BW | yes | sync | $N \times N$ time-integrating |

Table 3.1. Comparison of optical crossbar capabilities.

* Limited by output (detector array) electronics. Higher numbers (when given) apply when fibers guide light to discrete detectors or other optical components.

versa, and techniques for control.

An intriguing and powerful possibility is to use these crossbars as building blocks to make larger networks. In order to make use of this important possibility, means of cascading these optical crossbars need to be devised. This involves studying the input and output formats of the data in the different optical systems for compatibility. The light efficiency of the crossbar must also be factored in, as it determines how often detection and regeneration of the optical signals is needed. Methods of physically coupling the output of one system to the input of the next must be devised, and additionally the signals may need to be sent to multiple crossbars in different locations. Of course, one way of doing this would be to use electrical lines to connect up the optical crossbars; but it is preferable to keep as much of the overall system optical as possible, in order to maximize the advantages that the optics provides, such as high bandwidth and large number of lines. All-optical amplifiers have been demonstrated [KOBA 84], [DAGE 86a]. While there are practical problems to be solved to build 1-D arrays of optical amplifiers, it can, in principle, be done [DAGE 86b]. Optical cascading of crossbars can be done with fibers or holograms. It has been shown [JENK 84] that holograms can be quite powerful for interconnecting optical gates. Even the capability of connecting a small number (4 or 5) of sizable crossbars together could yield a system that is substantially more powerful than a single crossbar.

## REFERENCES

[ATHA 83] R. A. Athale, "Optical Matrix Algebraic Processors: A Survey," *Proc. 10th Int. Optical Computing Conf.*, Cambridge, MA, April 1983, pp. 24-31.

[BOCK 84] R. P. Bocker, "Optical Digital RUBIC (Rapid Unbiased Bipolar Incoherent Calculator) Cube Processor," *Opt. Eng.* vol. 23, January/February 1984, pp. 26-33.

[BROO 84] G. Broomell, J. R. Heath, "An Integrated-Circuit Crossbar Switching System," *Proceedings of the 4th International Conference on Distributed Computing Systems,* May 1984, pp. 278-287.

[BURG 83] T. Burggraff, A. Love, R. Malm, A. Rudy, "The IBM Los Gatos Logic Simulation Machine Hardware," *Proceedings of the International Conference on Computer Design,* 1983, pp. 584-587.

[DAGE 86a] M. Dagenais, W. F. Sharfin, "Extremely Low Optical Switching Energy Nanosecond Response Bistable Devices," *Proc. Soc. Photo-Opt. Instr. Eng.* Vol. 625, 1986, To appear.

[DAGE 86b] M. Dagenais, *Private Communication.*

[DENN 82] M. M. Denneau, "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference,* Las Vegas, 1982, pp. 55-59.

[FENG 81] T. Y. Feng, "A Survey of Interconnection Networks," *Computer,* Vol. 14, No. 12, December 1981, pp. 12-27.

[GOOD 78] J. W. Goodman, A. R. Dias, L. M. Woody, "Fully Parallel High-Speed Incoherent Optical Method for Performing Discrete Fourier Transforms," *Optics Letters,* Vol. 2, 1978, pp. 1-3.

[JENK 84] B. K. Jenkins, *et al.*, "Architectural Implications of a Digital Optical Processor," *Appl. Opt.*, Vol. 23, 1984, p. 3465.

[JENK 85] B. K. Jenkins, "Recent Developments in Digital Optical Computing," *Annual Meeting, Optical Society of America,* Washington, D.C., October 1985.

[KOBA 84] S. Kobayashi, T. Kimura, "Semiconductor Optical Amplifiers," *IEEE Spectrum,* May 1984, pp. 26-33.

[SAWC 84] A. A. Sawchuk, T. C. Strand, "Digital Optical Computing", *Proc. IEEE,* Vol. 72, 1984, pp 758-779.

[SAWC 85] A. Sawchuk, B. Jenkins, C. S. Raghavendra, A. Varma, "Optical Interconnection Networks", Proc. 1985 International Conference on Parallel Processing, August 1985, pp 388-392.

[SMIT 84] P. W. Smith, " Applications of All-Optical Switching and Logic," *Phil. Trans. R. Soc. Lond. A,* Vol. 313, 1984, pp. 349-355.

# Distributed Recovery in Applicative Systems

Frank C.H. Lin [*]
ESL, Inc.
Sunnyvale, California 95051

Robert M. Keller[*]
Quintus Computer Systems, Inc.
Mountain View, California 94041

**Abstract:** Applicative systems are promising candidates for achieving high performance computing through aggregation of processors. This paper studies the fault recovery problems in a class of applicative systems. The concept of *functional checkpointing* is proposed as the nucleus of a distributed recovery mechanism. This entails incrementally building a resilient structure as the evaluation of an applicative program proceeds. A simple rollback algorithm is suggested to regenerate the corrupted structure by redoing the most effective functional checkpoints. Another algorithm, which attempts to recover intermediate results, is also presented. The parent of a faulty task reproduces a functional twin of the failed task. The regenerated task inherits all offspring of the faulty task so that partial results can be salvaged.

**Keywords**: fault tolerance, error recovery, distributed systems, applicative systems, data flow architecture, functional language.

## 1. Introduction

An important feature of a multiprocessor system, including applicative multiprocessing systems, is the ability to sustain partial system failures. By an*applicative system* in this paper, we mean a partitioned-memory system such as Rediflow [8, 9, 18] which coherently executes *applicative*, or *functional*, programs.

The evaluation of an applicative program generates an implicit call tree. The result of the root task is the answer of the program. Every task in the call tree represents a partial result which is used by its parent task to compute other partial results. Because the semantics of applicative of languages has no notion of destructive modification, a parent task is capable of regenerating all of its child tasks based upon the argument and function information.

Many fault-tolerance techniques for general multiprocessor systems have been proposed [1]. Some of these schemes can be adapted to applicative systems. However, applicative systems possess some interesting characteristics, e.g., *determinacy*, that merit distinct fault recovery considerations [6, 7, 14].

In this paper, fault tolerance issues in a class of applicative systems are studied. We assume that any task can be executed by any processor and that tasks are dynamically assigned to execution processors at run time. A single processor failure is also assumed. A processor is assumed to be either faulty or fault-free. A faulty processor must voluntarily declare itself faulty, or otherwise be identified as faulty by other processors.

It is further assumed that if a processor fails, it will no longer transmit any valid messages. This assumption can be enforced by commanding a faulty node to keep silent and not to respond to any inquiry. Alternatively, a faulty node may answer an inquiry with an invalid message. Several techniques are available for a processor to determine node malfunctioning. Parity checking on the system bus or resident memory, illegal instruction trap, protection violation, or a subsystem breakdown may trigger the CPU reporting a processor failure. Duplication of processors within a node, called "passive node diagnosis" [12], is also a common technique for building self-checking nodes.

It is assumed that a processor makes its best effort to communicate with a destination node. If the destination cannot be reached due to a network problem, the unreachable node is considered faulty. Problems with the interconnection network may be detected via coding or timeout mechanisms.

Our approach exploits the determinacy property of applicative programs. A distributed checkpointing scheme, *functional checkpointing*, is proposed in the next section. As the evaluation of an applicative program proceeds, a distributed resilient evaluation structure is incrementally established across the network of processors. Any single processor breakdown is salvaged by the implicit redundant path of the robust structure. A simple rollback recovery algorithm, which basically discards all partial results, is discussed in section 3. In section 4, another recovery algorithm, *splice recovery*, is proposed to salvage as many intermediate results as possible. Tasks which are equivalent to those trapped inside the faulty processor are generated to replace the failed tasks. Partial results produced by the failed tasks are inherited by the recovery tasks.

## 2. Functional Checkpoints

Checkpointing is familiar in the fault-tolerant computing literature [1]. In a uniprocessor system, checkpointing is normally performed by storing machine state on nonvolatile devices periodically. Such a periodical checkpointing technique has been extended to multiprocessor systems [3, 5, 7, 15]. The basic idea is to virtually stop all computational operations while periodic global checkpointing takes place.

[*] Work reported herein was supported by a grant from the IBM Corporation, while both authors were at the University of Utah.

Periodic global checkpointing may not serve the best interests of fault tolerant applicative systems. For example, nonvolatile storage for storing system states may not be necessary, if recovery of a faulty processor is accomplished outside the node. Checkpoint information may be stored on one or more peer processors. Furthermore, periodic global synchronization among a large number of processors is potentially inefficient [2].

We propose a distributed checkpointing strategy for applicative systems. The approach attempts to exploit the determinacy property of applicative programs.

By a *functional* checkpoint, we mean a recovery point for a function application in an applicative system. A partial state of the system is stored so that recovery of the function is possible. The partial system state used in a functional checkpoint is related to a *single* function only. Normally, a functional checkpoint does not have enough information to recover an entire node, not to mention recovering a system. The sole purpose of the partial state is just to back up a function application.

The idea of functional checkpointing is to disseminate the responsibilities of recovering a faulty node to processors which have immediate relationship with the faulty node. Complete recovery is done by collective efforts from various associated processors to retrieve the corrupted tasks.

### 2.1 Determinacy

*Determinacy*, or *referential transparency*, is the characteristic of applicative programs which makes them attractive for distributed execution. A program is called determinate if an identical answer always results from any function invocation for given arguments. In other words, a functional program is free from side effects.

Determinacy suggests that an appropriate time for a functional checkpoint is when a parent task spawns a child function. A task packet is formed for the new function and then waits for execution. The packet contains *all* necessary information, either directly or indirectly accessible, to activate the child task. Furthermore, determinacy insures that different activations of the same task packet will always yield the same result. Thus, even if a task is aborted during computation, a new invocation will not be contaminated by its predecessors.

### 2.2 Checkpoint Properties

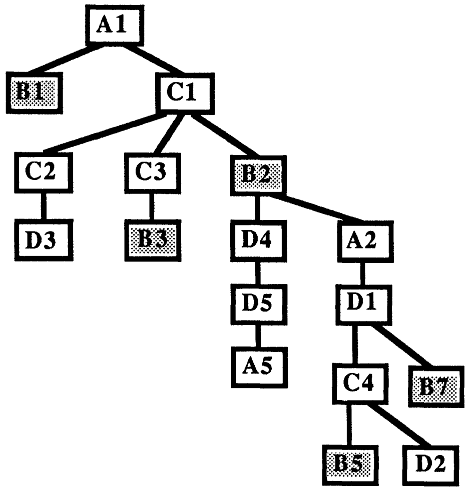
Periodic checkpointing is a synchronous operation whereas functional checkpointing is **asynchronous**. Each processor holds the privilege and responsibility of checkpointing its offspring tasks. A processor may opt to arrange the checkpoints in a partial order such that more efficient recovery can be implemented (section 3). Checkpoint coordination between processors is not necessary.

Functional checkpointing can be implemented **implicitly**. As a child task is spawned to a new node, the parent task may retain a copy of the task packet. This retained copy is all that the parent needs to regenerate the child task, should the node evaluating the child task fail. Therefore, functional checkpointing can be fully embedded in the evaluation process.

## 3. Rollback Recovery

Using functional checkpointing as a framework, a simple rollback recovery mechanism can be devised. An applicative call tree is mapped onto a set of processors. Each processor may have an arbitrary number of tasks. When a processor fails, the call tree may be broken into pieces. However, the piece that contains the root task is always capable of regenerating all severed pieces.

Suppose that an applicative program has been spawned into the call tree as shown in Figure 1. For ease of discussion, tasks Ai (i = 1, 2) are mapped onto processor A, tasks Bi are executed in processor B, etc. Suppose that processor B fails. Then tasks Bi are destroyed. The call tree is thus fragmented into three pieces: {A1,C1,C2,C3,D3}, {A2,D1,D2,C4}, and {D4,D5,A5}.



| A | B | C | D |
|---|---|---|---|
| --- | A2 | ----- | A5 |
| B1 | --- | B2,B3,B5 | B7 |
| C1 | --- | ----- | C4 |
| D1 | D4 | D2,D3 | --- |

Figure 1: A call tree mapped onto processors A,B,C, and D, and corresponding distribution of checkpoints

Assuming the check point of an application is kept on the processor of its parent. Processor A contains the functional checkpoint for B1, processor C contains checkpoints for B2, B3 and B5, and processor D contains checkpoints for B7. To recover from the failure of B, the system needs to command processor A to respawn B1, and command processor C to regenerate B2 and B3. Task B2 will in turn generate new tasks which are equivalent to D4 and A2. Since an applicative program has no side effects, it does not require any *undo* operation, and hence there is no *domino effect* [13].

Note that task C4 holds the checkpointing data for B5. Processor C may regenerate B5 when B fails. However, the recovery of B5 is not fruitful because antecedent task A2 cannot report its result to B2. Reactivation of B5 only increases the system overhead. Therefore, an efficient way to salvage a group of genealogical dependents is to redo only the most ancient ancestor and ignore the rest.

### 3.1 Level Stamps

Genealogical dependencies among tasks can be monitored by a simple level numbering scheme. Assume that the root task carries a null level number, a task at level one will bear a unique one digit identification. Tasks in subsequent levels are stamped by appending one more digit to the number of their parents. The term "digit" is used here generically and is not limited to a specific radix representation.

Since each task is associated with a unique level stamp, it is obvious that ancestor-descendant relationships can be observed by comparing stamps. Note that a level stamp is not a time stamp. Its uniqueness is guaranteed by the program structure. Stamping of tasks can be fully asynchronous.

### 3.2 Recovery Scheme

Each processor maintains a table of linked lists. The Nth entry of the table contains all topmost checkpoints from the host processor to processor N. Referring to FIgure 1, for example, when processor C spawns task B2 to processor B, C compares the level stamp of B2 with all checkpoints in entry B. If B2 is a descendant of an existing functional checkpoint, C does nothing. Otherwise, processor C makes a checkpoint for B2 in entry B.

When processor C identifies the failure of processor B, C simply reissues all the checkpointed tasks found in entry B of the table. By doing so, processor C fulfills its responsibility of recovering B. Other processors take similar actions to recover their descendant tasks being trapped in B. The complete recovery of a faulty processor is a collective effort from processors which have checkpointed applications on the failed processor.

During task evaluations, a processor is required to abort a task if new arguments of the task cannot be obtained due to failures of other processors. A task is also aborted if the result of the task cannot be forwarded to the parent task. The aborted tasks and their descendants may be recollected during garbage collection operations.

### 3.3 Dynamic Allocation and Recovery

The possibility of discarding intermediate results without extensive undo operations is a property of applicative programs. However, the ability to recover by simply reissuing checkpointed tasks depends on the availability of a dynamic allocation strategy, such as the *gradient model* approach [10].

Recovering tasks in a static allocation environment requires manipulations of some linkage information. For example, tasks being allocated to a failed processor have to be reassigned to other processors. Descendants of the reassigned task have to modify their return addresses accordingly. Furthermore, the balanced state derived from the static allocation method may not be maintained easily after a processor fails.

Dynamic allocation does not distinguish between tasks generated for recovery and original tasks. All tasks are treated equally during load-balancing activities. The parent-child linking information is dynamically produced. Hence, there is no need to update these linkages when the task is reassigned.

### 3.4 Orphan Tasks

Rollback recovery inevitably leaves a few orphan tasks after some recovery has taken place, e.g., task D4 in Figure 1 becomes an orphan when processor B fails. The problem is that a task might not know whether it is an orphan without expenditure of a considerable amount of system resources.

Returns from orphan tasks are theoretically harmless since they are forwarded to a faulty processor and no side-effect can be induced. However, the partial results produced by orphan tasks are in fact correct answers of their associated functions. Failure of a node does not contaminate these incomplete answers; it just breaks the linkage among them. These partial results are usable if the regenerated parent task knows where to retrieve them, or if the orphan tasks know the new address to which to forward their answers. The desire to salvage partial results motivates the design of the following recovery scheme.

## 4. Splice Recovery

Applicative systems facilitate evaluation of a functional program by dynamically unfolding the underlying structure of the algorithm and disseminating parallel tasks to many processing nodes. At any instant, task distribution in a system represents a snapshot of the program structure. Generation of a task creates a new substructure and establishes a linkage between the parent and children. Return packets from a child task normally eliminate the children that are no longer needed.

The simple rollback scheme cuts off the branch or branches originating from a faulty node and regrows new branches. The method basically abandons all intermediate results computed by the orphan tasks. This section suggests a different approach, *splice recovery* , which attempts to retrieve all possible intermediate results.

### 4.1 Resilient Evaluation Structure

The splice approach is to continuously establish a *resilient evaluation* structure during program computations. A resilient structure is one containing redundant information which allows a system to rebuild the original structure after a failure has been identified. By rebuilding the structure, the system may salvage many partial results.

We have seen that when a processor fails, an applicative call tree may break into several pieces. The idea of splice recovery is to provide necessary bridging information such that broken pieces can be put together again. When a parent discovers the failure of a child task, the parent task generates a twin task of the faulty child. This twin task **inherits** all offspring of the faulty task with the help of the grandparent pointer.

A grandparent pointer of a task is a pointer from the task to its ancestor in the grandparent *processor*. For example, the grandparent pointer of task B3 in Figure 1 points to task A1, and task D4 has a grandparent pointer to C1 (Figure 2).

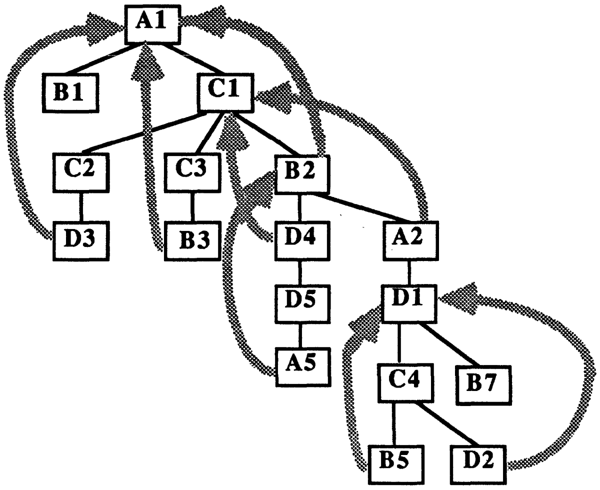


Figure 2: Grandparent pointers

Assuming as before that processor B fails, processor C may start recouping the loss of B2 as soon as C realizes that node B is dead. A twin task of B2, say B2', is created by the parent C1 to inherit tasks D4 and A2 (Figure 3). A full emulation of task B2 would require task B2' to possess physical binding information between B2 and D4, and between B2 and A2. Unfortunately, this information must be embedded not only inside the faulty node, but also within every descendant processor. Changing the return addresses of every descendant task at various sites could be very tedious.

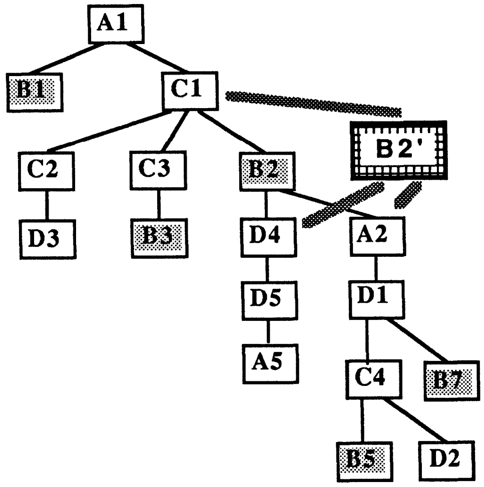


Figure 3: Task B2 is inherited by task B2'

Instead of fully emulating a faulty task, we opt to make B2' inherit descendant tasks of B2. Suppose that when D4 tries to return the evaluated answer to parent B2, it detects that node B is dead. The algorithm commands D4 to forward the result to grandparent C1. Processor C receives these unexpected partial answers from grandchildren and asserts that the parent of these grandchildren is faulty. Then, processor C forms the recovery task B2' by duplicating the task packet of B2.

If processor C has already reproduced B2' when the return from D4 arrives, task A simply forwards what it has received to step-child B2'. The role of a grandparent node in this recovery scheme is two-fold: it reproduces the dead task and it transports the orphan results to their step-parent when these returns become available. Having the grandparent relay partial results eliminates the problem of updating return addresses in every orphan task.

A recovered task, like any other, starts executing its function code as soon as it is committed to a physical processor. When it encounters a function call, it forms a task packet and spawns the child. However, offspring of a recovered task may or may not have been demanded by the preceding faulty task. Let P represent a faulty task, and C a child task of P. Let P' be the recovery task for P. C' is generated by P' and is the equivalent of C, as suggested in Figure 4.

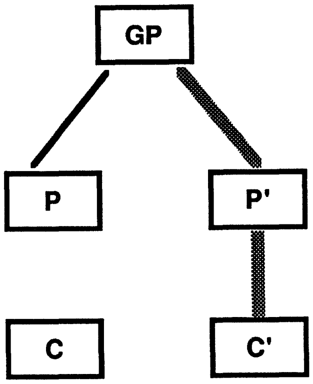


Figure 4: Tasks in splice recovery model

The relationship between child task C and its clone C' has the following possibilities: (Figure 5)

(1) C has never been invoked;
(2) C will never complete;
(3) C completes before P dies;
(4) C completes after P dies, but before P' is invoked;
(5) C completes after P' is invoked, but before C' is invoked;
(6) C completes after C' is invoked;
(7) C completes after C' has completed;
(8) C completes after P' has completed.

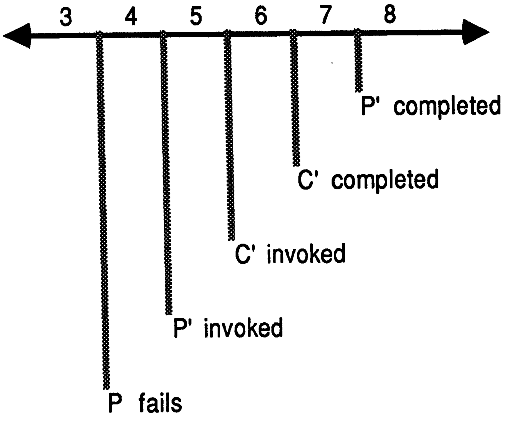


Figure 5: All possible orderings with respect to completion of C

**Case 1 or 2:** C has never been invoked or C will never complete. In either case, no result of C is produced. Task C is practically nonexistent and will be garbage collected. Only C' may produce an answer.

**Case 3:** C completes before P dies. Task C may have already finished the computation and returned the answer back to parent P before P breaks down. The result of task C is stored inside the parent P. When P fails, the system loses all partial results which have been saved in P. The recovery task P' must recalculate C by activating task C'.

**Case 4 and 5:** an old result comes before the new invocation. Task C finishes computation after the parent P dies. C sends its result to the grandparent task which transfers the result to the step-parent P'.

The difference between cases 4 and 5 is that in case 4, the grandparent has to reproduce P' first. When child task C' is executed by task P', P' will not spawn C' because the answer is already there. There is still only one result C' in the system.

**Case 6:** C completes after C' is invoked. Suppose that P' has already spawned C' when the result from C arrives at P'. Theoretically, the result from C and the would-be answer from C' are identical. Therefore, parent P' takes the answer from C and proceeds with the execution. The addition of C' may produce a duplicate answer to P'. Since they are identical, the second copy is simply ignored.

**Case 7:** C completes after C' has completed. This is the reciprocal situation of case 6. Note that due to the asynchrony nature of task evaluations, late invocation of an identical task may yield a result faster than the earlier invocation.

**Case 8:** old result arrives after everything is completed. The processor which contained P' may no longer recognize the arrived answer. The result is discarded.

**4.2 Protocol for Splice Recovery** The main idea of the splice recovery method is to build a resilient structure along with a program evaluation. The redundant information must be in place long before a recovery is initiated. This section describes the usage of these redundancies from the view of a single processor.

```
LOOP
  CASE received packet OF
  forward result:
      Interpret the level stamp.
      CASE level stamp OF
        child: Place data at the location indicated by the
               level stamp. If a task can be continued,
               resume the task.
         grandchild: Create a step-parent for the grandchild
                     if there isn't one already.
                     Transfer the result to its step-parent.
        others: Ignore the packet
      ENDCASE

  fetch data: If the location has been evaluated, forward the
              data. Otherwise, DEMAND_IT.

  task packet: Execute the task. DO each instruction
       If an unevaluated function encountered,
       DEMAND_IT.
       If cannot proceed, suspend the task.
      UNTIL completion. Send the result to the parent.
      If the parent is dead,
            notify the grandparent and
            send the result to the grandparent.

  error-detection: Find the topmost offspring of all
        branches, respawn all of these apply tasks.
        Establish transport mechanism for relaying
        partial results.
  ENDCASE
ENDLOOP.
```

The routine *DEMAND_IT* is the fundamental evaluating process of an applicative program. We elaborate the algorithm in the following form.

```
DEMAND_IT:
        Create a task packet.
        Level-stamp the task packet
        Attach parent and grandparent identifications
           to the task.
        Queue the task packet to load balancing manager.
        Functional checkpoint the packet.
End DEMAND_IT.
```

As a rule of thumb, if a processor receives a packet and cannot find a proper rule to handle it, the processor simply ignores the received message. Note that the overhead of splice recovery protocol is small. By using the level stamps as tags for a program structure, the apparent overhead for establishing a resilient structure is a physical identification of grandparent node which may be just an integer.

### 4.3 Correctness of the Recovery Scheme

The successful evaluation of an applicative program is signaled by the completion of the root task. A necessary condition for completing the root evaluation is to satisfactorily compute all immediate descendants of the root. This observation, when applied recursively, implies that all tasks must be evaluated correctly.

In order to guarantee a task can be evaluated, the task has to be generated in the first place. This means that every task is flawlessly reproducible even if some processor may fail during the evaluation. *Reproducibility* of tasks is the main criterion for a resilient applicative system.

#### 4.3.1 Reproducibility

If the failed processor contains the root of a task tree, the regeneration of the root does not come naturally with recovery schemes. The user must restart the program, or a preevaluation functional checkpoint needs to be implemented.

One simple method to generate a preevaluation checkpoint is to create a super-root which acts as the parent processor of all user programs. When a user program is initiated, the super-root checkpoints the program so that a duplicate copy of the program can be found in the system should the root fail.

With this modification, every task in an applicative program has a parent. A parent task is capable of generating and regenerating any immediate child task as long as the parent is informed by some error detecting mechanism. This satisfies the reproducibility requirement of a correct recovery algorithm.

### 4.3.2 Residue Effects

Without loss of generality, evaluation of an applicative tree can be typified by scrutinizing the spawning process of a three-task sequence. Figure 6 shows the state transition diagram of spawning and reduction of task G. Task G spawns task P which subsequently spawns C. Note that states b and d are transient. The existence of transient states is a result of the dynamic load balancing method.

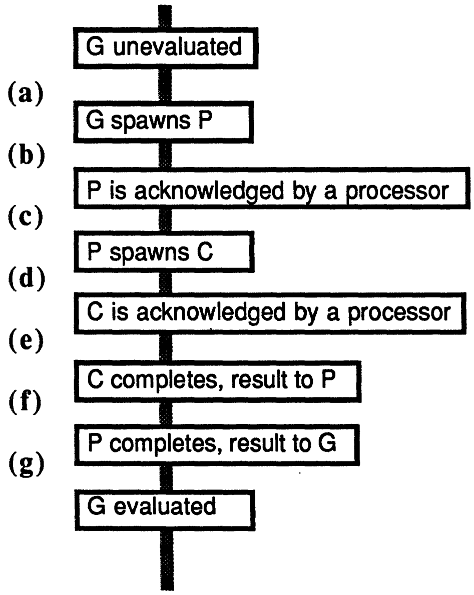


Figure 6: State transition diagram for evaluating G

The pointers being produced and reduced among tasks of each state are depicted in Figure 7. The pointer from P to its grandparent and the pointers to P from its grandchildren are omitted for clarity. Assume that task P fails during the evaluation of G, residue effects may affect any one of the related tasks at any stage of the state transition. A residue-free fault tolerant measure must assure that tasks G and C are not affected by the failure of P from state a through state g.

The failure of P obviously has no effect in state a. In state b, failure of the processor which absorbs P means that parent task G will not receive a positive acknowledge from task P. As a result, processor G times out and reissues a new task P. The system acts as if the first invocation of P did not take place.

In state c, task G receives an acknowledge from P and establishes a parent-to-child pointer to P. The new pointer may provide additional fault detection capability, but the impact of the failure remains similar to that of state b.

Residual effects, resulting from the failure of P, may happen at state d and successive states. Parent task G is left in the same situation as if it were in state b or c. However, there is a child task C lingering around the system. Task C may be stranded due to incomplete information from parent P. In this case, C commits suicide and the recovery from G is free from residual effects, or task C may complete the evaluation and try to return the result. C sends the result to G after failing to communicate with parent P. The case analysis in section 4.1 applies here.

State e has exactly the same recovery condition as state d, since the transient state d becomes state e as soon as task C finds an idle processor. The discussions on state d can be applied here and will not be repeated. State f is similar to state c as far as recovery is concerned.

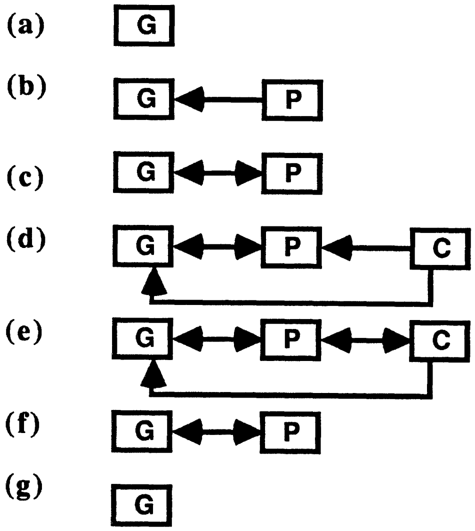


Figure 7: Available pointers among tasks

## 5. Discussion and Related Research

### 5.1 Robust Storage Structures

Using a resilient structure for fault-tolerant computing is not a new idea. Waldbaum [19], and Taylor, et al. [16, 17] proposed a robust storage structure to ensure data integrity in uniprocessor or shared memory machines. Item count, identifier field, and/or additional pointers are commonly added for error detection and recovery purposes. This paper extends the concept of resilient structure to a distributed applicative evaluation graph.

Conceptually, an evaluation structure is quite different from a storage structure. A storage structure is an object manipulated by programs, while an evaluation structure is a program itself. Furthermore, most techniques developed for resilient storage structure seem to be impractical in distributed systems. For example, item count of a linear list is a convenient way for checking broken links in a shared memory machine. To maintain a correct item count and verify it regularly in a network of processors would require significant traversing overhead.

### 5.2 Multiple Faults

Both the rollback and splicing recoveries use functional checkpoints to tolerate hardware failures. Although single node failure is assumed throughout the discussion, it is obvious that rollback recovery is not limited to tolerate only single node failure. The difference between multiple faults and single fault in the rollback algorithm is the placement of the recovery border in the evaluation graph.

Splicing recovery can handle some combinations of multiple faults gracefully. For example, multiple failures on different branches of a structure do not disturb the recovery algorithm at all. Separate recoveries take place at different parts of the program in parallel. However, if both the parent and grandparent processors of a task fail simultaneously, the orphan task would be stranded. It is noted that the resilient structure concept can be further extended to include pointers to the great grandparent and beyond to tolerate multiple failures on one branch of the graph.

### 5.3 Hardware Redundancy

In a hardware redundant fault tolerant system, several redundant machines execute an identical program on replicated data objects. An applicative system can emulate hardware redundancy by simply replicating the task packets. Eventually, a task is executed by several processors at random times. The results are sent back to the originating node asynchronously. The originating node compares these results and selects a majority consensus as the correct answer.

A fundamental difference between applicative replicated task redundancy and pure hardware redundancy is that applicative systems execute redundant tasks asynchronously, while most hardware redundant systems employ synchronous operation. Asynchronous operations are subject to timing delays because a node has to wait for the return from slower processors. But a node does not have to wait for the slowest answer if it has received the identical results from the majority of replicated tasks. Replicating tasks provides a means of emulating hardware redundancy in applicative systems. The user may specify certain critical sections of a program for such a highly reliable operation.

### 5.4 Related Research

Fault tolerant problems in data-driven systems have been studied [6, 7, 11, 14]. Misunas proposed a triple modular redundancy implementation of a dataflow machine [4, 11]. Three complete copies of the program are stored in the memory. Copies of each instruction are carefully distributed so that each copy is executed by a different processor and utilizes different communication paths. Thus, the failure of any single block affects at most one copy of the program.

Hughes [7] described a variation of periodic checkpointing, where a host processor periodically stored the whole system state. Also discussed was a recovery technique, node-by-node correction, which used a control unit of the system as a monitoring device. Erroneous packets were recomputed and re-sent.

Srini [14] suggested a node reassignment algorithm for error recovery purposes. The algorithm depends on a global system memory for collecting and communicating recovery messages. The checkpointed node state is stored in the global memory.

Grit [6] proposed a structural recovery method where each node in the system is limited to spawning child tasks to its immediate neighbors. At system initialization time, a node receives a list of recovery sites for each of its immediate neighbors. When a node fails, a neighbor notifies the recovery site. The recovery node polls all possible parent and child nodes of the failed processor and tries to reconstruct the lost task.

## 6. Summary

This paper discusses the reliability aspect of applicative multiprocessor systems and suggests means for fail-soft treatment. The concept of functional checkpointing is proposed. Unlike conventional checkpoint schemes, functional checkpointing is concise, distributed and asynchronous. Two fault recovery techniques based on the notion of functional checkpointing are proposed. The thrust of these recovery models is to minimize the overhead while the system is in a normal, fault-free operation.

The simple rollback recovery method attempts to reconstruct the faulty section of the program structure by redoing the functions from the most efficient parent task or tasks. In other words, the recovery starts from the most recent functional checkpoints. The scheme is simple and has very little overhead in a normal operation. But, if a fault happens at a later stage of the evaluation, the rollback recovery may be costly.

The splice recovery scheme also uses the most recent functional checkpoints for error recovery as in the rollback method. In addition, the splicing scheme tries to salvage as much intermediate partial results as possible. The salvage is made possible by a backward grandparent linkage along with a program graph stamping mechanism. This approach enables the parent tasks of a faulty processor to regenerate the corrupted substructure of the program and splice the recovery results into the framework preceding the failure.

## 7. References

[1] T. Anderson and P.A. Lee. Fault Tolerance: Principles and Practice. Prentice Hall International, 1981.

[2] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. Computing Surveys 15(1):3-43, March, 1983.

[3] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In Proc. of 13th Int'l Conf. on Fault-Tolerant Computing, pages 48-55. IEEE, June, 1983.

[4] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. In Proc. 2nd Annual Symposium on Computer Architecture. IEEE, 1974.

[5] M.J. Fischer, N.D. Griffeth and N.A. Lynch. Global states of a distributed system. IEEE Trans. on Software Engineering SE-8(3):198-202, May, 1982.

[6] D.H. Grit. Towards fault tolerance in a distributed applicative multiprocessor. In Proc. of the 14th Int'l Conf. on Fault Tolerant Computing, pages 272-277. IEEE, June, 1984.

[7] J.L.A. Hughes. Error detection and correction techniques for dataflow systems. In Proc. of the 13th Int'l Conf. on Fault-Tolerant Computing, pages 318-321. IEEE, June, 1983.

[8] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In AFIPS, pages 613-622. AFIPS, June, 1979.

[9] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. IEEE Computer 17(7):70-82, July, 1984.

[10] F.C.H. Lin and R.M. Keller. Gradient model: A demand-driven load balancing scheme. In Proc. 6th Int'l Conf. on Distributed Computing Systems, IEEE, Cambridge, Massachusetts, May, 1986.

[11] D.P. Misunas. Error detection and recovery in a data-flow computer. In Proc. 1976 Int'l Conf. on Parallel Processing, pages 123-131. IEEE, August, 1976.

[12] K.N. Oikonomou and R.Y. Kain. Abstractions for node level passive fault detection in distributed systems. IEEE Trans. on Computers c-32(6):543-550, June, 1983.

[13] B. Randell. System structure for software fault tolerance. IEEE Trans. on Software Engineering SE-1(2):220-232, June, 1975.

[14] V.P. Srini. Node reassignment in a dataflow system. In Proc. 4th Int'l Conf. on Distributed Computing Systems, pages 15-27. IEEE, San Francisco, CA., May, 1984.

[15] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. In Proc. of the 13th Int'l Conf. on Parallel Processing, pages 32-41. IEEE, August, 1984.

[16] D.J. Taylor, D.E. Morgan and J.P. Black. Redundancy in data structures: Improving software fault tolerance. IEEE Trans. on Software Engineering SE-6(6):585-594, November, 1980.

[17] D.J. Taylor, D.E. Morgan and J.P. Black. Redundancy in data structures: Some theoretical results. IEEE Trans. on Software Engineering SE-6(6):595-602, November, 1980.

[18] S.R. Vegdahl. A survey of proposed architectures for the execution of functional languages. IEEE Trans. on Computers C-33(12):1050-1071, December, 1984.

[19] G. Waldbaum. Audit programs - A proposal for improving system availability. Res. Rep. RC2811, IBM, February, 1970.

# Fault Tolerant Scheme on Partial Differential Equations

Gyo OSAWA, Toshio KAWAI, and Hideo AISO
Faculty of Science and Technology
KEIO University
Yokohama, 223, JAPAN

*ABSTRACT*

Parallel machines in which the Processing Units (PUs) are connected with a nearest neighbor mesh connection structure are ideal for solving partial differential equations. Here, a dynamical fault recovery scheme with such a structure is proposed.

In the case of the detection of a faulty PU within the execution process, connections of PUs are changed; the model is re-discretized by using the calculus of finite differences; the execution is continued from the faulty point.

This scheme was implemented on PAX-32, a parallel computer consisting of 32 microcomputers, and 2-dimensional Poisson equation was solved. As a result of simulation, the effectiveness of this scheme is demonstrated.

## 1. INTRODUCTION

Parallel computers with the nearest neighbor mesh (NNM) connection are ideal for solving partial differential equations (PDEs) by using iterative methods. For instance, PAX[1,2] or NASA's Finite Element Machine[3,4] has been proposed.

Through progress in VLSI technology, highly parallel computers with a great number (e.g., thousands or millions) of microprocessors can be developed. In these systems, the realization of fault tolerant computing is indispensable, and some inter-PU connection networks have been proposed. Most of these are a sort of multistage network (e.g., OMEGA[5], DELTA[6]), or have some redundancies (e.g., Chordal Ring[7], Lens[5]). There are only a limited number of references on the *dynamic fault recovery* within the execution processes.

When solving PDEs with an NNM structured machine, the processes may be divided broadly into two categories as follows:

1. Discretization Process: A PDE is converted with one of discretizing methods.
2. Iteration Process: The converted equations are solved by using some iterative methods.

These processes require a great number of computations; this type of recovery method is considerably important.

In this paper, we discuss the dynamic fault recovery within only the latter process, where it takes much more time than the former essentially. In the case of discretizing 2-dimensional Poisson equation by using the calculus of finite differences, the fault recovery algorithm and simulation results with PAX-32 are also proposed.

## 2. Fault Recovery in NNM Structured Machine

When trying to perform the fault recovery dynamically in an NNM structured machine, such actions as below may be taken:

1. Calculations for a faulty PU are executed by other fault-free PUs (e.g., HOBONET[8]). If these fault-free PUs are the surrounding PUs, a load balance for each PU may be unequal. On the other hand, if one of the redundant PUs takes over the executions for the faulty PU, irregular data-transfers may occur, or the fault recovery algorithms may be more difficult. These schemes are inefficient.
2. A faulty PU is logically disconnected, and processes belonging to this PU cease from executions. This scheme causes worm-eaten results, but has the advantage of easy recovering. In the case of numerical methods, solutions are obtained only on the mesh points. This means that a loss of data on a few mesh points can be negligible, if the accuracy loss is small.

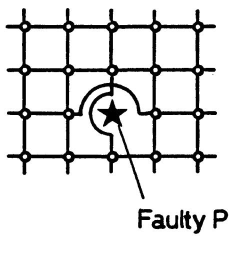


Fig. 2.1 Bypass Recovery

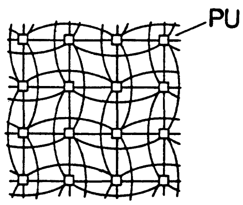


Fig. 2.2 Links for Bypass Recovery

The scheme in question for us is the latter one. In our scheme, damages caused from the faulty PUs can be reduced to the minimum, and executions of fault-free PUs continue. As shown in Fig. 2.1, with the detection of a faulty PU, interconnections of the surrounding PUs are changed and the faulty PU is disconnected. If there have been redundant links in the NNM in advance as shown in Fig. 2.2, this type of fault recovery can be executed easily. We call this a *Bypass Recovery*.

## 3. Discretization on irregular intervals

We first use the calculus of finite differences as a discretizing method, because, the bypass recovery can then be applied efficiently.

Suppose that any PU is assigned to one node of a PDE model respectively. If the bypass recovery mentioned above is performed, the model is changed around the faulty PU (Fig. 3.1). So in the process of the fault recovery, re-discretization of the model should be performed by using the finite difference method on irregular intervals. In the model of Fig. 3.1, the discrete equation on x-direction is

$$u_{i,j} = \frac{2u_{i-1,j} - 3u_{i,j} + u_{i+2,j}}{3h^2} + o(h)$$

where $h$ is the interval between the two neighboring nodes. This way of discretization can also apply to the y-direction's or a model with various physical constants; as well as, to sum up, all differential terms. The order of error changes from $o(h^2)$ to $o(h)$ around the faulty PU by re-discretization. However, the accuracy of results obtained will only be slightly affected quantitatively.

## 4. Method of Assigning PU to Nodes of Model

A PU assigning algorithm is needed, where the number of nodes of a model exceeds the number of PUs. Generally, an algorithm is taken as shown in Fig. 4.1(a). In this method, however, some continual nodes belonging to a faulty PU get lost. On the other hand, in the *modular mapping* as shown in Fig. 4.1(b), the number of lost nodes by one faulty PU is as the same as those in Fig. 4.1(a). In this method, it takes more communicating time since all the nodes on a PU must communicate with all the nodes in the 4 adjacent PUs. The modular mapping, however, has the advantage that lost nodes do not continually exist but scattered. Thus damages on the results of calculations by a faulty PU can be kept lower in this method. In conclusion, this mapping method is adopted for a PU assigning algorithm.

## 5. Implementation of Poisson Equation on PAX-32

The fault recovery algorithms mentioned above are implemented on the PAX-32 computer. PAX-32 is a parallel machine, consisting of 32 microprocessors combined into a nearest neighbor connected torus array, as shown in Fig. 5.1. As for a PU faulty, one PU out of 32 PUs is selected randomly within the iteration process. This machine does not have redundant links mentioned in chapter 2; corresponding data-transfers are supported with software routines. The target model is a 2-dimensional Poisson equation,

$$\Delta u = -1.0$$

A region of the model, boundary conditions, and physical constants are shown in Fig. 5.2.

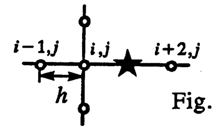


Fig. 3.1 Finite Difference Method on Irregular Intervals

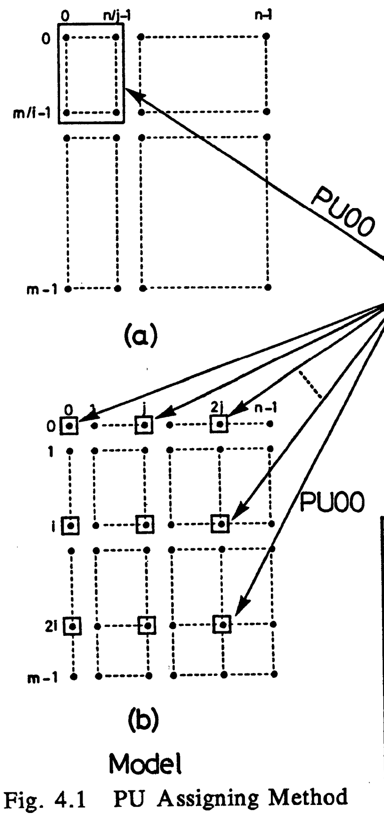


Fig. 4.1 PU Assigning Method

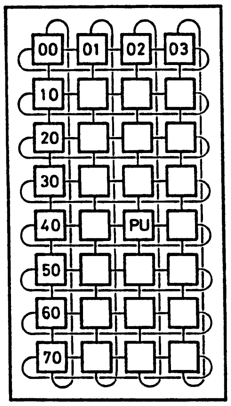


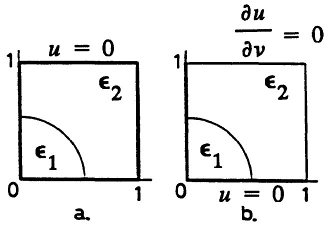
Fig. 5.1 PAX-32 System Configuration

Fig. 5.3 illustrates the calculation time versus the number of nodes per PU. This figure shows that the calculation time is proportional to about the square of the number of nodes. The calculation time depends upon the iteration count and the number of nodes per PU. Undoubtedly, this time is directly proportional to the number of nodes. So, if the iteration count is proportional to the number of nodes, this rate is in agreement with those of theoretical analysis.

Fig. 5.4 illustrates the iteration count versus the total number of nodes. This count $K$ is proportional to the number of nodes. The $K$ can be calculated from the theoretical study, and is expressed as follows:

$$K = - \frac{4 + \log_{10}( 1 - \lambda )}{\log_{10}\lambda} ,$$

1. Region and Boundary Conditions



2. Physical Constants

α. $\epsilon_1 = 1$ , $\epsilon_2 = 1$

β. $\epsilon_1 = 1$ , $\epsilon_2 = 10$

γ. $\epsilon_1 = 1$ , $\epsilon_2 = 100$

3. Number of Nodes within a Region

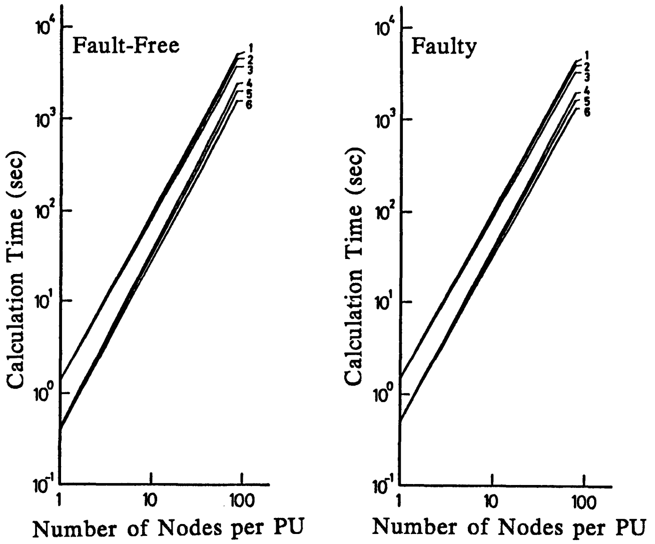
1. 32 ( 8* 4)
2. 128 (16* 8)
3. 800 (40*20)
4. 2304 (48*48)

Fig. 5.2 Simulation Conditions

where λ is the maximum eigenvalue of the coefficient matrix of the model. Some of the theoretical $K$ is illustrated in Fig. 5.5. In this figure, the slight difference between the theoretical $K$ and the actual $K$ is due to the approximation of the theoretical equation. As a result, both the calculation time and the iteration count agree with those of theoretical analysis.

TABLE 5.1 shows such error rates as defined below:

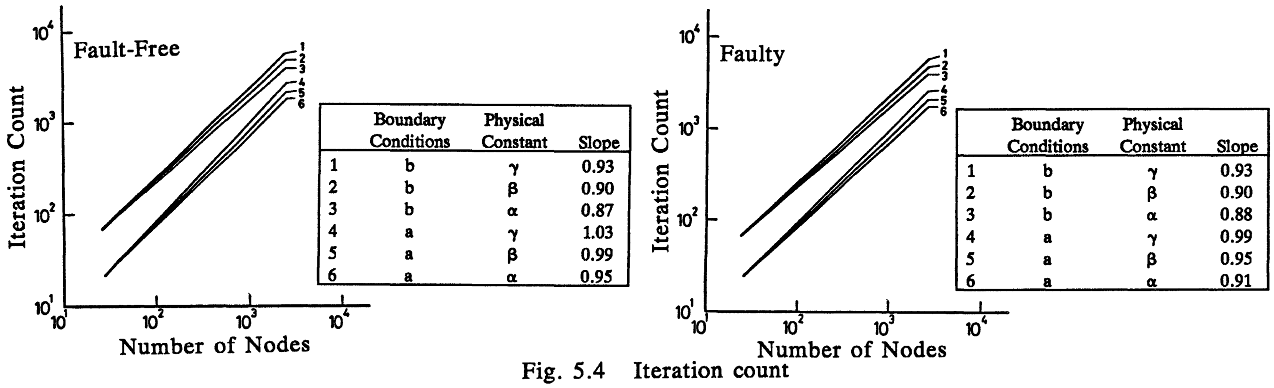
$$\textit{error rate} = \frac{\textit{result at faulty} \; - \; \textit{result at fault-free}}{\textit{result at fault-free}}$$



| | Boundary Conditions | Physical Constant | Slope |
|---|---|---|---|
| 1 | b | γ | 1.84 |
| 2 | b | β | 1.81 |
| 3 | b | α | 1.78 |
| 4 | a | γ | 1.92 |
| 5 | a | β | 1.88 |
| 6 | a | α | 1.85 |

| | Boundary Conditions | Physical Constant | Slope |
|---|---|---|---|
| 1 | b | γ | 1.85 |
| 2 | b | β | 1.82 |
| 3 | b | α | 1.79 |
| 4 | a | γ | 1.90 |
| 5 | a | β | 1.86 |
| 6 | a | α | 1.82 |

Fig. 5.3 Calculation Time on PAX-32



| | Boundary Conditions | Physical Constant | Slope |
|---|---|---|---|
| 1 | b | γ | 0.93 |
| 2 | b | β | 0.90 |
| 3 | b | α | 0.87 |
| 4 | a | γ | 1.03 |
| 5 | a | β | 0.99 |
| 6 | a | α | 0.95 |

| | Boundary Conditions | Physical Constant | Slope |
|---|---|---|---|
| 1 | b | γ | 0.93 |
| 2 | b | β | 0.90 |
| 3 | b | α | 0.88 |
| 4 | a | γ | 0.99 |
| 5 | a | β | 0.95 |
| 6 | a | α | 0.91 |

Fig. 5.4 Iteration count

Note that lost results belonging to a faulty PU are left out of account. This is a result of simulations with 10 kinds of initial numbers of random number. This table shows that effects of a faulty PU are almost negligible.

## 6. Concluding Remarks

In this paper, we proposed a new scheme called *Bypass Recovery* for disconnecting faulty PUs, which is useful with an NNM connection structure. This is a dynamic fault recovery scheme, and efficient recovering can be performed by using the calculus of finite differences on irregular intervals as discretization.

Simulating this scheme on PAX-32, it appears that the proposed scheme realizes high efficiency and reliability.

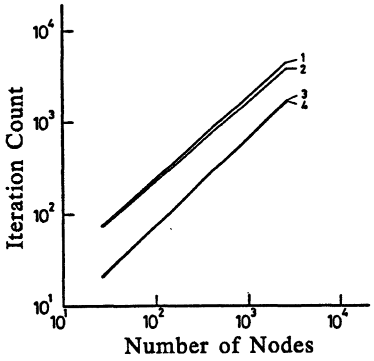
It is desirable that this scheme can apply to the FEM. Basically, re-creation of the coefficient matrix of the model at the faulty point can be performed; the application will be possible with a small modification.

## ACKNOWLEDGEMENT

We are greatly indebted to Mr. Takashi Yokota for the concept of this study. Without him, this study could not have been realized. Also we would like to express our sincere gratitude to Prof. Mario Tokoro and Dr. Hideharu Amano for their advices. In the use of the PAX-32 computer, we thank to Prof. Tsutomu Hoshino of the University of Tsukuba.

## References

1. T. Hoshino, T. Kawai, T. Shirakawa, and et al., "PACS: A Parallel Microprocessor Array for Scientific Calculations," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 195-221, Aug. 1983.
2. T. Hoshino, T. Kamimura, and et al., "PARALLELIZED ADI SCHEME USING GECR ( GAUSS-ELIMINATION-CYCLIC-REDUCTION ) METHOD AND IMPLEMENTATION OF NAVIER-STOKES EQUATION IN THE PAX COMPUTER," *Proc. of the 1985 Intl. Conf. on Parallel Processing*, pp. 426-433, Aug. 1985.
3. H. F. Jordan, "A Special Purpose Architecture for Finite Element Analysis," ICASE Report, No. 78-9, March 1978.
4. L. M. Adams, P. Mehrotra, and et al., "THE FEM-2 DESIGN METHOD," *Proc. of the 1983 Intl. Conf. on Parallel Processing*, pp. 132-134, Aug. 1983.
5. Duncan H. Lawrie, Raphael A. Finkel, and Marvin H. Solomon, "The Lens Interconnection Strategy," *IEEE Trans. Comput.*, vol. C-30, no. 12, pp. 960-965, Dec. 1981.
6. Janak H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Trans. Comput.*, vol. C-30, no. 10, pp. 771-780, Oct. 1981.
7. B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Trans. Comput.*, vol. C-30, no. 4, pp. 291-295, Apr. 1981.
8. G. Osawa, T. Yokota, and et al., "HOBONET: An Inter-PU Connection Network with Fault-Tolerancy," *Proc. of the 1984 Intl. Conf. on Parallel Processing*, pp. 165-168, Aug. 1984.



| | Boundary Conditions | Physical Constant | | Slope |
|---|---|---|---|---|
| 1 | b | α | Actual | 0.87 |
| 2 | b | α | Theoretical | 0.85 |
| 3 | a | α | Theoretical | 0.96 |
| 4 | a | α | Actual | 0.95 |

Fig. 5.5 Theoretical Iteration Count ( Fault-Free )

TABLE 5.1 Error Rate ( unit : % )

| Boundary Condition | Physical Constant | Number of Nodes | Min. | Max. | Average | S.D. |
|---|---|---|---|---|---|---|
| a | α | 32 | -3.67 | 0.0 | -0.02 | 0.07 |
| a | α | 128 | -1.99 | 1.23 | -0.01 | 0.07 |
| a | α | 800 | -0.57 | 1.08 | 0.02 | 0.05 |
| a | α | 2304 | -0.79 | 0.89 | 0.11 | 0.12 |
| a | β | 32 | -3.86 | 0.0 | -0.03 | 0.08 |
| a | β | 128 | -1.93 | 1.19 | 0.03 | 0.08 |
| a | β | 800 | -2.97 | 1.97 | -0.18 | 0.42 |
| a | β | 2304 | -3.25 | 3.78 | 0.06 | 0.39 |
| a | γ | 32 | -3.93 | 0.0 | -0.03 | 0.08 |
| a | γ | 128 | -3.90 | 2.24 | -0.11 | 0.43 |
| a | γ | 800 | -4.62 | 8.64 | -0.39 | 0.99 |
| a | γ | 2304 | -5.41 | 6.83 | 0.15 | 0.95 |
| b | α | 32 | -0.21 | 0.01 | -0.01 | 0.02 |
| b | α | 128 | -0.42 | 3.67 | 0.16 | 0.32 |
| b | α | 800 | -0.55 | 5.72 | 0.29 | 0.42 |
| b | α | 2304 | -0.46 | 7.38 | 0.64 | 0.73 |
| b | β | 32 | -0.14 | 0.0 | -0.01 | 0.01 |
| b | β | 128 | -10.53 | 3.80 | -0.01 | 0.88 |
| b | β | 800 | -3.68 | 5.63 | 0.14 | 0.64 |
| b | β | 2304 | -2.09 | 6.90 | 0.70 | 0.96 |
| b | γ | 32 | -0.13 | 0.0 | -0.01 | 0.01 |
| b | γ | 128 | -9.06 | 6.54 | 0.14 | 2.08 |
| b | γ | 800 | -10.92 | 8.42 | -0.21 | 1.36 |
| b | γ | 2304 | -5.54 | 11.75 | 0.80 | 1.88 |

# The Traveling Salesman Problem:
# The Development of a Distributed Computation

Nick Lai
*Computer Science Division*
*University of California, Berkeley*
*Berkeley, CA 94720*

Barton P. Miller
*Computer Sciences Department*
*University of Wisconsin*
*Madison, WI 53706*

## ABSTRACT

The Traveling Salesman Problem (TSP) is computationally expensive to evaluate. It can, however, be readily decomposed into subproblems that can be computed in parallel. Developing a distributed program taking advantage of such a decomposition, however, remains a difficult problem.

We developed such a distributed program to compute the TSP solutions, using a new set of distributed program performance tools to better understand our TSP program. These tools allowed us to discover the performance bottlenecks in our program and to revise the program to significantly improve its execution speed. This paper is a case study of the use of these performance tools.

### 1. Introduction

When a programmer implements a program that uses parallel computation, the goal is typically to increase the speed of the computation. Unfortunately, this goal is difficult to obtain. One reason for this is that communication between processes can be slow. This cost becomes greater when processes on different machines communicate. To obtain the goal of increased speed, the programmer must develop algorithms that minimize the effect of this communication cost.

It is essential for the programmer to be able to measure the performance of his or her program. The programmer needs to debug the program, and to know where the program is spending its time so that it can be modified or redesigned to increase its execution speed. The problems of debugging and performance monitoring become more difficult in a distributed environment due to a general lack of tools to aid the programmer. Tools for measuring distributed program performance have been implemented for Berkeley UNIX [4]. These tools allowed the development of the distributed program described here.

To examine the process of program development, we chose a problem that lends itself to this examination, the Traveling Salesman Problem. This problem is computationally intensive, but has well known solutions. We developed a program to find solutions to the problem without using parallel computation and, using that program as a basis, developed programs that would solve the problem using parallel processing. We debugged and measured our programs' execution using tools designed to passively measure parallel performance [5]. Using the information gained from these measurements, we improved our programs and measured how much these improvements enhanced performance. The TSP program development is a case study of the use of these measurement tools and of the program development process.

### 2. The Traveling Salesman Problem

### 2.1. Description of the Problem

The Traveling Salesman Problem (TSP) is a popular problem among both operations research experts and computer scientists. In general, the problem can be stated as follows: given N cities and the costs associated with going from each city to each of the other cities, find a minimum cost circuit that visits each of the N cities once and only once. Formally we have a *problem graph* G = (V, E) where V is the set of vertices (cities) and E is the set of edges, each representing the cost of going from each city to another. In a problem of size N, we have N vertices and m = N(N - 1) edges. The problem can also be represented as an N x N matrix, with each element $c_{ij}$ being the cost of going from city $i$ to city $j$. This matrix is called the *representative matrix* of the problem.

We address the problem in its most general form. In particular, we address the case of the asymmetric non-euclidean TSP. The problem is *asymmetric* because, in general, $c_{ij}$ does not equal $c_{ji}$. The problem is *non-euclidean* because the costs are arbitrary nonnegative numbers, unconstrained by the triangle inequality. There are many possible algorithms that can be used to efficiently solve the TSP. We have decided to follow the algorithm presented in [8].

### 2.2. General Algorithm

We use a *branch-and-bound* algorithm to solve the TSP. This algorithm is based upon the development of a *state tree* in which we record the paths that the TSP program has examined and the costs associated with those paths. Each node in the state tree corresponds to a collection of edges in the problem graph that make up a *partial path* through the problem. Each node of the state tree also contains a lower bound cost. The lower bound cost is the least cost that can be obtained by including that partial path in a full circuit. It is calculated by taking the representative matrix, and for each edge $c_{ij}$ setting all elements in row $i$ and column $j$ of the matrix to 0, and then reducing the matrix. The constants used to reduce the matrix are then added to the summation of the cost of each edge in the partial path; the result is the lower bound cost.

By examing the lower bound costs associated with the nodes of the state tree, the program can intelligently choose the best path to examine. This allows us a potentially large reduction in computation time with respect to that needed by solutions which compute the cost of every possible circuit through the problem. A decision to examine a particular path is called a *branching* decision, and it is based upon an examination of the lower bounds of all of the leaf nodes of the state tree [1]. Every time a branching decision is made, new nodes are created in the state tree that correspond to the possible extensions of the selected partial path.

Figure 2.1 shows an example of how a TSP problem is translated from a problem graph to a representative matrix to a state tree. In this example we have a small (three vertices, six edges) problem. The first representation is the problem graph, which shows the vertices and the directed edges of the problem. The representative matrix shows the cost of going from each of the three vertices to each of the other reachable nodes. Finally, the figure shows the development of the state tree. Vertex $A$ of the problem graph is arbitrarily selected as the root node of the state tree, and, by reducing the representative matrix, we find that the problem's lower bound is 17. We next set up the state tree nodes that correspond to the possible edges that include $A$, namely, $AB$ and $AC$. After setting row A and column B in the representative matrix to 0, we reduce the matrix with constants that sum to 10. Adding these constants to the cost of edge $AB$, which is 7, we find that the lower bound for a solution containing the edge $AB$ is 17. Similarly we find that the lower bound for $AC$ is 22. Examining all of the leaf nodes of the state tree, we decide to pursue the possibilities of a path containing $AB$, and set up nodes in the state tree for all of the problem graph edges that may be used after including $AB$ in a solution. There is only one such problem graph edge, $BC$, and a state tree node is set up for this edge, and its lower bound is computed to be 17. Once again, we examine the lower bound costs of the leaf nodes of the state tree, $BC$ and $AC$, and choose $BC$ because it has the lowest lower bound. Only one more edge exists beyond $BC$ : $CA$, which completes the circuit and solves the problem with a cost of 17, which is lower than any other possible circuit through the problem.

### 3. Computational Environment

The program was developed on a VAX 11/750 with 2 megabytes of memory. Occasionally, a other VAX'ss of the same configuration were used for data collection. These VAX computers are connected by a 3 megabit Ethernet. The operating system running on these VAXs is Berkeley UNIX 4.2BSD. This version of Berkeley UNIX supports virtual memory and processes with address spaces up to 16 megabytes. Most importantly, it supports interprocess communication [3]. We used the interprocess communication facilities built into 4.2BSD UNIX. The primary object of communication under this facility is the *socket*. A socket is an endpoint of communication. These sockets can be connected to form bidirectional communication streams.

## 4. Measurement Tools

The DPM measurement tools were used for this performance study. These tools are described in [4] and [5]. The raw data produced by the measurement tools is analyzed by a program that builds a graph of all interactions between the processes. The program uses this graph to compute the amount of parallelism achieved by the program, given various assumptions about message transmission costs. It can also project performance in the case of multiple CPU's for various assignments of processes to processors. The parallelism analysis is described in detail in [6].

The measurement procedure is an iterative one. First, we execute the distributed program, collecting the performance trace data. We may repeat this step, varying (for the TSP program) the problem size (number of cities) and number of processes that are used to computed the solution. Second, we analyze the performance data, computing the actual amount of parallelism, and computing the projected amount of parallelism for other assignments of processes to processors. Third, we use this data to modify our program. These three steps can be repeated until a satisfactory solution is reached.

The data analyses are being used to predict program performance. The question arises: How accurate are these predictions? While we do not test our predictions in the TSP study, previous case studies have shown the accuracy of these predictions to be within 5% of the actual program performance [7].

## 5. Evolution of a Program

The unit of measurement of performance that we use in this paper is the parallelism factor, **P**. This is a measure of speed-up in the computation, or how much of our computation is done in parallel. In the case where there is no parallel execution, **P** is 1; in the case where there are *n* processes in a computation and each process is concurrently performing $1/n$ of the work, **P** is equal to *n*. For this study, improving the performance of the computation means, for a given problem size (number of cities), increasing the value of **P**. In section 5.6 we comment on the effect that speed-up has on machine utilization.

In the measurements of the TSP programs, we use two forms of the factor **P**. The first form of **P** assumes that communication delays between processes are zero (i.e., communication is instantaneous), and that no process competes for the CPU with any other process. This gives an upper bound on the amount of parallelism that could be achieved by the measured execution of the program. The second form of **P** incorporates communication delays and contention caused by multiple processes executing on the same machine. The parallelism analysis can project the value of **P** for different assignments of processes to machines.

During the course of this research project, we developed three primary versions of the TSP program. These are referred to (in order) as the Non-Blocking Server, the Low-Node Caching and the Overlapping Requests versions. Note that each of these versions represents successive improvements to its predecessors, and *not* complete re-implementations.

Due to time considerations, we were not able to measure the performance of all of our implementations over a large variety of TSP problem sizes. For comparison purposes, all the implementations were measured for problems of size sixteen, with four, eight, twelve and sixteen processes.

### 5.1. Single Process Solution

We began by writing a single process implementation of our algorithm. This familiarized us with the TSP while ignoring the complexities of interprocess communication.

The basic algorithm of the program is as follows: First a matrix representing the problem is created. Second, the root node of the state tree is created. Each node of the state tree is identified with an edge's entry from the representative matrix, and represents a partial path through that matrix. This partial path can be obtained by traversing up the state tree to the root node. At each step of this traversal, the edge that the state tree node represents is a part of the partial path. The node also holds the lower bound cost associated with the partial path of which it is a part, and the lower bound cost of the solution that could be obtained by following that partial path. A node of the problem is arbitrarily selected as the starting point of the solution.

In each iteration of the main loop of the program, a node is selected as having the best possible chance of being on the optimal path. This is done by traversing the entire state tree and choosing the leaf node with the lowest lower bound value. This node represents the path most likely to yield an optimal circuit.

Child nodes are now set up for the chosen node. For each node that could be reached from the chosen node (that is, all of the reachable cities that are not already in the path), a child node is created, and lower bound costs for the child node are computed.

This loop is repeated until a complete path is found. The first complete path found will be an optimal path because the cost associated with it will be less than or equal to the cost associated with any other path.

### 5.2. Multiple Process Solution

There are many possible approaches to converting this computation into a parallel computation. We introduce parallelism by creating *child* processes that do some of the calculations needed by the *parent* process for each iteration. In other words, each child performs the part of the calculation that it is given, and all of the decisions are made by the parent process.

Most of the computation time is spent when new nodes are added to the state tree and the lower bound cost for each new node must be calculated. Our implementation of this program calculates these lower bound costs for each new node in parallel. When the program starts, a number of child processes are created. Instead of having the single parent calculate the lower bounds, these computations are handed to the child processes in the form of requests. Later, the results are returned to the parent, which stores the values in the appropriate node. There are other possible partitionings of this problem, but for this case study we chose a single design and attempted to improve its performance. single

In the first multiple process version, the parent sends requests to each of the children and then awaits responses from all of the children before sending out the next batch of requests. Although this version of the program ran and produced accurate answers, we did not start the actual performance measurements on our program until we had solved the simple problem described in the next section.

### 5.3. Non-blocking Server

After examining the program and its performance, it became apparent that most of the benefits of parallel execution were being lost because the parent process waited until all children sent back answers before sending out the next batch of computation requests. This was solved by having the parent hand out new requests to a child as soon as it returned an answer. We considered this version to be our first successful implementation, and the following measurements show its performance.

Figures 5.1, 5.2 and 5.3 show the projection of how this implementation of the program performed. Figure 5.1 shows the achievable degree of parallelism for problems with various matrix sizes and numbers of processes. This graph assumes that communication costs are zero, and that the processes do not compete with each other for the CPU. Figure 5.2 presents the performance of the program when each process is run on its own machine, over varying problem sizes and numbers of processes. Figure 5.3 illustrates the amount of parallelism achieved by our program when two processes are allocated to each processor under a variety of problem sizes and number of processes.

The results show that the amount of parallelism achieved increases with the size of the problem. As the problem size increases, a larger percentage of the program's time is spent computing the lower bounds. Since we do the computation of the lower bounds in parallel, it is natural that the degree of parallelism should increase with the problem size.

If we compare the corresponding curves from graphs 5.1 and 5.2 for a single problem size, we can see the cost (loss of parallelism) from communications delays. Comparing graphs 5.2 and 5.3, we see the loss of parallelism when two processes contend for a single CPU.

### 5.4. Low-Node Caching

The process status of the running program showed that it was using large amounts of memory. This memory was obviously being used for the state tree. However, this meant that we were spending a large percent of program time searching through the state tree for the lowest node. During this time the child processes were sitting idle, and parallelism was being lost.

To eliminate this waiting time, we decided to save 100 of the previously calculated nodes with the lowest values in an array. For each child node that is created, if the array is not full or if the value of the child node is less than the cost of the last (highest value) element in the array, this value is inserted, in order, into the array. The result is an ordered array of the leaf nodes from the state tree with the smallest values. Our branching decision is made by selecting the leaf node of the state tree that has the smallest value associated with it, in this case the first node in the array. When this node is selected, it is eliminated from the array.

Performance measurement of this Low-Node Caching version of our program shows a significant improvement in the parallelism achieved by the program. Our measurements where made for problems of size 16 with 4, 8, 12 and 16 processes. Figure 5.4 compares this Low-Node Caching version with the Non-Blocking Server version. These figures show that this Low-Node Caching version performed significantly better than the non-blocking server implementation. We also find that this improvement in performance is more pronounced as the number of processes increases.

This improvement is the result of the parent process doing less computation relative to the computations it gives to its children processes. That is, less time is spent in the parent process in this version, while the children are still doing the same amount of computation as in the Non-Blocking Server version.

### 5.5. Overlapping Requests

Experimental measurements show that communications costs are actually very high, taking approximately 8 ms for communications between processes on a single machine and 20 ms for processes on different machines [2]. This means that a child process remains idle while information is transferred back to its parent and while waiting for a new request to arrive from the parent.

To eliminate the idle time in the case where the number of requests exceeds the number of servers, we overlapped the requests to the children by initially sending out two requests to each child. Children no longer had to wait for the parent to receive the result before beginning work on a new request, since another request was usually waiting. This is an important innovation, and is aimed solely at trying to cut down the effect of the communications cost. As the following results show, this minor change enhanced the program's performance.

Figure 5.5 shows the performance of the Overlapping Requests implementation, as well the performance of the previous two versions, for a problem of size 16 with 4, 8, 12 and 16 processes.

Note that the Overlapping Requests version performs better than the Low-Node Caching version when the number of processes is small, but that the difference in performance between the two diminishes as the number of processes approaches the size of the problem. If there are as many processes as the size of the problem, then all of the computations are done by having each child do only one computation, thus there is no opportunity to use overlapping to reduce communication delay effects.

#### 5.5.1.1. Is Faster Always Better?

In this study, we have based our evaluation of program performance on the amount of parallelism or speed-up achieved in the different versions of the TSP program. By using P as our only evaluation criterion, we have stated: faster is better. This view of performance dictates that even a small increase of parallelism is worth the addition of more hardware.

It is also important to know how well we are using our computing resources. This can be stated as: how much of our available computing resources are we using? For the TSP programs, we ask: how much of the machines (CPU's) involved in the computation are we utilizing? We can compute CPU utilization from the parallelism factor. We define $\text{Utilization} = P/N$, where $N$ is number of machines used in the computation. Figure 5.6 graphs the values of the utilization (corresponding to the graphs in Figure 5.5) for the three versions of the TSP program.

Our first observation from the graph in Figure 5.6 is that when two processes run on each machine, we have a higher utilization (and from Figure 5.5, a lesser amount of parallelism) than when one process runs on each machine. This means that while running one process per machine is a less efficient use of resources, it results in a faster execution.

Our second observation from Figure 5.6 is that each successive version of the TSP program made better (more efficient) use of the CPU's. We know that the successive versions are better (and not just requiring more computation time) since the amount of CPU time needed for each successive version also decreased. The fact that the curves for the Overlapping Requests version and the Low-Node Caching version meet when 16 processes are used reflects the fact that the problem size is the same as the number of processes.

A person who uses a distributed program, such as the TSP program, wants his/her program to execute as quickly as possible. The parallelism factor, P, gives a measure of how much we can increase the speed of execution. It is also important to know how efficient is a program. We need metrics such as CPU utilization to evaluate efficiency.

## 6. Conclusion

The purpose of this study was to evaluate a collection of measurement tools for distributed programs through the development of a simple program. The question was: could these tools help us better understand our program, and help us improve its performance?

While the main intent of this study was to evaluate the performance monitoring abilities of the measurement tools, these tools also proved useful for debugging. When the first version of the TSP program was being tested, its execution speed seemed slower than expected. A visual examination of the trace records produced by the measurement tools quickly (that is, in less than one minute) showed that the master process in the computation was only creating a single child process. This was caused by a simple typing error in the program. After this error was corrected, the performance seemed unchanged. A second inspection of the traces showed that, while the correct number of child processes were being created, the master was sending requests only to a single child – again a typing error. After this, the program worked correctly. The discovery of these errors is an indication that the measurement tools are providing some of the additional information needed for developing distributed programs.

The main direction of this study was the investigation of the performance of the TSP program. In particular, we wanted to maximize the amount of parallel activity in the program. The distributed program measurement tools, in conjunction with other existing tools, provided the information necessary to measure the program's performance, identify bottlenecks, correct the problem, and evaluate the corrections.

Traditional performance tools provide information as to *how much* time a program spends in its various procedures and modules. Our measurement tools in conjunction with the analysis of parallelism also provide an insight into *when* a program spends its time in these modules.

## References

[1] E. Horowitz & S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, 1976.

[2] E. Hunter, "A performance study of the Ethernet under Berkeley UNIX 4.2BSD", Technical Report UCB/CSD, University of California, Berkeley, 1984.

[3] S. Leffler, W.N. Joy, and R. Fabry, "4.2BSD Networking Implementation Notes," Technical Report UCB/CSRG, University of California, Berkeley, July 1983.

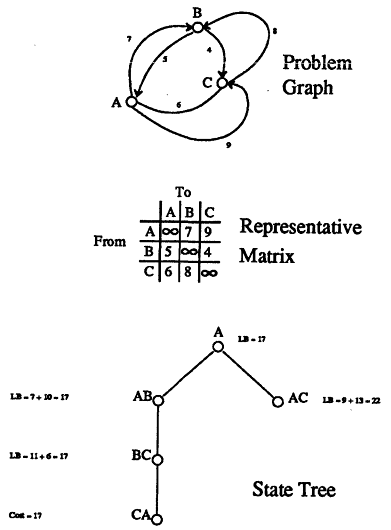
[4] B.P. Miller, S. Sechrest, and K. Macrander, "A Distributed Program Monitor for Berkeley UNIX", *Software - Practice & Experience*, 11, 2 (February 1986).

[5] B.P. Miller, "DPM: A Measurement System for Distributed Programs", Computer Sciences Technical Report 592, University of Wisconsin, Madison, May 1985. Submitted for publication.

[6] B.P. Miller, "Parallelism in Distributed Programs: Measurement and Prediction", Computer Sciences Technical Report 574, University of Wisconsin, Madison, May 1985. Submitted for publication.

[7] B.P. Miller, "Performance Characterization of Distributed Programs", Ph.D. Dissertation, Technical Report UCB/CSD 84/197, University of California, Berkeley (May 1984).

[8] J. Mohan, "A study in Parallel Computation -- The Traveling Salesman Problem," Technical Report CMU-CS-82-136, Carnegie-Mellon University, August 1982.



|  | A | B | C |
|---|---|---|---|
| A | ∞ | 7 | 9 |
| B | 5 | ∞ | 4 |
| C | 6 | 8 | ∞ |

Figure 2.1: Representations of the TSP

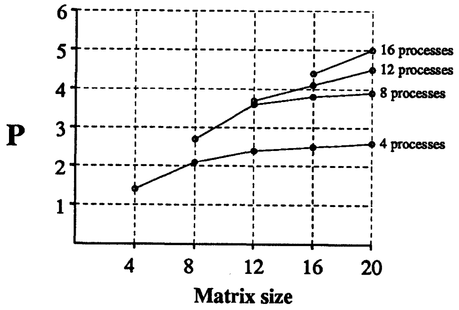


**Figure 5.1:**
**Upper Bound Parallelism of the Non-Blocking TSP Program.**

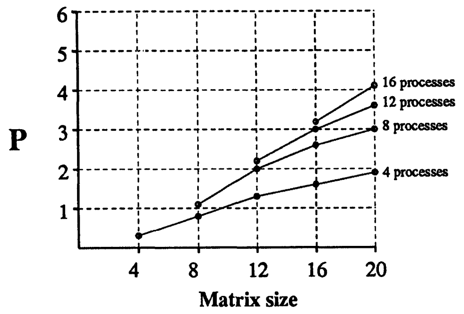


**Figure 5.2:**
**Parallelism vs. Matrix Size, One Process per Machine.**

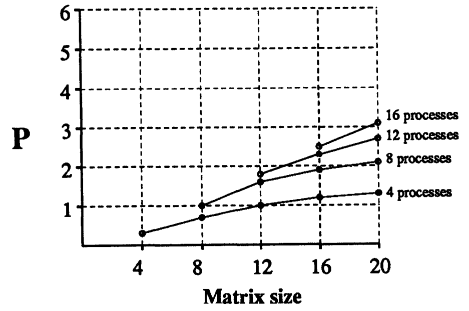


**Figure 5.3:**
**Parallelism vs. Matrix Size, Two Processes per Machine.**

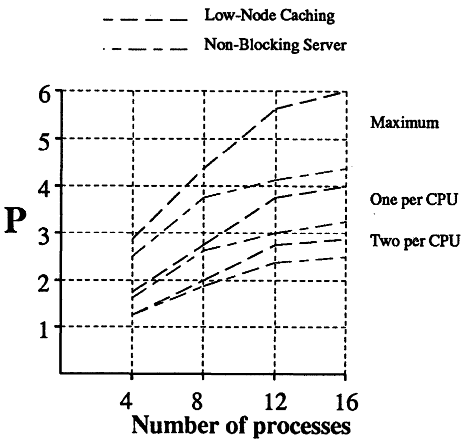


**Figure 5.4:**
**Low-Node Caching versus Non-Blocking Server**

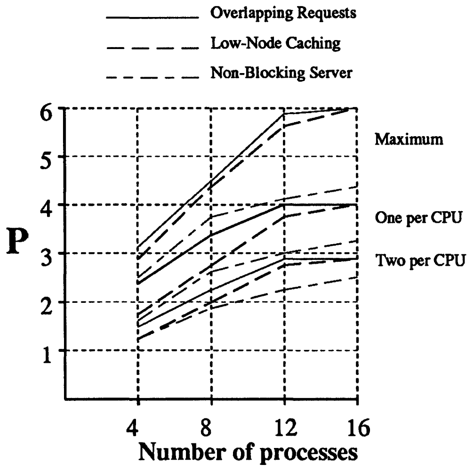


**Figure 5.5:**
**Comparison of the Three Implementations.**

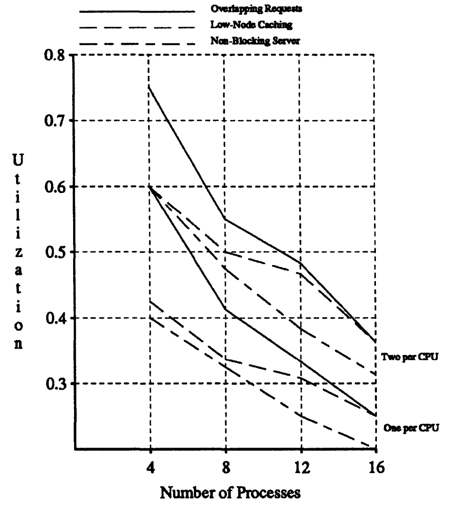


**Figure 5.6:**
**Machine Utilizations for the Three Implementations**

# Optimized Demand-driven Evaluation of Functional Programs on a Dataflow Machine

*Satoshi ONO* , *Naohisa TAKAHASHI* *and* *Makoto AMAMIYA*

NTT Electrical Communications Laboratories
Nippon Telegraph and Telephone Corporation
3 - 9 - 11 Midori-cho Musashino-shi Tokyo 180 Japan

## Abstract

This paper presents a new program transformation algorithm for functional programs named the mixed-mode fixpoint evaluation method. This method realizes optimized demand-driven computation on a dataflow machine. This evaluation scheme takes advantage of both data-driven and demand-driven evaluations in that it offers sufficient parallelism and safeness for nonstrict sequential functions. This can be achieved by eliminating demand propagation predictable during compilation time, and by replacing the lengthy recipe creation and evaluation required in demand-driven function applications with the call-by-value method. These optimizations are based on an inter-function global dataflow analysis called the Extended Dependency Property Set.

## 1. Introduction

Functional programming languages have various attractive features facilitating the writing of short and clear programs, as well as verifying and transforming programs automatically due to their formal mathematical semantics. For the efficient execution of functional programs, several language-oriented machines based on dataflow and reduction models have been proposed and implemented.[15]

Although dataflow machines inherently have conformed excellently to a special subset of functional programming languages[3], several drawbacks exist[3] in dataflow models compared to reduction models. One of these is the difficulty in implementing the fixpoint computation rule[8] for nonstrict[16] functions. In order to overcome this drawback, several studies have been conducted in an attempt to achieve demand-driven computation on dataflow models. These studies have focused on demand-controlled stream creation,[7,12] the *lazy cons* operator[2] and the semi-result concept.[17] These approaches extend the computation domain to nonstrict data structures such as infinite sequences. However, they have only limited capability for handling nonstrictness of sequential[16] functions.

In addition, several inefficiencies commonly arise during demand-driven evaluation. One is the cost of demand propagation from the outermost expression to the innermost expressions. An algorithm based on simplified global dataflow analysis has been proposed.[14] Such a simple method, which first makes the assumption that all nonprimitives are nonstrict (or strict), however, has been shown to have only a restricted analysis capability.[4] Another inefficiency is the cost required for creating and evaluating recipes.[6] The recipe-based demand-diven method (call-by-recipe) requires much more complicated computation than does the simple call-by-value method adopted in the data-driven evaluation.

This paper proposes a new method for realizing an optimized fixpoint computation rule on dataflow machines, named the *mixed-mode fixpoint evaluation method.* The name is so chosen because the method realizes the fixpoint computation rule for the class of sequential functions, and because both data-driven and demand-driven evaluations are scattered throughout the computation process.

The method consists of two principal algorithms: the *demand propagation algorithm* and the *recipe creation elimination algorithm.* The demand propagation algorithm predicts successive demand propagation sequences by analyzing the dependency between functions and values. Then, it transforms dataflow graphs such that each demand is propagated directly toward the innermost expressions as long as the safeness property is preserved. The recipe creation elimination algorithm analizes the lifetime of recipes and detects all cases where a call-by-recipe can be safely replaced with a call-by-value.

To insure the safeness of the above optimizations, these algorithms make use of an inter-function global dataflow analysis named the *Parameter Dependency Property Set* (PDPS).[9,10] In this paper, the concept of the PDPS is generalized to the *Extended PDPS* (EDPS) in order to process the recipe lifetime property. In order to reduce the amount of computation required for the optimization, a hierarchical approach is adopted. As the first step, a program is analyzed in the inter-function level. Then, each function is analyzed and optimized internally based on the collected global data.

The algorithms proposed in this paper make use of a hardware-implemented lazy synchronization mechanisms,[2] such as that used in the Structure Memory[1,5] for storing structured data. In combination with this kind of intelligent operation memory, the proposed method realizes the fixpoint computation rule for the class of sequential functions on nonstrict data structures.

0190-3918/86/0000/0421 $01.00 © 1986 IEEE

## 2. Background

### 2.1 Global Dataflow Analysis

Mixed-mode fixpoint evaluation requires that evaluation of a function application be initiated only if the result contributes essentially to the final computation result. In contrast, the hardware-supported redex (*reducible expression*) detection mechanism in dataflow machines is so restricted that only local information about value availability is used as a criterion for initiating computation. This gap inevitably leads us to a global dataflow analysis which gathers the property data of program fragments (function, function application, etc.) that can be made apparent only by analyzing the entire program text.

For reducing the computation required for such global analysis, the concept of *autonomousness* is introduced. Suppose that an inter-function analysis computes the property $P$ for each function in a program. Property $P$ is said to be autonomous if the property $P$ of a function $f$ is computable using only the property data of functions referred to by the function $f$ (i.e., computable without using the definitions of referred functions).

The result of the conventional strictness analyses,[4,13] i.e., requisite parameters, is not autonomous. For example, suppose the program

$$f(x,y) = g(x, x+y, x-y) \;;$$
$$g(x,y,z) = \textbf{ if } x \geq 0 \textbf{ then } y \textbf{ else } z \textbf{ fi } .$$

Function $f$ refers to $g$, and the requisite parameter of $g(x,y,z)$ is $x$. The requisite parameters for $f(x,y)$ are $x$ and $y$. However, this fact cannot be deduced from the requisite parameter information concerning $g$.

The autonomousness of property $P$ is desirable, since computation required for the inter-function analysis can be reduced using the caller-callee relation of the functions in a program. In addition, autonomousness means that the property contains enough information to enable computation of the property of the elements referring to it. Therefore, inter-function property data can be propagated to clarify the property of each element in a function without referring to the entire program text.

### 2.2 Parameter Dependency Property Set (PDPS)

The PDPS is the generalization of requisite parameter in such a way that it satisfies autonomousness. The PDPS for a function $f$ is a set of *Minimally Sufficient Parameter Sets* (MSPSs) for $f$ that is defined below.

[Definition 1] Minimally Sufficient Parameter Set

Given an $(m+n)$-ary function, $f(x_1, x_2, \ldots, x_m, y_1, \ldots, y_n)$, a set of parameters $\{x_1, \ldots, x_m\}$ is said to be minimally sufficient if and only if there exist values $a_1, \ldots, a_m$ such that

(a) $f(a_1, \ldots, a_m, y_1, \ldots, y_n)$ gives a defined value even if $y_1, \ldots, y_n$ are undefined.

(b) Suppose $y_1 \equiv \omega, \ldots, y_n \equiv \omega$ where $\omega$ represents an undefined value. Then, $f(a_1, \ldots, a_m, y_1, \ldots, y_n)$ becomes undefined if at least one of the values $a_1, \ldots, a_n$ are replaced with $\omega$.

The PDPS computation algorithm for monotonic recursive function systems has been previously provided.[9,10] The PDPS of an expression, defined similarly regarding free variables of the expression as parameters, can be computed using the algorithm called PDPS reduction.[9] It is apparent that the intersection of all MSPSs in the PDPS of function $f$ becomes the set of requisite parameters of $f$.

For example, consider the function, $g(x,y,z)$, defined in Subsection 2.1. $D(g(x,y,z)) = \{\{x,y\},\{x,z\}\}$, where $D(e)$ stands for the PDPS of $e$. The PDPSs of subexpressions $x+y$ and $x-y$ are both $\{\{x,y\}\}$. The PDPS of the function $f(x,y)$ becomes $\{\{x,y\}\}$. This means that the requisite parameters of $f(x,y)$ are $x$ and $y$.

## 3. Realization of Mixed-mode Fixpoint Evaluation

### 3.1 Value Property Classification

Parameters and free variables (hereafter, simply referred to as values) of a function can be discussed in terms of two properties in combination with each other: *requisiteness* and *recipe lifetime.* Requisiteness refers to the causal relation between values, whereas recipe lifetime means the form of value references in the function body.

Regarding requisiteness, a value is classified as either *requisite, suspended* or *irrelevant.* A requisite value is a value such that if it becomes undefined, the result always becomes undefined. The intersection of all MSPSs in the PDPS of the function gives the set of requisite values. If there exists a case such that the result becomes undefined when the value becomes undefined, but the value is not requisite, it is said to be suspended. $S(f)$, the set of suspended values for a function $f$, can be computed as

$$S(f) = \langle \text{ Union of all MSPSs of } f \; \rangle - \langle \text{ Requisite values of } f \; \rangle \,,$$

where "−" is a set difference operator. A value is said to be irrelevant if the result is computed independently of the value.

From the viewpoint of recipe lifetime, a value is classified as either *scalar* or *nonscalar.* A value is said to be scalar if the value always requires evaluation when it is referred. In contrast, a value is said to be nonscalar if a case exists such that the result of a function application is computable using the value only in its recipe form. In other words, if the demand to the value does not necessarily mean the evaluation of the expression that gives the value, the parameter is said to be nonscalar. To give an example, numerical values in arithmetic expressions are scalar, while parameters of a *lazy cons* operator are nonscalar.

### 3.2 Mixed-mode Fixpoint Evaluation

The principle of mixed-mode fixpoint evaluation is that actual parameter passing is deferred until the parameter becomes requisite for computing the resulting value, and that the parameter is passed on in an evaluated form whenever evaluation of the actual parameter is safe.

To achieve the above principle, two primary function application criteria are adopted. First, if a parameter is requisite, the actual parameter is unconditionally passed on to the function body. If a parameter is suspended, it is passed on only if the expanded function sends a demand for it. If a parameter is irrelevant, no code for it is generated.

Second, if a parameter is scalar and it is requisite or demanded, the actual parameter is evaluated, and the resulting value is passed on to the expanded function. Thus, if the actual parameter is a recipe, it is forced at first, and then passed on. If a parameter is nonscalar and it is requisite or demanded, a recipe for the expression that gives the actual parameter is created (if not already created), and a pointer to the recipe is passed on to the expanded function. If the recipe is forced in the body of the expanded function, the evaluation of the recipe is initiated with the resulting value taking the place of the recipe as in call-by-need computation.[6] In this paper, the recipe creation and evaluation mentioned above are assumed to be executed in an intelligent operation memory such as a Structure Memory. For details of the implementation, please refer to [2].

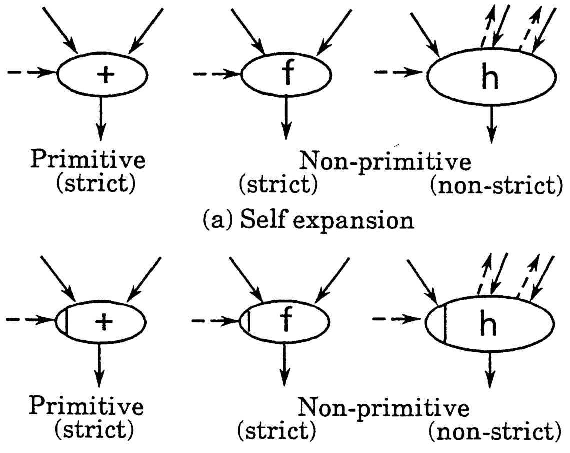
The function application method of the mixed-mode fixpoint evaluation is an optimized version of a joint data-driven, demand-driven scheme in terms of three principal points. First, evaluated values are passed on instead of recipes to an expanded function if scalar parameters are requisite or demanded. Second, recipe creation for a suspended nonscalar parameter is delayed until it is actually demanded. Third, the method completely eliminates computation of irrelevant parameters.

### 3.3 Function Expansion Strategy

The function expansion strategy controls the function expansion time during function application. One strategy is that a function is expanded when all requisite parameters are available. This is a natural extension of the data-driven computation for strict functions, and is called the *strict expansion strategy*. Functions may be expanded at the time the first requisite parameter becomes available. This is called the *hasty expansion strategy*. In this paper, the strict expansion strategy is assumed. If the hasty expansion strategy is adopted, the algorithm in Section 4 should be slightly modified.

Figure 1(a) is a graphic representation of function applications. Primitives and strict functions are computed in a data-driven manner. For each suspended parameter of a nonstrict function, a *demand arc* (dashed arrow) extends from the node. When a token is put on this arc, the corresponding parameter is demanded. Conditional function "**if_then_else**" is also represented as one of the nonstrict functions, although it is a macro-node expanded in-line during program compilation.

In demand-driven computation, the case exists in which function expansions should be controlled by demands. This is because a function should be applied if and only if the resulting value of the function application is requisite. For this purpose, we introduce a new notation for the dataflow graph as shown in Fig. 1(b). The horizontal dashed arrow is a demand arc for function expansion. This notation is referred to as *demand expansion*. In the strict expansion context, a function is expanded if and only if all requisite parameters are available and a demand token exists on the demand arc. In contrast, conventional notation is referred to as *self expansion*.



(a) Self expansion

(b) Demand expansion

Fig. 1 Graphic Representation of Nodes

## 4. Demand Propagation Algorithm

During the function application of the mixed-mode fixpoint evaluation, the necessity for a suspended parameter is signalled by a demand token from the expanded function body. For a scalar parameter, this demand is propagated to evaluate the corresponding actual parameter. The proposed method optimizes the propagation using the result of inter-function analysis. This section describes the outline of this optimizing algorithm.

### 4.1 Principle of Demand Propagation

In the evaluation process of a function application, value tokens are generated as a result of sub-expression evaluation. Such tokens are named *Partial Resulting Values* (PRVs). In the following, two important concepts regarding PRVs and demands are introduced.

[Definition 2] Total Requisite Value Set (TRVS)

For a PRV $v$, a set of PRVs and/or parameters (hereafter referred to as values) exists such that the values in the set are requisite to the evaluation of $v$. This set is named the *Total Requisite Value Set* of $v$. The PRV $v$ itself is also included in $T(v)$, where $T(v)$ stands for the TRVS of $v$.

[Definition 3] Demand-generated
Requisite Value Set (DRVS)

Suppose that the function definition contains a function application of a nonstrict function, and that $dv$ is a demand arc for the suspended parameter $v$ of this function application. A set of values would then exist such that the values in the set become requisite when a demand is generated on the demand arc $dv$. This set is named the *Demand-generated Requisite Value Set* (DRVS) of $dv$, and is indicated by $R(dv)$.

Using the above concepts, the optimization principle is described as follows.

(1) Suppose that PRV $v$ is a result of some function application, and that PRV $r$ is an actual parameter corresponding to the requisite parameter of the applied function. If PRV $v$ becomes requisite, the PRVs in $T(r)$ are evaluated in a data-driven manner.

(2) Suppose that PRV $s$ is an actual parameter corresponding to the suspended parameter of some function application. If this parameter becomes requisite in the applied function body, it is signalled to the caller of the function through the demand arc $ds$. With this demand signal, the PRVs in $R(ds)$ are evaluated in a data-driven manner.

An outline of the computation algorithm for the TRVS and the DRVS will be discussed in Subsection 4.3.

### 4.2 Example of Demand Propagation

The optimized demand propagation described in the previous subsection is accomplished through the transformation of dataflow graphs. This transformation process and the operation of the transformed program are intuitively explained using an example.

Consider the following function:

$$\begin{aligned} p(x,y,z) = \{c = f_2(z);\quad & i = f_3(y);\\ e = h_2(g_1&(y,z), c, g_2(c,i));\\ j = h_1(f_1&(x), e, i);\ \textbf{return}\ j\}, \end{aligned}$$

where $f_k$ $(k = 1,2,3)$ are unary strict functions and $g_k$ $(k = 1,2)$ are binary strict functions. In addition, the PDPS of the nonstrict function $h_k(x,y,z)$ $(k = 1,2)$ is assumed to be $\{\{x,y\},\{x,z\}\}$. All values are assumed to be scalar.

Figure 2 shows a data-driven version of the function codes. For each PRV in this figure, a unique name is attached. All parameters for $p$ should be evaluated prior to the function application. At the beginning of the transformation, a demand arc is created for each requisite parameter. Figure 3 shows this partially transformed graph. The PDPS of $p(x,y,z)$ is computed to be $\{\{x,y\},\{x,y,z\}\}$. Therefore, $x$ and $y$ are requisite parameters, and $z$ is a suspended parameter. In this figure, the demand arc for $z$ is represented as $dz$, a naming convention which will be used throughout this paper. Nonstrict functions $h_1$ and $h_2$ should also be transformed in the same way as function $p$. Strict functions require such transformation only if they contain nonstrict functions.

The completely transformed program is shown in Fig. 4. In the figure, a new primitive node, "*Demands Merge, Value Distribution*" (DMVD), is introduced. This node corresponds to the combination of a *d-union operator*[12] and a distribution node. The DMVD operator accepts multiple (demand input, value output) arc pairs as well

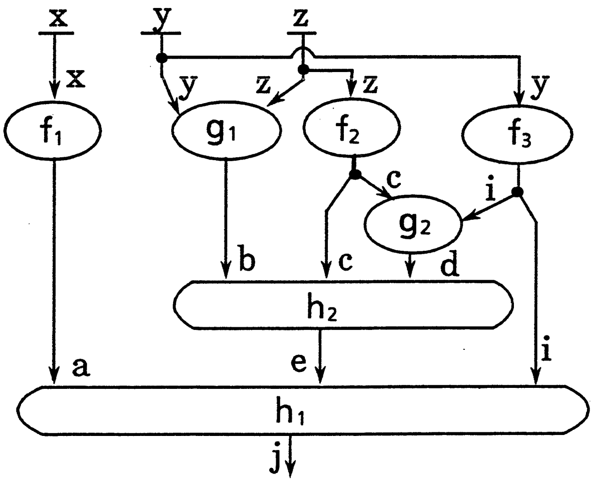


Fig. 2 Original Data-driven Program

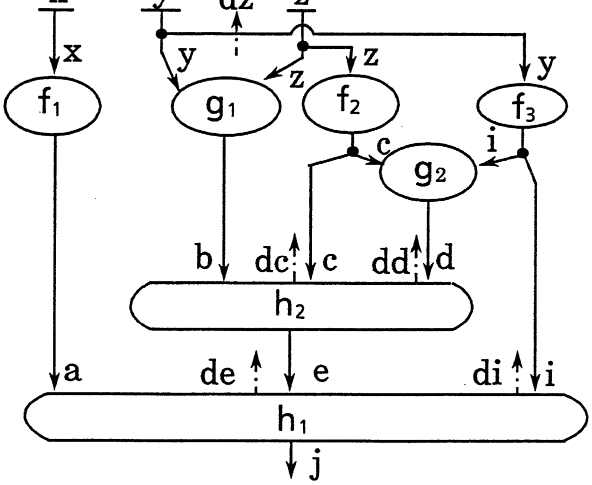
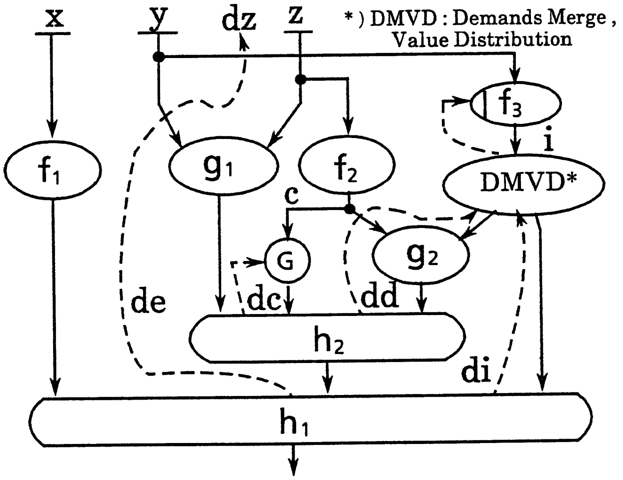


Fig. 3 Partially Transformed Program



*) DMVD : Demands Merge, Value Distribution

Fig. 4 Completely Transformed Program

as one ⟨demand output, value input⟩ arc pair. The operator receives at most one demand token from each demand input arc. After receiving the first demand token, the demand token is forwarded through its demand output arc. When a value comes to its value input arc, it is distributed to all value output arcs that have demanded the value through associated demand arcs. If the value is available when a demand comes, it is immediately forwarded to the associated value arc. Another new primitive is the *gate* (represented as **G** in the figure). The *gate* primitive is defined as a strict function, $gate(x, y) = x$. The arc corresponding to parameter $y$ of $gate(x, y)$ is called the *control* arc (described as a horizontal arc in the figure).

When function $p$ is applied, PRV $j$, the final PRV to be returned, is requisite. In addition, $T(j) = \{x, y, a, j\}$. Therefore, parameters $x$ and $y$ are passed on in evaluated forms, and the PRVs $a$ and $j$ are computed in a data-driven manner. When the second parameter of function $h_1$ becomes requisite, demand $de$ is generated. Since $de$ is connected to $dz$, the suspended parameter $z$ is evaluated and passed on. The PRVs $b$, $c$ and $e$ are then evaluated in a data-driven manner because $R(de) = \{b, c, e, z\}$.

### 4.3 Outline of Demand Propagation Algorithm

Since the demand propagation algorithm explanations are considerably lengthy and complicated, the algorithm is explained only intuitively in this paper. For details, please refer to [11].

The demand propagation algorithm consists of four phases. In Phase 1, the PDPS is computed for every function defined in a program. This phase is an inter-function analysis. In Phase 2, each function is analyzed using PDPSs. As a result, the *Value Dependency Property Set* (VDPS) is computed. As will be described below, the VDPS is the generalization of the TRVS in the same way as requisite parameters are generalized into the PDPS. This phase and the succeeding phases are intra-function analyses. Phase 3 analyzes each suspended parameter of the function application in a function definition, and computes the DRVS for every demand arc. In Phase 4, demand arcs are connected to proper points such that each demand arc triggers evaluation of values belonging to its DRVS.

In the following, Phases 2-4 are explained using the function example $p(x, y, z)$ introduced in the previous subsection. Here, we assume that the PDPSs of all functions referred to in $p$ have already been computed as a result of Phase 1.

[Phase 2] The final value returned by function $p$ is PRV $j$. In Fig. 2, parameters $x$, $y$, $z$ and PRVs $a, \ldots, e$, $i$ are also included. However, not all values are required to compute PRV $j$, since functions $h_k(x, y, z)$ $(k = 1, 2)$ are nonstrict. For example, if function $h_1$ demands the third parameter, only values $x, y, a$ and $i$ are required to compute PRV $j$. Such a set of values is named the *Minimally Sufficient Value Set* (MSVS) of PRV $j$. PRV $j$ itself is included in each MSVS of $j$. The set of all MSVSs of $j$ is called the VDPS of $j$, and is indicated by $V(j)$. For example, $V(j)$ in Fig. 2 is computed as

$$V(j) = \{\{a, b, c, d, e, i, j, x, y, z\}, \{a, b, c, e, j, x, y, z\}, \{a, i, j, x, y\}\} .$$

The intersection of all MSVSs in $V(j)$ gives $T(j)$. As a result of Phase 2, the VDPS of the final value is computed for each function in a program. Since the dataflow graph of a function is acyclic, the VDPS computation algorithm is similar to the PDPS computation algorithm for a block expression.[9]

[Phase 3] In Phase 3, the DRVS is computed for each demand arc. In general, when a demand $dv$ is generated for value $v$, some values have already been demanded or evaluated. A set of these values is named the *Pre-demand Requisite Value Set* of $dv$, and is indicated by $Pre(dv)$. Similarly, the *Post-demand Requisite Value Set* is defined as a set of demanded / evaluated values after demand $dv$, and is indicated by $Post(dv)$. The DRVS of $dv$, namely, $R(dv)$, is computed as

$$R(dv) = Post(dv) - Pre(dv) .$$

For example, consider the demand $di$ in Fig. 3. In order to generate demand $di$, evaluation of PRV $a$ as well as parameter $x$ must be finished. In addition, PRV $j$ must be demanded. Furthermore, parameter $y$ is requisite to compute PRV $j$. Therefore, $Pre(di) = \{a, j, x, y\}$. Similarly, $Post(di)$ becomes $\{a, i, j, x, y\}$. Therefore, $R(di) = \{i\}$. Similarly, $R(de) = \{b, c, e, z\}$, $R(dc) = \{\}$ and $R(dd) = \{d, i\}$.

Let us now discuss the algorithm for $Pre(di)$ and $Post(di)$ using Fig. 3. The demand $di$ is a demand for PRV $i$, and the PRV $i$ is used to compute the PRV $j$. At the time the demand $di$ is generated, PRV $j$ has already been demanded. After this demand, PRV $i$ also becomes demanded. Assume $V_p$ to be the VDPS of the final PRV of function $p$. In this case, $V_p$ is equal to $V(j)$. Then, each MSVS in $V_p$ represents a possible combination of the PRVs demanded / evaluated during the function application of $p$. Thus, $Pre(di)$ is computed as the intersection of all MSVSs in $V_p$ that contain PRV $j$. Similarly, $Post(di)$ is the intersection of all MSVSs in $V_p$ that contain PRVs both $i$ and $j$.

[Phase 4] Each demand arc is used as a trigger for the evaluation of values belonging to its DRVS. In this phase, the values in a DRVS of each demand are classified by the availability of requisite values. This algorithm is explained below using the example presented in Figs. 3 - 4.

Consider the demand $de$ whose DRVS is $\{b, c, e, z\}$. The parameter $z$ is a suspended parameter of function $p$, and demand arc $dz$ is associated with $z$. PRVs $b$, $c$ and $e$ cannot be evaluated without parameter $z$. In addition, if $z$ is evaluated, the requisite parameters for these PRVs become available. Therefore, demand arc $de$ is connected to $dz$.

Next, consider the demand $dd$ where $R(dd) = \{d, i\}$. PRV $d$ cannot be evaluated without PRV $i$. In contrast, the requisite value for PRV $i$, namely parameter $y$, is passed on in an evaluated form. Thus, the function application $f_3(y)$ is demand expanded using demand arc *dd*.

For the demand *dc*, the DRVS is empty. This fact means that although *dc* is a demand for PRV $c$, the value is already requisite even before it is demanded. Therefore, a *gate* node, **G** is inserted as shown in Fig. 4. The parameter passing of PRV $c$ is deferred by this node until demand *dc* is generated.

If more than two demand arcs are connected to one node, a DMVD node is inserted for merging demands and distributing the resulting value. For example, in Fig. 3, function application $f_3(y)$ should be demand expanded by both demands *dd* and *di*. Therefore, a DMVD node is inserted in Fig. 4.

## 5. Recipe Creation Elimination Algorithm

### 5.1 Extended Dependency Property Set

Parameters for primitives for arithmetic, logical and relational operations such as "+", "*and*", ">" are all requisite and scalar. By contrast, nonstrict data constructors such as *lazy cons* require all parameters to be requisite and nonscalar (i.e., recipes should always be passed). In this section, we describe the computation algorithm for these properties for the class of monotonic recursive functions.

For non-recursive functions, the recipe lifetime analysis is rather self-evident. For example, consider the function $f$ such that

$$f(x,y,z) = \textbf{ if } x > 0 \textbf{ then } x + y \textbf{ else } cons(z, \text{NIL}) \textbf{ fi.}$$

In function $f$, parameter $x$ is requisite, and parameters $y$ and $z$ are suspended. Furthermore, $x$ and $y$ are scalar, and $z$ is nonscalar.

An *Extended Dependency Property Set* (EDPS) notation is introduced in order to describe both requisiteness and recipe lifetime properties in one data structure. The EDPS is defined as a set of *Extended MSPSs* (EMSPSs). The EMSPS is a set of ordered pairs ⟨value name, recipe lifetime property⟩. For example, $E(f(x,y,z))$, the EDPS of above function $f(x,y,z)$, is described as

$$E(f(x,y,z)) = \{\{\langle x,S\rangle, \langle y,S\rangle\}, \{\langle x,S\rangle, \langle z,N\rangle\}\},$$

where $\langle x,S\rangle$ and $\langle z,N\rangle$ mean that value $x$ is scalar, and value $z$ is nonscalar, respectively.

### 5.2 EDPS Computation for Recursive Functions

Consider the composite function, $f(x) = f_1(f_2(x))$. Parameter $x$ is scalar only if the parameters of both $f_1$ and $f_2$ are scalar. For example, parameter $x$ of $f(x) = cons(2 * x, \text{NIL})$ is nonscalar, whereas $f(x) = (2 * x) + 3$ is scalar.

For general composite functions, the EDPSs can be computed using the above rule and the general PDPS reduction rule.[9] At this time, the same parameter can be used in both scalar and nonscalar form. An example of this conflict is shown below as

$$f(x,y,z) = \textbf{ if } x > 0 \textbf{ then } cons(x,y) \textbf{ else } y + z \textbf{ fi.}$$

The EDPS of $f(x,y,z)$ becomes

$$\{\{\langle x,S\rangle, \langle x,N\rangle, \langle y,N\rangle\}, \{\langle x,S\rangle, \langle y,S\rangle, \langle z,S\rangle\}\}$$

if $\langle x,S\rangle$ and $\langle x,N\rangle$ are distinguished.

If the same parameter is classified as scalar and nonscalar in a single EMSPS, it can be safely classified as scalar. This is because if the computation of $x$ diverges, the final value of the function application always becomes undefined. Therefore,

$$E(f(x,y,z)) = \{\{\langle x,S\rangle, \langle y,N\rangle\}, \{\langle x,S\rangle, \langle y,S\rangle, \langle z,S\rangle\}\}.$$

If the same parameter is classified as scalar in one EMSPS, and nonscalar in another EMSPS, the parameter is classified as nonscalar. This is because computation may exist which requires the parameter to be only in a recipe form. For example, $y$ in the above example is classified as nonscalar in the "**then**" part, and as scalar in the "**else**" part of the conditional expression. The first EMSPS corresponds to the "**then**" part, and the second to the "**else**" part. Thus, the $y$ of $f(x,y,z)$ is classified as nonscalar. As a result, the parameters of $f(x,y,z)$ are classified as

$$\begin{array}{ll} \text{Requisite} : \{x,y\}\,, & \text{Suspended} : \{z\}\,, \\ \text{Scalar} \quad : \{x,z\}\,, & \text{Nonscalar} \; : \{y\}\,. \end{array}$$

The EDPS of recursive functions can be computed similarly to the PDPS computation in another work.[9] The only difference is the reduction domain, which is not PDPS but EDPS. To show an example, consider the following functions which create an infinite sequence of factorial values:

$$seqf(n) = str\_cons(fact(n), seqf(n+1))\,;$$

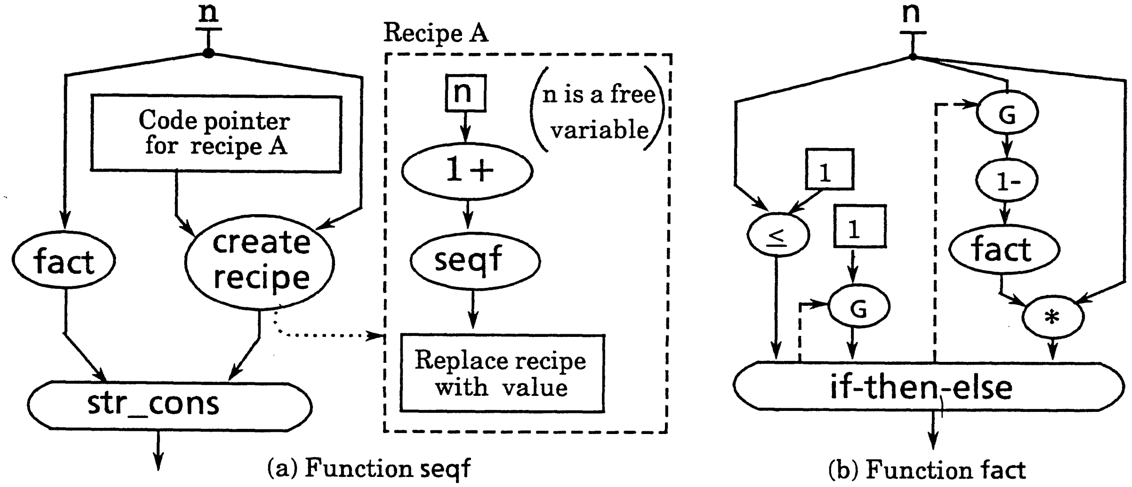
$$fact(n) = \textbf{ if } n \leq 1 \textbf{ then } 1 \textbf{ else } n * fact(n-1) \textbf{ fi.}$$

where *str_cons* is a stream-oriented asymmetrical *cons* operator whose EDPS is $\{\{\langle x,S\rangle, \langle y,N\rangle\}\}$. The EDPSs of these functions are found to be

$$E(seqf(n)) = \{\{\langle n,S\rangle\}\} \text{ and } E(fact(n)) = \{\{\langle n,S\rangle\}\}.$$

### 5.3 Recipe Creation Elimination based on EDPSs

As described in Subsection 3.2, scalar parameters are passed on in an evaluated form. For example, a graph for functions *seqf* and *fact* described in Subsection 5.2, is shown in Fig. 5. The parameter $n$ of function *fact* is requisite and scalar. Therefore, in recipe $A$ in Fig. 5, the expression $n+1$ is evaluated before it is passed on to the function body. In the figure, the "create recipe" node is a macro-node which accepts the pointer of the machine code for an expression as well as the environment for the expression. It then creates the recipe, and returns its pointer. The "replace recipe with value" node is also a macro-node which replaces the recipe with the evaluated value. After replacement, the value is immediately available for any expression that refers to the recipe without repetitive evaluation. An example of dataflow machine implementation of these macro-nodes as well as *lazy_cons* / *str_cons* functions is described elsewhere.[2]



(a) Function seqf

(b) Function fact

Fig. 5 Program Example of Infinite Sequence

### 5.4 Autonomousness of the EDPS

When only the function application method is concerned, the EDPS can be said to be redundant for scalar / nonscalar distinction. For example, for function $f(x, y, z)$ in Subsection 5.2, the EDPS

$$\{\{\langle x, S\rangle, \langle y, N\rangle\}, \{\langle x, S\rangle, \langle y, S\rangle, \langle z, S\rangle\}\}$$

can be reduced to $\{\langle x, S\rangle, \langle y, N\rangle, \langle z, S\rangle\}$.

However, such a reduced property is problematic in that it is not autonomous. As an example, consider the functions $f_1(x, y, z)$ and $f_2(x, y, z)$, whose reduced properties are both $\{\langle x, S\rangle, \langle y, N\rangle, \langle z, S\rangle\}$ :

$$f_1(x, y, z) = \textbf{ if } x > 0 \textbf{ then } x + y \textbf{ else}$$
$$\textbf{if } z > 0 \textbf{ then } cons(x, z) \textbf{ else } cons(y, z) \textbf{ fi fi};$$
$$f_2(x, y, z) = \textbf{ if } x > 0 \textbf{ then } cons(x, y) \textbf{ else}$$
$$\textbf{if } z > 0 \textbf{ then } x + z \textbf{ else } y + z \textbf{ fi fi}.$$

For the function applications $f_1(a, b, b)$, actual parameter $b$ is scalar, since $b$ is used as scalar in both the "**then**" part and the "**else**" part of the outermost conditional expression. However, $b$ is nonscalar in $f_2(a, b, b)$. Such a distinction cannot be drawn when the reduced property is used. In contrast, when EDPS is used, these cases are correctly distinguished. The EDPSs of $f_1$ and $f_2$ are

$$E(f_1(x, y, z) = \{\{\langle x, S\rangle, \langle y, S\rangle\}, \{\langle x, S\rangle, \langle y, N\rangle, \langle z, S\rangle\}\} ;$$
$$E(f_2(x, y, z) = \{\{\langle x, S\rangle, \langle y, N\rangle\}, \{\langle x, S\rangle, \langle y, S\rangle, \langle z, S\rangle\}\} .$$

Therefore, the EDPS of $f_1(a, b, b)$ becomes $\{\{\langle a, S\rangle, \langle b, S\rangle\}\}$, indicating that $b$ is scalar. In contrast, the EDPS of $f_2(a, b, b)$ becomes $\{\{\langle a, S\rangle, \langle b, N\rangle\}, \{\langle a, S\rangle, \langle b, S\rangle\}\}$, confirming that $b$ is nonscalar.

The inter-function level properties used in this paper, namely requisiteness and recipe lifetime, can be described using only the EDPS. Since the EDPS is autonomous for both requisiteness and recipe lifetime properties, the computation required for inter-function optimization can be significantly reduced.

### 6. Conclusion

This paper proposed a new program transformation algorithm for functional programs, named the mixed-mode fixpoint evaluation method. This method realizes optimized demand-driven computation on a dataflow machine. The evaluation scheme takes advantage of both data-driven and demand-driven evaluations in that it offers sufficient parallelism and safeness for nonstrict sequential functions. This can be achieved by eliminating demand propagation predictable during compilation time, and by replacing lengthy recipe creation and evaluation incurred in demand-driven evaluation with the call-by-value method.

The optimization algorithm makes use of two kinds of properties, namely, requisiteness and recipe lifetime. These properties are described using an Extended Dependency Property Set (EDPS). Since the EDPS is autonomous, the computation required for the optimization can be significantly reduced.

The algorithm proposed in this paper assumes a lazy synchronization mechanism, such as that used in a Structure Memory for storing structured data. In combination with this kind of intelligent operation memories, the proposed algorithm realizes the fixpoint computation rule for the class of sequential functions on nonstrict data structures.

This paper intentionally leaves the machine architecture abstract. Further research will be carried out on the construction of an effective and high-speed dataflow machine that can execute all operations described in this paper.

### Acknowledgements

The authors would like to thank Yoshihito Toyama, a research engineer in the First Section of the Information

Science Department for his helpful comments on drafts of this manuscript. They also wish to thank the members of the Dataflow System Research Group in the Second Section for their thoughtful discussions and comments.

## References

1. Amamiya,M., R.Hasegawa, O.Nakamura and H.Mikami, "A List-Processing-Oriented Data Flow Machine Architecture," *Proc. National Computer Conference*, (1982), 143-151.
2. Amamiya,M. and R.Hasegawa, "Dataflow Computing and Eager and Lazy Evaluations," *New Generation Computing,* **2**, (1984), 105-129.
3. Amamiya,M., R.Hasegawa and S.Ono, "Valid, A high-Level Functional Programming Language for Data Flow Machines," *Review of ECL,* **32**, 5, NTT, (1984), 793-802.
4. Clack,C. and S.L.P.Jones, "Strictness analysis - a practical approach," in J.P.Jouannaud (ed.), *Lecture Notes in Computer Science,* **201** , *Functional Programming Languages and Computer Architecture*, Springer-Verlag, (1985), 35-49.
5. Hasegawa,R., H.Mikami and M.Amamiya, "An Architecture for List-Processing-Oriented Data Flow Machine," *Review of ECL,* **32**, 5, NTT, (1984), 771-782.
6. Henderson,P., "Functional Programming / Application and Implementation," Prentice-Hall, (1980).
7. Jagannathan.R. and E.A.Ashcroft, "Eazyflow: A Hybrid Model for Parallel Processing," *Proc. International Conference on Parallel Processing*, IEEE, (1984), 514-523.
8. Manna,Z. "Mathematical Theory of Computation," McGRAW-HILL, (1974).
9. Ono,S., N.Takahashi and M.Amamiya, "Non-strict Partial Computation with a Dataflow Machine," *Proc. 6th RIMS Symposium on Mathematical Methods in Software Science and Engineering*, TR. **547**, RIMS Kyoto Univ., (1985), 196-229.
10. Ono,S., N.Takahashi, "Algorithm for Computing Dependency Property Sets in Recursive Function Systems," *Trans. IECE Japan,* **J69-D**, 5, (1986), 714-723, (in Japanese).
11. Ono,S., and N.Takahashi, "Optimized Demand-driven Evaluation of Functional Programs on a Dataflow Machine," *Proc. Dataflow Workshop 1986*, IECE Japan, (1986), 39-48. (in Japanese)
12. Pingali,K. and Arvind, "Efficient Demand-Driven Evaluation. Part 1," *ACM Trans. Programming Languages and Systems,* **7**, 2, (1985), 311-333.
13. Seki,H., K.Inoue, K.Taniguchi and T,Kasami, "Optimization of Functional Language ASL/F Programs," *Trans. IECE Japan,* **J67-D**, 10, (1984), 1115-1122, (in Japanese).
14. Tanaka,J., "Optimized Concurrent Execution of an Applicative Language," Ph.D thesis, Univ. of Utah, (1984).
15. Treleaven,P.C., D.R.Brownbridge, and R.P.Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys,* **14**, 1, (1982), 93-142.
16. Vuillemin,J., "Correct and Optimal Implementations of Recursion in a Simple Programming Language," *JCSS,* **9**, (1974), 332-354.
17. Yamaguchi,Y., K.Toda and T.Yuba, "A Performance Evaluation of A Lisp-Based Data-Driven Machine (EM-3)," *Proc. 10th Int. Symp. Computer Architecture*, (1983), 363-369

# Optimizing Matrix Operations on a Parallel Multiprocessor with a Hierarchical Memory System

William Jalby[1] and Ulrike Meier[2]
Center for Supercomputing Research and Development
University of Illinois, Urbana, IL 61801

ABSTRACT - Memory organizations of supercomputers (CRAY 2, CEDAR) tend to become more and more complex, and correspondingly data management in these memories becomes a crucial factor for achieving high performance. We study here an architecture combining vector and parallel capabilities on a two-level shared memory structure. For this class of architecture, we analyze and optimize matrix multiplication algorithms so as to obtain high efficiency kernels which can be used for many numerical algorithms such as LU and Cholesky factorizations, as well as Gram-Schmidt and Householder orthogonal factorization schemes. The performance of such kernels on the Alliant FX/8 multiprocessor is described in detail.

## 1. INTRODUCTION

One of the main issues in supercomputer architecture is the design of a memory system that is able to provide the arithmetic operators with operands at a sufficient rate. Several solutions have been proposed; first fully parallel memories (implemented either in a global way: BSP or in a distributed one: ILLIAC IV or more recently Hypercube machines [15]), followed by highly interleaved memories (CRAY, CDC CYBER 205, FUJITSU) and now complex hierarchical memory systems which consist of two levels - a first level of small fast memory and a second level of slower larger memory (which is, in fact, generally interleaved itself). Implementations of these multilevel memory organizations for parallel processors have been studied [5][17] and are currently being used in several supercomputers; the ST-100 provides a fully programmable cache, the CONVEX C1 and ALLIANT FX/8 a hardware managed cache, and the CRAY 2 a programmable local memory for each processor.

While the use of global parallel or interleaved memories only slightly affects the design of numerical algorithms (the main constraint is to avoid accessing vectors by using increments of a power of two , in order to avoid memory bank conflicts [11]), the use of distributed or hierarchical memories requires a careful design of algorithms; more precisely, data management becomes a key factor in realizing high performance. Significant progress has been achieved in developing efficient algorithms for the distributed memory case and in understanding the impact of the communication medium on the algorithm design [7][13][14]. On the other hand, much work remains to be done in the case of a hierarchical memory system especially when it is combined with both vector and parallel capabilities (e.g ALLIANT FX/8). For those architectures, algorithms must not only be designed as to be suitable for vector and parallel processing but also to provide good data locality. These requirements may be contradictory. For example, increasing the vector length may destroy data locality and so give a poor performance. In this paper, our goal will be to study the tradeoffs involved in designing high-performance algorithms for such architectures.

For reducing the complexity of the problem, the decomposition of algorithms into high level modules (e.g. vector/vector, matrix/vector and matrix/matrix operations) seems to be a promising approach.This technique has been successfully used on vector machines, e.g. BLAS, [3] and on MIMD machines (extended BLAS). In our case, the use of BLAS (vector/vector operations) or even extended BLAS (matrix/vector operations) may not be efficient since they mainly contain primitives involving an amount of data of the same order as the number of floating point operations; for example DAXPY (one of the BLAS) operating on vectors of length $N$ will manipulate $3N+1$ data elements for executing only $2N$ operations. This may often result in an inefficient use of the cache, since less than one floating point operation per data accessed has to be performed. However, multiplying two square matrices of order $N$, involves $3N^2$ data elements for $(2N-1)N^2$ operations, so data elements fetched from the memory can be used several times before they are stored back again. We study this primitive in depth since many algorithms may be expressed in terms of matrix / matrix operations as e.g. the Modified Gram-Schmidt algorithm [10] or the Householder reduction [2][8]. The use of the here described methods instead of the conventional BLAS within these two algorithms improved their performance on the ALLIANT FX/8 by a factor of two to three.

In the first section, we describe our target architecture. In the following sections we develop and analyze algorithms for multiplying large matrices. Finally, we present some computational results on an Alliant FX/8 that support our theoretical results in the preceding sections.

## 2. THE TARGET ARCHITECTURE

In this section, we describe briefly the main features of the architecture we plan to study. Our target machine consists of $p$ vector processors (CE = computational element) sharing a memory which is organized in two levels: first a fast small one (cache), second a slower bigger one (main memory). The processors are connected to the cache through a network allowing each of them to read or write any of the cache locations. The set of processors is provided with a synchronization mechanism enabling it to distribute the computations of a single program unit among the $p$ CE's.

A good example of the architecture described above is the ALLIANT FX/8. This machine consists of p=8 pipelined CE's, each of them having a register-oriented architecture (vector register length: 32) elements. The CE's share the

[1] on leave from INRIA, domaine de Voluceau - Rocquencourt, B.P.105 78150 Le Chesnay, France

[2] on leave from Zentralinstitut fuer Angewandte Mathematik, Kernforschungsanlage Juelich GmbH, Postfach 1913, D-5170 Juelich, West-Germany

This work was supported in part by the **National Science Foundation** under Grants No. US NSF DCR84-10110 and US NSF DCR85-09970, the **US Department of Energy** under Grant No. US DOE DE-FG02-85ER25001, and the IBM Donation.

0190-3918/86/0000/0429 $01.00 © 1986 IEEE

physical memory as well as a 16K-word cache. The bandwidth between main memory and CE's is half of that between the cache and the CE's. However, due to managing cache misses, accessing a vector from main memory may require a time larger than twice the time required for accessing the same vector if it is present in cache.

## 3. DESCRIPTION OF THE MATRIX MULTIPLICATION ALGORITHM

For the sake of simplicity, we consider here only the matrix operation: $C = C + A * B$. The similar cases $C = A * B$ and $D = C + A * B$ can be easily derived.

In order to evaluate the matrix multiplication algorithms it is necessary to model the conditions under which we assume they will be executed. We suppose that the replacement policy of the cache blocks is optimal (similar to Belady's MIN algorithm [1]); this assumption yields a standard of comparison on which the design of several algorithms can easily be based.

Previous studies [12][16] have shown that data locality can be improved efficiently by partitioning matrices into square submatrices. This method was primarily used for reducing the number of page faults on a sequential computer using a virtual memory system (first level: transistor memory, second level: disk memory). Our case presents two major differences: first the two different kinds of parallelism available in our machine and second the management of data motion between the two memory levels. In a virtual memory system, the mapping of data arrays into pages is crucial and leads to use square submatrices each of which can be contained in one page. We do not restrict ourselves to square submatrices and we will even see later that the best performance is obtained by using "very" rectangular blocks.

Let the $n_1 X n_2$-matrix $A$ be partitioned into $m_1 X m_2$-blocks $A_{ij}$, the $m_2 X m_3$-matrix $B$ into $n_2 X n_3$-blocks $B_{ij}$ and the $n_1 X n_3$-matrix $C$ into $m_1 X m_3$-blocks $C_{ij}$.

The matrix multiplication is performed as follows:

```
do $i = 1, k_1$
  do $j = 1, k_3$
    do $k = 1, k_2$
      $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
end do
```

where $n_1 = k_1 m_1$, $n_2 = k_2 m_2$ and $n_3 = k_3 m_3$ with $k_1$, $k_2$ and $k_3$ being integers greater than 1.

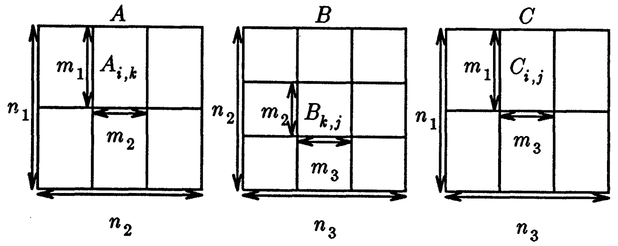


Fig. 1. Partitioning of $A$, $B$ and $C$

The block operations $C_{i,j} = C_{i,j} + A_{i,k} * B_{k,j}$ contain a reasonable amount of potential parallelism, so our algorithms proceed by first partitioning the matrices and then executing the block operations one after another; parallelism and vectorization being inside the multiplication of two submatrices.

Several points with respect to the above algorithm need to be addressed. These are:

(1) the order in which the submatrix computations are performed,

(2) the algorithm used for multiplying two submatrices of $A$ and $B$, and

(3) the sizes of the submatrices.

Concerning the first point, the three loops may be interchanged leading to a different sweep of the matrices. We will only consider the defined case where the inner loop is the k-loop. The cases where the i-loop or the j-loop are the inner loops can be derived accordingly.

The second and the third points are strongly related. The choice of submatrix sizes is obviously vital since potential parallelism is a direct function of these sizes. On the other hand, a bad design of the algorithm (second point) may prevent using the submatrix sizes which provide a good data locality.

The total time required to perform the matrix multiplication can be expressed as:

$$T = lT_l + n_S T_S \tag{3.1}$$

where $l$ denotes the total number of loads executed directly from the memory, $T_l$ the time of one load, $n_S$ the number of submatrix multiplications that have to be performed, and $T_S$ the time for one submatrix multiplication with the three submatrices kept in the cache. The first term represents the total time spent in fetching data from the memory (pure transfer time) while the second represents mainly the time spent in computations (pure computation). Each of them is a complex function of the three parameters $m_1$, $m_2$ and $m_3$, and trying to minimize their sum is difficult. For overcoming this difficulty, we decouple the problem in two independent subproblems: minimization of the transfer time and minimization of the computation time. For each, we determine a region in the parameter space where the value of the cost function is close to the minimum. By choosing a set of parameters within the intersection of the two regions, near-optimal performance can be achieved in most cases.

## 4. COMPUTATION TIME OPTIMIZATION

Let us describe the algorithm for multiplying $A_{ik}$ and $B_{kj}$:

```
do $r = 1, m_3$
  do $s = 1, m_1$
    do $t = 1, m_2$
      $c_{s,r} = c_{s,r} + a_{s,t} b_{t,r}$
end do
```

where $c_{sr}$, $a_{st}$ and $b_{tr}$ denote the elements of $C_{ij}$, $A_{ik}$ respectively $B_{kj}$.

We introduce parallelism by performing the r-loop concurrently on all $p$ processors using vectorization for the s-loop in each of them. Each processor computes an $m_2$-adic operation, the product of the submatrix $A_{ik}$ with a column of $B_{kj}$.

Let us analyze the effect of varying the three parameters on the performance of this kernel, in order to determine general guidelines for the best choice of $m_1$, $m_2$ and $m_3$:

(1) Since the operations on the columns of the submatrices of $A$ are vector operations of length $m_1$, $m_1$ should be chosen as large as possible. For CRAY-like computers however it is preferable to choose $m_1$ as a multiple of the length of a vector register.

(2) Since each processor computes a linear combination of $m_2$ columns of $A_{ik}$, we observe here the effect of the $m_2$-adic operation. The performance is an increasing function of $m_2$, and as shown in [3][4] performance close to the peak is obtained for small values (less than 50) of $m_2$; for example, an optimal value of 16 has been observed for CRAY, and 32 for the ALLIANT FX/8.

(3) Since we operate simultaneously on the columns of the submatrices of $B$, the peak performance is reached when $m_3$ is a multiple of the number of processors, i.e. the computational load is perfecly distributed among the processors.

Another possibility would be to introduce concurrency as well as vectorization for one loop, e.g. the s-loop. The disadvantage of this method is that a large value of $m_1$ may be required to achieve good performance in the submatrix operations. This tends to conflict with the data locality requirements.

We will not consider the problem of how to obtain a minimal $T_S$ any further here as it is too computer dependent. A practical approach for the ALLIANT FX/8, however, is given in [10], a more general approach might be to use a more precise parametrization of vector and parallel processors as in [9], however it will only modify slightly our main results.

## 5. MINIMIZATION OF THE TRANSFER TIME

In this section, we determine the optimal values for submatrix sizes in order to minimize the number of data loads from main memory. For doing this, it is necessary to express the number of data loads as a function of the submatrix sizes. We will consider first the case of large $n_1$, $n_2$ and $n_3$, since for small $n_1$, $n_2$ and $n_3$ all three matrices can be kept in the cache.

In order to determine the total number of loads into the cache by using the given partitioning scheme we assume the following :

(1) Multiplication of the single blocks $A_{ik}$ and $B_{kj}$ is performed according to the method described in section 3.

(2) Each block $C_{ij}$ is loaded into the cache only once and kept there for the duration of the k-loop.

(3) The blocks $A_{ik}$ and $B_{kj}$ have to be loaded each time they are involved in a submatrix multiplication.

Assumption (2) results from the order of the loops that we chose. $C_{ij}$ is independent of the inner loop. If we reorder the loops and choose e.g. the j-loop as the inner loop, $A_{ik}$ could be kept in the cache.

Now the number of loads for computing each submatrix $C_{ij}$ is $m_1n_2+m_3n_2+m_1m_3$ and therefore the total number of loads for computing $C$ is

$$l = \frac{m_1+m_3}{m_1m_3}\, n_1n_2n_3+n_1n_3. \tag{5.1}$$

Note that $l$ is of the same order of magnitude as the number of floating point operations for this matrix multiplication: $fl=2n_1n_2n_3$. This is not the case if the three matrices $A$, $B$ and $C$ are small enough to be kept in the cache during the computation of $C$, where the number of loads equals the total number of data elements: $n_1n_2+n_2n_3+n_1n_3$.

Since $n_1$, $n_2$ and $n_3$ are given, the problem of minimizing $l$ is reduced to minimizing

$$\rho = \frac{m_1+m_3}{m_1m_3} = \frac{1}{m_1} + \frac{1}{m_3} \tag{5.2}$$

under the constraint

$$m_1m_2+m_1m_3 < CS \tag{5.3}$$

where $CS$ denotes the cache size. Inequality (5.3) is justified by regarding hypotheses (2) and (3), and by assuming that the submatrix multiplication is performed so that $A_{ik}$ is swept columnwise. In this case only $p$ row elements of $B_{kj}$ are used at the same time and are not needed afterwards hence, they are neglected in (5.3). If $m_3=p$, $A_{ik}$ can also be neglected, but since a small $m_3$ will lead to a bad data locality, as can easily be verified, we concentrate here on the case $m_3>p$.

The solution of (5.2) that can be obtained by assuming equality in (5.3) gives us the following estimate of the optimal parameters as a function of $m_2$ and the cache size $CS$:

$$m_1 \approx \frac{CS}{m_2+\sqrt{CS}} \quad ; \quad m_3 \approx \sqrt{CS}. \tag{5.4}$$

The optimal $\rho$ is then given by

$$\rho_{opt} \approx \frac{m_2}{CS} + \frac{2}{\sqrt{CS}}. \tag{5.5}$$

(5.4) requires $m_1$ and $m_3$ to be large which fits well the constraints for the computation time. (5.5) indicates that the effect of $m_2$ is not dominant, allowing us to choose $m_2$ large enough to get an efficient $m_2$-adic operation.

Note also that enlarging the cache by a factor of 4 leads to reducing $\rho_{opt}$ only by a factor of 2.

Let us consider the case where one (or two) of the three parameters $n_1$, $n_2$ and $n_3$ is (are) small. In these cases at least one of the matrices to be multiplied is either tall and narrow or short and wide, i.e. "very" rectangular. The idea here is to choose the small dimensions as submatrix sizes: i.e. when $n_i$ is small, set $m_i=n_i$, $i=1,2$ or 3. The optimal parameters $m_1$, $m_2$, $m_3$ and $l$ for these cases can be derived according to the standard case. It can easy be verified that in these cases $l$ is close to the lower bound achievable (i.e. each matrix has to be loaded yat least once). For the cases where one parameter is small, the optimal parameters and the corresponding $l$ are given in Table 1.

|  | $m_1$ | $m_2$ | $m_3$ | $l$ |
|---|---|---|---|---|
| $n_1$ small | $n_1$ | $m_2$ | $\frac{CS}{n_1}-m_2$ | $\left(1+\frac{n_1^2}{CS-n_1m_2}\right)n_2n_3+n_1n_3$ |
| $n_2$ small | $\frac{CS}{n_2+p}$ | $n_2$ | $m_3$ | $\left(1+\frac{n_2^2+n_2p}{CS}\right)n_1n_3+n_2n_1$ |
| $n_3$ small | $\frac{CS}{m_2+n_3}$ | $m_2$ | $n_3$ | $\left(1+\frac{n_3^2+m_2n_3}{CS}\right)n_1n_2+n_1n_3$ |

Table 1. Parameters for rectangular matrices (one matrix size small)

## 6. COMPUTATIONAL RESULTS

In this section, we apply the theoretical results obtained above to the ALLIANT FX/8. We deal only with the case where $n_1$, $n_2$ and $n_3$ are large.

Let us determine here our optimal partitioning. A good choice seems to be $m_1=96$, $m_2=32$ and $m_3=128$. Note that $m_1$ is a multiple of the length of a vector register, $m_2$ the optimal value for the $m_2$-adic operation and $m_3$ a multiple of the number of CE's. Fig.7 shows the experimental results for our optimal partitioning and for two other partitionings achieving the same kernel performance as the optimal one but involving a larger number of loads. We can observe, as predicted by the theory a good correspondence between the decrease of the value $\rho$ ($\rho$=0.15625, for $m_1$=32 and $m_3$=8; $\rho$=0.03125, for $m_1=m_3$=32; and $\rho$=0.01823, for $m_1$=96 and $m_3$=128) and the improvement in performance. Also note that our optimal partitioning achieves a constant high performance for a large range of matrix sizes.

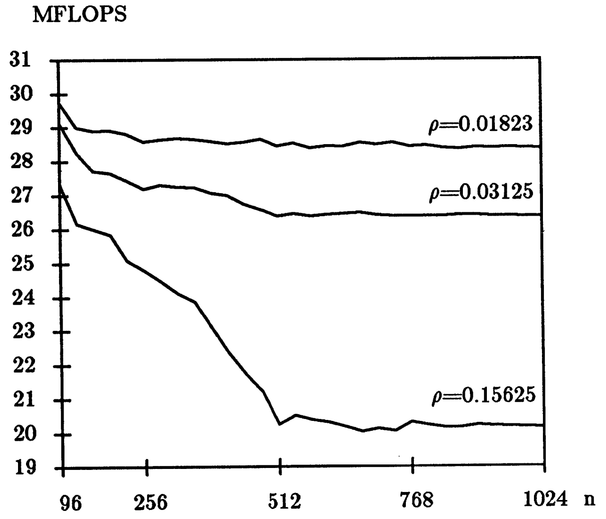


Fig. 2. Performance of the multiplication of two square matrices of order $n$

ACKNOWLEDGEMENTS - We are very grateful to Kyle Gallivan, Dennis Gannon and Ahmed Sameh who took the time to make detailed and constructive comments on the first draft, resulting in a much better exposition.

## REFERENCES

[1] L. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," IBM Syst. J. 5, pp. 78-101, 1966

[2] C. Bischof, C. Van Loan, "The WY Representation for Products of Householder Matrices," TR 85-681, DCS, Cornell University, 1985.

[3] J. Dongarra, J. Bunch, C. Moler, G. W. Stewart, LINPACK Users' Guide, SIAM, 1979.

[4] J. Dongarra, S.C.Eisenstat, "Squeezing the Most out of an Algorithm in CRAY Fortran," ACM ToMS 10, pp. 221-230, 1984

[5] M. Dubois, "A Cache-Based Multiprocessor with High Efficiency," Proc. ICPP, pp. 646-648, Aug. 1985.

[6] D. Gajski, D. Kuck, D. Lawrie, A. Sameh, "CEDAR A Large Scale Multiprocessor," Proc. ICPP, pp. 524-529, Aug. 1985.

[7] D.B. Gannon, J. Van Rosendale, "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms," IEEE ToC C-33, pp. 1180-1195, Aug. 1985.

[8] W. Harrod, "Parallel Algorithms for Linear Least Square problems," CSRD report, Univ. of Ill., 1986.

[9] R.W. Hockney, C.R.Jesshope "Parallel Computers," Adam Hilger, 1981.

[10]W. Jalby, U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Two-Level Memory Hierarchy," CSRD-Report 555, Univ. of Ill., Feb. 1986.

[11]P.M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, 1981.

[12]A. McKellar, E. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," Comm. ACM 12, pp. 153-165, 1969.

[13]Y.Saad, M.H. Schultz, "Parallel Methods for Solving Banded Linear Systems," RR YALEU/DCS/RR-387, Dec. 1985.

[14]A. Sameh, "Solving the Linear Least Squares Problem on a Linear Array of Processors," Purdue Workshop on Algorithmic. Special. Computer Organizations, Sept. 1982.

[15]C.L. Seitz "The Cosmic Cube," Comm. ACM 28, pp. 22-33, 1985.

[16]K. Trivedi, "Prepaging and Applications to Structured Array Problems," Report No. UIUCDCS-R-74-662, DCS, Univ. of Ill. at Urbana-Champaign, 1974.

[17]P.C.C. Yeh, J.H. Patel, and E.D. Davidson, "Performance of Shared Cache for Parallel-Pipelined Computer Systems," Proc. Int. Symp. Comp. Arch., pp. 117-123, June 1983.

# Multiprocessor Jacobi Algorithms for Dense Symmetric Eigenvalue and Singular Value Decompositions

Michael Berry and Ahmed Sameh
Center for Supercomputing Research and Development
University of Illinois, Urbana, IL 61801

**Abstract** – – We present two parallel algorithms based on Jacobi's method for real symmetric matrices to determine the complete eigensystem of a dense real symmetric matrix and the singular value decomposition of rectangular matrices on a multiprocessor. Our intent is to study the advantages of using Jacobi and Jacobi-like schemes over new and existing EISPACK and LINPACK routines on an Alliant FX/8 computer system. For the dense symmetric eigenvalue problem, we show promising results for small-order matrices. A "one-sided" Jacobi-like algorithm which produces the singular value decomposition of a rectangular matrix is shown to provide superior performance for rectangular matrices in which the number of rows is much larger than the number of columns.

## 1. Introduction

Consider the standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{1.1}$$

where $\mathbf{A}$ is a real $n \times n$ dense symmetric matrix. One of the best known methods for determining all the eigenvalues and eigenvectors of (1.1) was developed by the nineteenth century mathematician, Jacobi. We recall that Jacobi's sequential method reduces the matrix $\mathbf{A}$ to diagonal form by an infinite sequence of plane rotations

$$\mathbf{A}_{k+1} = \mathbf{U}_k\mathbf{A}_k\mathbf{U}_k^T, \quad k = 1,2,\cdots,$$

where $\mathbf{A}_1 \equiv \mathbf{A}$ and $\mathbf{U}_k = \mathbf{U}_k(i,j,\theta_{ij}^k)$ is an orthogonal plane rotation matrix which deviates from the identity matrix only in the elements

$$u_{ii}^k = u_{jj}^k = c_k = cos(\theta_{ij}^k) \text{ and } u_{ij}^k = -u_{ji}^k = s_k = sin(\theta_{ij}^k).$$

The angle $\theta_{ij}^k$ is determined so that $a_{ij}^{k+1} = a_{ji}^{k+1} = 0$. It can be shown that for $i < j$

$$a_{ij}^{k+1} = a_{ij}^k \cos 2\theta_{ij}^k - ½(a_{ii}^k - a_{jj}^k) \sin 2\theta_{ij}^k,$$

and hence the annihilation of $a_{ij}^{k+1}$ requires

$$\tan 2\theta_{ij}^k = \frac{2a_{ij}^k}{a_{ii}^k - a_{jj}^k}$$

where $\theta_{ij}^k$ is chosen such that $|\theta_{ij}^k| \leq \pi/4$.

For numerical stability, we determine the plane rotation by

$$c_k = \frac{1}{\sqrt{1+t_k^2}}, \text{ and } s_k = c_k t_k, \tag{1.2}$$

where $t_k$ is chosen as the smaller root (in magnitude) of the quadratic equation

$$t_k^2 + 2\alpha_k t_k - 1 = 0, \quad \alpha_k = cot\ 2\theta_{ij}^k.$$

Hence, $t_k$ may be written as

$$t_k = \frac{(sign\ \alpha_k)}{|\alpha_k| + \sqrt{1+\alpha_k^2}} \tag{1.3}$$

With each $\mathbf{A}_{k+1}$ remaining symmetric and differing from $\mathbf{A}_k$ only in rows and columns $i$ and $j$, the modified elements are

$$a_{ii}^{k+1} = a_{ii}^k + t_k a_{ij}^k,$$

$$a_{jj}^{k+1} = a_{jj}^k - t_k a_{ij}^k,$$

and

$$a_{ir}^{k+1} = c_k a_{ir}^k + s_k a_{jr}^k, \tag{1.4}$$

$$a_{jr}^{k+1} = -s_k a_{ir}^k + c_k a_{jr}^k. \tag{1.5}$$

where $r \neq i,j$. If we represent $\mathbf{A}_k$ by

$$\mathbf{A}_k = \mathbf{D}_k + \mathbf{E}_k + \mathbf{E}_k^T \tag{1.6}$$

where $\mathbf{D}_k$ is diagonal and $\mathbf{E}_k$ is strictly upper triangular, and

$$\sigma_k = \|\mathbf{D}_k\|_F,$$

$$\omega_k = \|\mathbf{E}_k\|_F,$$

where $\|\cdot\|_F$ denotes the Frobenius norm, then from

This work was supported in part by the National Science Fundation under Grant Nos. US NSF DCR84-10110 and US NSF DCR85-09970, the U. S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and the IBM Donation.

0190-3918/86/0000/0433 $01.00 © 1986 IEEE

(1.4) and (1.5) it follows that

$$\omega_{k+1}^2 = \omega_k^2 - (a_{ij}^k)^2,$$

and

$$\sigma_{k+1}^2 = \sigma_k^2 + (a_{ij}^k)^2.$$

If at each step $k$ we annihilate an element $a_{ij}^k$ which is at least of average magnitude, i.e.,

$$(a_{ij}^k)^2 \geq \frac{2}{n(n-1)} \omega_k^2,$$

then

$$\omega_{k+1}^2 \leq \left(1 - \frac{2}{n(n-1)}\right) \omega_k^2,$$

and $\mathbf{A}_k$ approaches the diagonal matrix $\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_n)$. Similarly, $(\mathbf{U}_k \cdots \mathbf{U}_2\mathbf{U}_1)^T$ approaches a matrix whose $j$-$th$ column is the eigenvector corresponding to $\lambda_j$ .

Several schemes are possible for selecting the sequence of elements $a_{ij}^k$ to eliminate via the plane rotations $\mathbf{U}_k$. Unfortunately, Jacobi's original scheme, which consisted of sequentially searching for the largest off-diagonal element, is too time consuming for implementation on a multiprocessor. Instead, a simpler scheme in which the off-diagonal elements $(i,j)$ are annihilated in the cyclic fashion $(1,2), (1,3), \cdots, (1,n); (2,3), \cdots, (2,n); \cdots; (n-1,n)$ is certainly more suitable. We refer to each sequence of $n$ rotations as a sweep. Quadratic convergence for this sequential cyclic Jacobi scheme has been well documented, see [13] and [15]. Convergence usually occurs within 6 to 10 sweeps, i.e. from $3n^2$ to $5n^2$ Jacobi rotations.

A parallel version of this cyclic Jacobi algorithm is obtained by the simultaneous annihilation of several off-diagonal elements by a given $\mathbf{U}_k$ rather than only one as is done in the serial version. For example, let $\mathbf{A}$ be of order 8 and consider the orthogonal matrix $\mathbf{U}_k$ as the direct sum of 4 independent plane rotations in Figure 1, where the $c_i$'s and $s_i$'s for $i = 1,2,3,4$ are simultaneously determined.

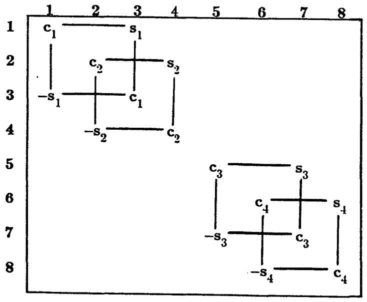


*Figure 1. Sample* $\mathbf{U}_k$ *of Multiprocessor Jacobi Algorithm for* $n = 8$.

Then,

$$\mathbf{U}_k = \mathbf{R}_k(1,3) \oplus \mathbf{R}_k(2,4) \oplus \mathbf{R}_k(5,7) \oplus \mathbf{R}_k(6,8),$$

where $\mathbf{R}_k(i,j)$ is that rotation which annihilates the $(i,j)$ off-diagonal element. If we consider one sweep to be a collection of orthogonal similarity transformations that annihilate the element in each of the $n(n-1)/2$ off-diagonal positions (above the main diagonal) only once, then for a matrix of order 8 the first sweep will consist of 8 successive $\mathbf{U}_k$'s with each one annihilating 4 elements simultaneously. For the remaining sweeps, the structure of each subsequent transformation $\mathbf{U}_k$, $k > 8$ is chosen to be the same as that of $\mathbf{U}_j$ where $j = 1 + (k-1)\ mod\ 8$. In general, the most efficient annihilation scheme consists of $(2r-1)$ similarity transformations per sweep, where $r = \lfloor (n+1)/2 \rfloor$, in which each transformation annihilates different $\lfloor n/2 \rfloor$ off-diagonal elements, see [10]. Although several annihilation schemes are possible, the Multiprocessor Jacobi Algorithm we present in the next section utilizes an annihilation scheme which requires a minimal amount of indexing for computer implementation. Whereas Brent and Luk [2] have demonstrated the implementation of similar Jacobi schemes using systolic arrays on a multiprocessor, the success of using block-matrix Jacobi schemes on a linear array of processors has also been shown (see [14]). However, the algorithms presented in this paper have yet to be implemented with blocking techniques suitable for the Alliant FX/8 computer.

## 2. A Multiprocessor Jacobi Algorithm

The algorithm we present requires $n$ similarity transformations per sweep for a dense real symmetric matrix of order $n$ ($n$ may be even or odd). Each $\mathbf{U}_k$ is the direct sum of either $\lfloor n/2 \rfloor$ or $\lfloor (n-1)/2 \rfloor$ plane rotations, depending on whether $k$ is odd or even, respectively.

**Algorithm 1.**

**Step 1** (Apply orthogonal similarity transformations via $\mathbf{U}_k$ for current sweep)

**1.a** For $k = 1,2,3,...,n-1$ (serial loop)
Simultaneously annihilate elements in positions $(i,j)$, where

$$\begin{cases} i = 1,2,3,..., \lceil (n-k)/2 \rceil \\ j = (n-k+2)-i \end{cases},$$

and for $k > 2$

$$\begin{cases} i = (n-k+2), (n-k+3),..., n - \lfloor k/2 \rfloor \\ j = (2n-k+2)-i \end{cases}$$

**1.b** For $k = n$

Simultaneously annihilate elements in positions $(i,j)$, where

$$\begin{cases} i = 2,3,\ldots,\lceil n/2 \rceil \\ j = (n+2)-i \end{cases}$$

**Step 2** (Convergence test)

**2.a** Compute $\|D_k\|_F$ and $\|E_k\|_F$ as in (1.6).

**2.b** If $\dfrac{\|E_k\|_F}{\|D_k\|_F} < (\textit{tolerance})$, then stop.

Otherwise, go to **Step 1** to begin next sweep.

The annihilation patterns for $n = 8$ is shown in Figure 2, where the integer $k$ denotes an element annihilated via $U_k$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| X | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| | X | 5 | 4 | 3 | 2 | 1 | 8 |
| | | X | 3 | 2 | 1 | 8 | 7 |
| | | | X | 1 | 8 | 7 | 6 |
| | | | | X | 7 | 6 | 5 |
| | | | | | X | 5 | 4 |
| | | | | | | X | 3 |
| | | | | | | | X |

*Figure 2. Annihilation Scheme for Multiprocessor Jacobi Algorithm.*

In the annihilation of a particular $(i,j)$ element in **Step 1** above, we update the off-diagonal elements in rows and columns $i$ and $j$ as specified by (1.4) and (1.5) in Section 1. With regard to storage requirements, it would be advantageous to modify only those rows or column entries above the main diagonal and utilize the guaranteed symmetry of $A_k$. However, to take advantage of the vectorization supported by the Alliant FX/8 computer system, we disregard the symmetry of $A_k$ and operate with full vectors on the entirety of rows and columns $i$ and $j$ in (1.4) and (1.5) i.e., we are using a full matrix scheme. To avoid the necessity of synchronization on the Alliant FX/8, all row changes specified by the $\lfloor n/2 \rfloor$ or $\lfloor (n-1)/2 \rfloor$ plane rotations for a given $U_k$ are performed concurrently with one processor updating a unique pair of rows. After all row changes are completed, we perform the analogous column changes in the same manner. The product of the $U_k$'s, which eventually yields the eigenvectors for $A$, is accumulated in a separate two-dimensional array by applying (1.4) and (1.5) to the $n \times n$ identity matrix.

In **Step 2**, we monitor the convergence of the algorithm by using the ratio of the computed norms to measure the systematic decrease in the relative magnitudes of the off-diagonal elements with respect to the relative magnitudes of the diagonal elements. For double precision accuracy in the eigenvalues and eigenvectors, a *tolerance* of order $10^{-16}$ will suffice for **Step 2.b.** If we assume convergence, this multiprocessor algorithm can be shown to converge quadratically by following the work of Henrici [5], and Wilkinson [15]. In the next section, we discuss the adaptation of a "one-sided" Jacobi algorithm to compute the singular value decomposition on a multiprocessor.

## 3. Singular Value Decomposition

Suppose $A$ is a real $m \times n$ matrix with $m >> n$. The singular value decomposition of $A$ can be defined as

$$A = U\Sigma V^T \qquad (3.1)$$

where $U^TU = V^TV = I_n$ and $\Sigma = diag(\sigma_1, \cdots, \sigma_n)$. The orthogonal matrices $U$ and $V$ define the orthonormalized eigenvectors associated with the $n$ eigenvalues of $AA^T$ and $A^TA$, respectively. The singular values of $A$ are defined as the diagonal elements of $\Sigma$ which are the nonnegative square roots of the $n$ eigenvalues of $AA^T$.

As indicated in [11] for a ring multiprocessor, using a method based on the one-sided iterative orthogonalization method of Hestenes (see also [7] and [9]) is an efficient way to compute (3.1). Luk recommended this scheme for the singular value decomposition on the Illiac IV in [8], and systolic algorithms associated with this scheme have been presented in [2] and [1]. We now consider a few modifications to the scheme discussed in [11] for the determination of (3.1) on the Alliant FX/8.

Our main goal is to determine an orthogonal matrix $\tilde{V}$ as a product of plane rotations so that

$$A\tilde{V} = Q = (q_1, q_2, q_3, \cdots, q_n) \,, \qquad (3.2)$$

and

$$q_i^T q_j = \sigma_i^2 \delta_{ij} \,,$$

where the columns of $Q$, $q_i$, are orthogonal, and $\delta_{ij}$ is the Kronecker delta. We then may write $Q$ as

$$Q = \tilde{U}\Sigma \textit{ with } \tilde{U}^T\tilde{U} = I_n \,,$$

and hence

$$A = \tilde{U}\Sigma\tilde{V}^T \,.$$

Thus, we can obtain (3.1) if we can determine the matrix $\tilde{V}$ in (3.2).

We construct the matrix $\tilde{V}$ via the $(i,j)$ plane rotation

$$(a_i, a_j) \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = (\tilde{a}_i, \tilde{a}_j) \quad i < j \,,$$

so that

$$\tilde{\mathbf{a}}_i^{\mathbf{T}} \tilde{\mathbf{a}}_j = 0, \text{ and } \|\tilde{\mathbf{a}}_i\| > \|\tilde{\mathbf{a}}_j\| \ , \qquad (3.3)$$

where $\mathbf{a}_i$ designates the $i$-*th* column of matrix **A**. This is accomplished by choosing

$$c = \left[\frac{\beta+\gamma}{2\gamma}\right]^{1/2} \text{ and } s = \left[\frac{\alpha}{2\gamma c}\right] \text{ if } \beta > 0 \ , \qquad (3.4)$$

or

$$s = \left[\frac{\gamma-\beta}{2\gamma}\right]^{1/2} \text{ and } c = \left[\frac{\alpha}{2\gamma s}\right] \text{ if } \beta < 0 \ , \qquad (3.5)$$

where $\alpha = 2\,\mathbf{a}_i^{\mathbf{T}}\mathbf{a}_j$, $\beta = \|\mathbf{a}_i\|^2 - \|\mathbf{a}_j\|^2$, and $\gamma = (\alpha^2+\beta^2)^{1/2}$.

Note that (3.3) requires the columns of **Q** to decrease in norm from left to right, and hence the resulting $\sigma_i$ to be in monotonic non-increasing order. To orthogonalize the columns of **A** there are certainly several schemes that can be used to select the order of the $(i,j)$ plane rotations. Following the annihilation pattern of the off-diagonal elements in the sequential Jacobi algorithm mentioned in Section 1, we could certainly orthogonalize the columns in the same cyclic fashion and thus perform the one-sided orthogonalization serially. This process is iterative since orthogonality between columns established in one rotation may be destroyed in subsequent rotations, and convergence is governed by the column norm ordering in (3.3). Each sweep consists of the $n(n-1)/2$ plane rotations selected in cyclic fashion. Due to the similarity with the sequential cyclic Jacobi algorithm for real symmetric matrices and the postmultiplication of matrix **A** only, we refer to this procedure as a "one-sided" Jacobi method.

By implementing the annihilation scheme of the Multiprocessor Jacobi Algorithm presented in Section 2 as the orthogonalization scheme for the one-sided Jacobi method discussed above, we obtain a parallel algorithm for computing the singular value decomposition on a multiprocessor. For example, let $n = 8$ and $m >> n$ so that in each sweep of our One-Sided Multiprocessor Jacobi Algorithm we simultaneously orthogonalize pairs of columns of **A** according to the diagram in Figure 2. In other words, for $n = 8$ we can orthogonalize the pairs (1,8), (2,7), (3,6), (4,5) simultaneously with the postmultiplication of matrix $\tilde{\mathbf{V}}_1$ which consists of the direct sum of 4 plane rotations. In general, each $\tilde{\mathbf{V}}_k$ will have the same form of $\mathbf{U}_k$ in Figure 1 so that at the end of any particular sweep $s_i$ we have

$$\tilde{\mathbf{V}}_{s_i} = \tilde{\mathbf{V}}_1 \tilde{\mathbf{V}}_2 \cdots \tilde{\mathbf{V}}_n \ ,$$

and hence

$$\tilde{\mathbf{V}} = \tilde{\mathbf{V}}_{s_1} \tilde{\mathbf{V}}_{s_2} \cdots \tilde{\mathbf{V}}_{s_t} \ , \qquad (3.6)$$

where $t$ is the number of sweeps required for convergence. We present a formal description of this algorithm in the next section.

## 4. A One-Sided Multiprocessor Jacobi Algorithm

This algorithm is an adaptation of the Multiprocessor Jacobi Algorithm from Section 2 for the singular value decomposition of real $m \times n$ matrices, **A**, where $m >> n$. Each $\tilde{\mathbf{V}}_k$ is the direct sum of either $\lfloor n/2 \rfloor$ or $\lfloor (n-1)/2 \rfloor$ plane rotations, depending on whether $k$ is odd or even, respectively.

### Algorithm 2.

**Step 1** (Postmultiply matrix **A** by orthogonal matrix $\tilde{\mathbf{V}}_k$ for current sweep)

**1.a** Initialize the convergence counter, *istop*, to zero.

**1.b** For $k = 1,2,3,\ldots,n-1$ (serial loop)
Simultaneously orthogonalize the column pairs $(i,j)$, where $i$ and $j$ are given by **1.a** in **Step 1** of **Algorithm 1.**,

provided that for each $(i,j)$ we have

$$\frac{(\mathbf{a}_i^{\mathbf{T}}\mathbf{a}_j)^2}{(\mathbf{a}_i^{\mathbf{T}}\mathbf{a}_i)(\mathbf{a}_j^{\mathbf{T}}\mathbf{a}_j)} > (\mathit{tolerance}) \ . \qquad (4.1)$$

Note: if (4.1) is not satisfied for any particular pair $(i,j)$, then *istop* is incremented by 1 and that rotation is not performed.

**1.c** For $k = n$
Simultaneously orthogonalize the column pairs $(i,j)$, where $i$ and $j$ are given by **1.b** in **Step 1** of **Algorithm 1.**

**Step 2** (Convergence test)

If $\mathit{istop} = n(n-1)/2$, then compute $\sigma_i = \sqrt{(\mathbf{A}^{\mathbf{T}}\mathbf{A})_{ii}}$ , $i = 1,2,\ldots,n$, and stop.

Otherwise, go to beginning of **Step 1** to start next sweep.

In the orthogonalization of columns in **Step 1** we are implementing the plane rotations specified by (3.4) and (3.5), and hence guaranteeing the ordering of column norms and singular values upon termination. Whereas the Jacobi algorithms of Sections 1 and 2 must update rows and columns following each similarity transformation, this one-sided scheme performs only postmultiplication of **A** by each $\tilde{\mathbf{V}}_k$ and hence the plane rotation $(i,j)$ changes only the elements in columns $i$ and $j$ of matrix **A**. The changed elements can be represented by

$$\mathbf{a}_i^{k+1} = c\,\mathbf{a}_i^k + s\,\mathbf{a}_j^k \ , \tag{4.2}$$

$$\mathbf{a}_j^{k+1} = -s\,\mathbf{a}_i^k + c\,\mathbf{a}_j^k \ , \tag{4.3}$$

where $\mathbf{a}_i$ denotes the *i-th* column of matrix **A**, and $c$, $s$ are determined by either (3.4) or (3.5). Since no row accesses are required and no columns are interchanged, one would expect good performance for this method on a machine such as the Alliant FX/8 which can apply vector operations to compute (4.2) and (4.3). As with the Multiprocessor Jacobi Algorithm in Section 2, no synchronization among processors is required for the implementation on multiprocessors. Each processor is assigned one rotation and hence orthogonalizes one pair of the $n$ columns of matrix **A**.

Following the convergence test used in [9], in **Step 2** we count the number of times the quantity

$$\frac{\mathbf{a}_i^T \mathbf{a}_j}{(\mathbf{a}_i^T \mathbf{a}_i)(\mathbf{a}_j^T \mathbf{a}_j)} \tag{4.4}$$

falls, in any sweep, below a given *tolerance*. The algorithm terminates when the counter *istop* reaches $n(n-1)/2$, the total number of column pairs, after any sweep. Upon termination, the matrix **A** has been overwritten by the matrix **Q** from (3.2) and hence the singular values $\sigma_i$ can be obtained via the $n$ square roots of the diagonal entries of $\mathbf{A}^T\mathbf{A}$. The matrix **U** in (3.1), which contains the left singular values of the original matrix **A**, is readily obtained by scaling the resulting matrix **A** (now overwritten by $\mathbf{Q} = \tilde{\mathbf{U}}\Sigma$) by the singular values $\sigma_i$, and the matrix **V**, which contains the right singular vectors of the original matrix **A**, is obtained as in (3.6) as the product of the orthogonal $\tilde{\mathbf{V}}_k$'s. This product is accumulated in a separate two-dimensional array by applying the rotations specified by (4.2) and (4.3) to the $n \times n$ identity matrix. It is important to note that the use of the ratio in (4.4) is preferable over the use of $\mathbf{a}_i^T\mathbf{a}_j$, since this dotproduct can be necessarily small for relatively small singular values. On the Alliant FX/8 computer, the number of sweeps required for convergence using (4.4) ranged between 3 to 8 for a *tolerance* of order $10^{-14}$.

Although we have restricted our discussion of the One-Sided Multiprocessor Jacobi Algorithm to the singular value decomposition, this method is certainly applicable for solving the eigenvalue problem (1.1) for real nonsingular symmetric matrices. If $m = n$, **A** is a positive definite matrix, and **Q** is given by (3.2), it is not difficult to show that

$$\begin{cases} \sigma_i^2 = \lambda_i^2 \\ \mathbf{x}_i = \dfrac{\mathbf{q}_i}{\lambda_i} \end{cases} \quad i = 1,2,\ldots,n \ ,$$

where $\lambda_i$ denotes the *i-th* eigenvalue of **A**, $\mathbf{x}_i$ the corresponding normalized eigenvector, and $\mathbf{q}_i$ the *i-th* column of matrix **Q**.

Two advantages of this one-sided Jacobi scheme over the "two-sided" Jacobi method discussed in Section 2 are that no row accesses are needed and that the matrix $\tilde{\mathbf{V}}$ need not be accumulated. Before discussing the performance of the Jacobi algorithms on the Alliant FX/8 computer, we present an overview of the architectural and software characteristics of the Alliant FX/8 computer in Section 5.

## 5. The Alliant FX/8

Both multiprocessor Jacobi algorithms described thus far have been implemented along with several new and existing EISPACK and LINPACK routines on an Alliant FX/8 computer at the Center for Supercomputing Research and Development, University of Illinois at Champaign-Urbana. On the FX/8 computer, 8 computational elements (CEs) deliver 94.4 MFLOPS (millions of floating point operations per second) peak single precision (32-bit) vector, 46 MFLOPS peak double precision (64-bit) vector, and 35 MIPS (millions of instructions per second) scalar performance. Each CE is a microprogrammed general purpose computer with a 5 stage pipelined instruction processor; a pipelined vector and floating-point unit; eight 64-bit, 32 element vector registers; a 16 KB (kilobyte) instruction cache and concurrency control hardware. Parallel processing is achieved by the application of all processors in the computational complex to the execution of a single program.

The memory of the Alliant FX/8 consists of a physical memory which is expandable to 64 MB (megabyte) in 8 MB modules, and 2 expandable cache systems that support the CEs. Four-way interleaving on each physical memory module permits any module to supply the full memory bus bandwidth of 188 MB-per-second on sequential read access (and 150 MB-per-second on sequential write access). The cache systems use a write-back architecture to reduce memory bus traffic, and the computational processor cache expands to a four-way interleaved, 128 KB physical memory buffer that allows up to eight simultaneous accesses in each 170 nanosecond cycle. A crossbar interconnect links the computational processor cache to the CEs and dynamically connects any CE to any cache port at 376 MB-per-second sustained bandwidth.

## 6. Performance On The Alliant FX/8

In this section we evaluate the performance of our two multiprocessor Jacobi algorithms on the Alliant FX/8. For comparison purposes, we refer to the

Multiprocessor Jacobi Algorithm from Section 2 as MUJAC, and the One-Sided Multiprocessor Jacobi Algorithm from Section 4 as OMJAC. All experiments on the Alliant FX/8 were performed using 64-bit double precision floating-point numbers. For Figures 3 and 4 we solve the dense symmetric eigenvalue problem

$$\mathbf{AX}=\mathbf{X\Lambda} \ , \qquad (6.1)$$

where $\mathbf{A} = [\,a_{ij}\,]$ is $n \times n$ and $a_{ij} = dfloat\,[\,max(i,j)\,]$.

In Figure 3 we plot the number of MFLOPS achieved by the Alliant FX/8 when MUJAC and OMJAC are used to compute eigenvalues and eigenvectors for matrices of order $n$. While MUJAC and OMJAC are implemented completely in FORTRAN, MUJAC(A) and OMJAC(A) use ASSEMBLER subroutines to compute the row and column updates as well as column inner products. The decrease in performance for MUJAC and OMJAC for matrix orders greater than 100 can be greatly attributed to the memory limitations of the computational processor cache described in Section 5, i.e., double precision two-dimensional arrays of order 100 or greater cannot be stored continguously in the 128 KB cache. Although peak performance of 12 and 17 MFLOPS for MUJAC and MUJAC(A) for $n = 50$ were much larger than the 11 and 14 MFLOPS for OMJAC and OMJAC(A), the variation from the peak performance for all $n$ was certainly much smaller for the latter pair. Since the number of multiplications required by MUJAC to determine and perform a rotation (on a full matrix) is twice that of OMJAC, we would expect that OMJAC would require only one-half the computational time that MUJAC does. This was usually the case when both algorithms were used to compute eigenvalues and eigenvectors for dense symmetric matrices. Although MUJAC is generally 50% slower than OMJAC, the potential for vectorization per rotation is much greater for MUJAC and thus we obtain the disparity in MFLOPS between the algorithms for $n \leq 200$.

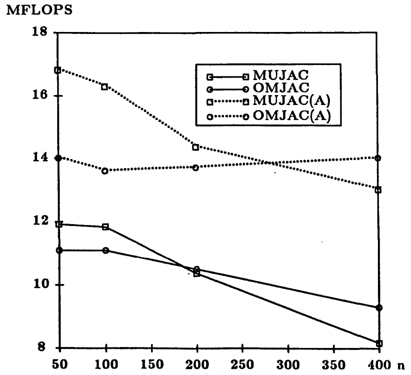


*Figure 3. Performance of Jacobi Schemes on Alliant FX/8.*

For Figure 4, let $t_8$, $t_v$, and $t_g$ represent the execution time on 8 processors with vectorization, 1 processor with vectorization, and 1 processor with only global optimization, respectively, and define

$$\sigma_{vc} = \frac{t_8}{t_g} \text{ and } \sigma_c = \frac{t_8}{t_v} \ ,$$

where $\sigma_{vc}$ represents speed-up for vectorization and concurrency, and $\sigma_c$ represents speed-up for concurrency only. From the graphs in Figure 4, we observe that MUJAC can maintain a larger $\sigma_{vc}$ than OMJAC for $n < 400$, and yet $\sigma_c$ for MUJAC decreases immediately from its peak of 5 at $n = 50$. Vectorization is the major reason for this result. Comparing the values of $\sigma_{vc}$ and $\sigma_c$ for MUJAC and MUJAC(A), we notice that $\sigma_{vc} \approx 2 \times \sigma_c$, and so we can increase the speed-up of MUJAC and MUJAC(A) by a factor of 2 if we use vectorization. However, for $n > 100$ we observe the memory limitation of the computational processor cache by the sharp declines in $\sigma_{vc}$ and $\sigma_c$ for both MUJAC and MUJAC(A). These declines are due not only to the cache memory limitiations incurred by large $n$, but also to the row accessing that must be done by MUJAC. OMJAC and OMJAC(A), on the other hand, do not show such sharp declines in $\sigma_{vc}$ and $\sigma_c$, and so the cache effect is not nearly as great when only column accesses are required. For OMJAC and OMJAC(A) we note that the graphs of $\sigma_{vc}$ and $\sigma_c$ are very similar, and so the vectorization effect is not as substantial as it was for MUJAC. However, where in the case of MUJAC and MUJAC(A) we can expect the speed-ups $\sigma_{vc}$ and $\sigma_c$ to significantly drop for large $n$, OMJAC and OMJAC(A) show little variation from their peak performance for large $n$. In fact, OMJAC(A) is nearly ideal in that it maintains a speed-up close to 8, the number of processors, for all $n$.

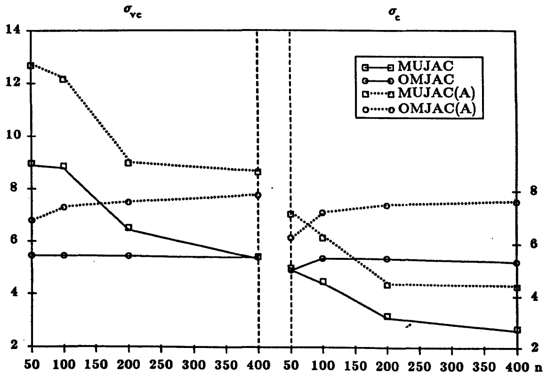


*Figure 4. Speed-ups for concurrency with and without vectorization on Alliant FX/8.*

## 7. Comparisons With EISPACK and LINPACK

We now compare the performance in speed and accuracy of MUJAC and OMJAC with that of new and existing EISPACK and LINPACK routines on the Alliant FX/8. For the dense symmetric eigenvalue problem we first compare MUJAC, OMJAC, and TRED2+TQL2 from EISPACK [12]. In order to compare a set of highly efficient subroutines that are optimized for vector as well as parallel processing, we compare MUJAC(A) and OMJAC(A) which use ASSEMBLER routines for applying rotations and computing dotproducts, with TQL2 and the new matrix-vector implementation of TRED2, TRED2V, developed by Dongarra et al. in [4]. On the Alliant FX/8, MUJAC(A) and OMJAC(A) are 30% faster than MUJAC and OMJAC, and TRED2V is on average 40% faster than TRED2. In Figure 5, we compare the timing of these three algorithms on 8 CEs with full optimization (global and vector) for

$$\mathbf{AX} = \mathbf{X\Lambda} \ ,$$

where $\mathbf{A} = [\, a_{ij} \,]$ is $n \times n$ and $a_{ij} = dfloat\,[\,(i+j-1)/n\,]$.

The largest $n$ for which MUJAC(A) executed faster than TRED2+TQL2 was 90, while OMJAC(A) consistently out-performed the other two algorithms and required only one-half the execution time of the EISPACK routines for each $n$. For $n \geq 100$ we exceed the Alliant FX/8's cache memory capability and consequently obtain sharper increases in time especially for MUJAC(A). Convergence for both MUJAC(A) and OMJAC(A) occurred after 3 to 4 sweeps, and for each $n$ the order of accuracy in the eigenvalues and eigenvectors computed by these Jacobi algorithms was identical to that obtained by TRED2V+TQL2.

For any symmetric matrix of order greater than 100, the advantage of using MUJAC (and OMJAC) rather then TRED2+TQL2 will primarily depend on whether or not convergence can be obtained within 2 to 3 sweeps. This rate of convergence is more prevalent not only in the more diagonally dominant matrices but also in symmetric matrices with a large number of multiple eigenvalues (see [16]).

For the singular value decomposition of a real $m \times n$ matrix $\mathbf{A}$ $(m >> n)$

$$\mathbf{A} = \mathbf{U\Sigma V^T} \ ,$$

where $\mathbf{U^TU} = \mathbf{V^TV} = \mathbf{I_n}$, and $\Sigma = diag(\sigma_1, \cdots, \sigma_N)$, we compare the speed and accuracy of OMJAC and OMJAC(A) with that of the appropriate routines from EISPACK and LINPACK, SVD and DSVDC. Recall that both routines SVD and DSVDC reduce the matrix $\mathbf{A}$ to bidiagonal form via Householder transformations and then diagonalize this reduced form using plane rotations. We will also compare our results with OMJAC and OMJAC(A) with the new matrix-vector implementation of SVD, SVDV, developed by Dongarra et al. in [4], which has been demonstrated to achieve 50% speed-up in execution time over SVD on machines such as the CRAY-1.

Suppose that the elements of the $m \times 32$ matrix $\mathbf{A} = [\, a_{ij} \,]$ are given by

$$a_{ij} = dfloat\,[\,(i+j-1)/n\,]$$

for $m >> 32$, and that we wish to determine all the singular values and singular vectors of the matrix $\mathbf{A}$. In Figure 6, we present the speed-ups for OMJAC(A) over each routine used to compute the singular values and singular vectors of the matrix $\mathbf{A}$ on the Alliant FX/8 with 8 CEs and vectorization for each $m$. With regard to accuracy, OMJAC(A) was somewhat less accurate than SVDV and .DSVDC for the smaller values of $m$, but very competitive for $m > 512$.

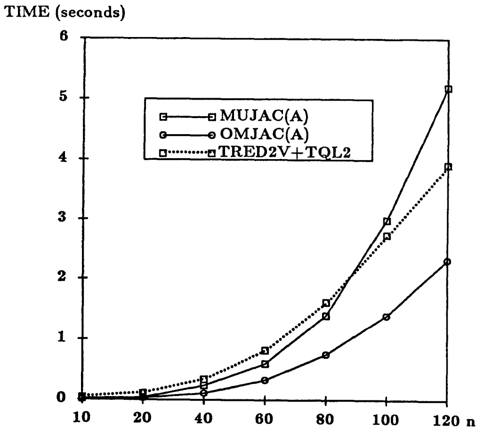


*Figure 5. Execution Times for Highly Optimized Routines.*

| m | $\frac{t_s}{t_o}$ | $\frac{t_d}{t_o}$ | $\frac{t_{sv}}{t_o}$ |
|---|---|---|---|
| 64 | 3.57 | 8.33 | 1.78 |
| 96 | 3.57 | 7.14 | 1.67 |
| 128 | 3.33 | 5.88 | 1.69 |
| 256 | 3.85 | 4.17 | 1.48 |
| 512 | 3.45 | 3.03 | 1.13 |
| 1024 | 3.33 | 2.78 | 1.42 |
| 2048 | 3.23 | 2.38 | 1.32 |
| 4096 | 3.23 | 2.63 | 1.40 |
| 8192 | 3.13 | 2.70 | 1.26 |

$t_d$ ≡ time for DSVDC
$t_o$ ≡ time for OMJAC(A) , [3 sweeps]
$t_s$ ≡ time for SVD
$t_{sv}$ ≡ time for SVDV

*Figure 6. Speed-ups for OMJAC(A) on Alliant FX/8 in Double Precision.*

## 8. Conclusions

For solving the dense symmetric eigenvalue problem on multiprocessors such as the Alliant FX/8, we have presented a parallel algorithm (based on Jacobi's method for symmetric matrices), MUJAC, that can produce accurate eigenvalues and eigenvectors significantly faster than the popular EISPACK routines, TRED2+TQL2, for matrices of order less than 100. We have also presented a Jacobi-like one-sided multiprocessor algorithm, OMJAC, that can be used to solve not only the symmetric eigenvalue problem but also the singular value decomposition of $m \times n$ matrices with $m >> n$. When used to determine eigenvalues and eigenvectors (though somewhat less accurate) on the Alliant FX/8, a highly efficient implementation of OMJAC using FORTRAN and ASSEMBLER, OMJAC(A), executes not only 50% faster than MUJAC but also substantially faster than the most efficient EISPACK routines, TRED2V+TQL2, for all matrix orders considered. When compared with the new and existing EISPACK and LINPACK routines for computing the singular value decomposition, OMJAC(A) is at least 2 to 3 times faster.

Future work with the Jacobi algorithms presented in this paper will involve the use of blocking schemes to optimize MUJAC and OMJAC for better cache management of the Alliant FX/8 as the order of the matrix gets large.

## References

[1] R. P. Brent, F. T. Luk, and C. Van Loan, "Computation of the singular value decomposition using mesh connected processors," *J. VLSI and Computer Systems,* vol. 1, no. 3, pp. 242-270, 1985.

[2] R. P. Brent and F. T. Luk, "The solution of singular value and symmetric eigenproblems on multiprocessor arrays," *SIAM J. Sci. Stat. Comput.,* vol. 6, pp. 69-84, 1985.

[3] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK Users' Guide,* SIAM, Philadelphia, 1979.

[4] J. Dongarra, L. Kaufman, and S. Hammarling, "Squeezing the most out of eigenvalue solvers on high-perfomance computers," Technical Memorandum No. 46, Mathematics and Computer Science Divison, Argonne National Laboratory, 1985.

[5] P. Henrici, "On the speed of convergence of cyclic and quasicyclic Jacobi methods for computing eigenvalues of Hermitian matrices," *J. Soc. Indust. Appl. Math.,* vol. 6, pp. 144-162, 1958.

[6] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," *J. Soc. Indust. Appl. Math.,* vol. 6, pp. 51-90, 1958.

[7] H. F. Kaiser, "The JK method: a procedure for finding the eigenvectors and eigenvalues of a real symmetric matrix," *The Computer Journal,* vol. 15, no. 33, pp. 271-273, 1972.

[8] F. T. Luk, "Computing the singular-value decomposition on the Illiac IV," *ACM Trans. Math. Software,* vol. 6, no. 4, pp. 524-539, 1980.

[9] J. C. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem," *The Computer Journal,* vol. 18, no. 1, pp. 74-76, 1975.

[10] A. Sameh, "On Jacobi and Jacobi-like algorithms for a parallel computer," *Math. Comp.,* vol. 25, pp. 579-590, 1971.

[11] A. Sameh, "Solving the linear least squares problem on a linear array of processors," *Algorithmically Specialized Parallel Computers,* Academic Press, pp. 191-200, 1985.

[12] B. T. Smith, J. M. Boyce, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines - EISPACK Guide, Second Edition,* Springer-Verlag, Berlin, 1976.

[13] A. Schönhage, "Zur Konvergenz des Jacobi-Verfahrens," *Numer. Math.,* vol. 3, pp. 374-380, 1961.

[14] C. Van Loan, "The block Jacobi method for computing the singular value decomposition," Technical Report No. 85-680, Department of Computer Science, Cornell University, Ithaca, New York, 1985.

[15] J. H. Wilkinson, "Note on the quadratic convergence of the cyclic Jacobi process," *Numer. Math.,* vol. 5, pp. 296-300. 1962.

[16] J. H. Wilkinson, *The Algebraic Eigenvalue Problem,* Clarendon Press, Oxford, 1965.

[17] J. H. Wilkinson and C. Reinsch, *Linear Algebra,* vol. 2 of *Handbook for Automatic Computation,* Springer-Verlag, Berlin, 1971.

# SOLVING POSITIVE DEFINITE LINEAR SYSTEMS ON VECTOR COMPUTERS

Hung-chi Kuo

Department of Atmospheric Science
Colorado State University
Fort Collins, Colorado 80523

Swarn P. Kumar

Artificial Intelligence Center
Boeing Computer Services
Seattle, WA 98124

## 1. Introduction

The problem of solving a real, symmetric and positive definite matrix of size n, on vector computers of type Cray-1 and Cyber-205 is considered. Three parallel algorithms based on simple Gaussian elimination (hereafter refered to as GE algorithm), Gaussian elimination with scaled partial pivoting (PGE algorithm) and the Cholesky decompositon (LDLT algorithm) are analyzed and compared in terms of their speedup and efficiency. Actual performance of these three algorithms on Cray-1 and Cyber-205 is also included.

A study based on these algorithms for MIMD (multiple-instruction multiple-data) type machines is presented by Kumar and Kowalik [4]. Details about Cray-1 and Cyber-205 architecture can be found in Hockney and Jessehope [2], Kascic [3] and Gentzsch [1].

## 2. The algorithms

Six algorithms to solve a randomaly generated symmetric positive definite matrix (diagonally dominant) in the vector computers Cray-1 and Cyber-205 are analyzed. These 6 programs are the scalar and vectorized version of the GE, PGE and LDLT algorithms. The algorithms are arranged in a way so that the vectors are contiguous in the memory. Thus, 'identical' codes are compared on the two machines.

The first algorithm is the Gaussian elimination method (GE) without pivoting. The factorization part of the algorithm is as follows:

```
For  k=1,-----,n-1 do;
   For  i=k+1,-----,n do;
      a(i,k):=a(i,k)/a(k,k)
   For  j=k+1,-----,n do;
      For  i=k+1,-----,n do;
        a(i,j):=a(i,j) -a(i,k)*a(i,j)
```

Note that the i loops of the above algorithm are the vectorized loops in both machines because of parallel computations and contiguous memory locations involved. The backward and forward substitution in this code are vectorized according to the "column sweep" method. It is noticed in the results that the factor, whether the substitution is vectorized or not, does not affect the CPU time much. This is because the $O(n^3)$ operation counts in the factorization dominates the $O(n^2)$ operation counts in the substitutions.

The second algorithm is essentially the same as GE algorithm except we introduce the "scaled partial pivoting" strategy. In general, the pivoting should give us a more accurate solutions. Thus, it is useful to investigate the price of a pivoting strategy on the vector machines. In this "scaled partial pivoting" strategy, the bookkeeping is stored in a vector and used later on in the forward substitution. The pivoting is a scalar process, therefore in the PGE program only the factorization and the backward substitution can be vectorized.

The final algorithm is the Cholesky method (LDLT). In this decompositon, $A = LDL^T$ where L is a lower triangular with unit elements and the D is a diagonal matrix with positive elements. To avoid the square root computation is the reason for introducing D matrix. The algorithm is as follows:

```
d(1) = a(1,1)
For  i = 2,-----,n do;
L(i,1) = a(i,1)/d(1)
For  k = 2,-----,n-1 do;
                   k-1
d(k) = a(k,k) -  Σ  d(p)*L(k,p)**2
                   p=1
For  i = k+1,-----,n do;
                      k-1
L(i,k) = (a(i,k) -  Σ  L(i,p)*d(p)*L(k,p))/d(k)
                      p=1
                   k-1
d(n) = a(n,n) -  Σ  L(n,p)**2*d(p)
                   p=1
```

The i loops are again the vectorized loops for both computers.

The operation counts for sequential GE is $n^3/3$ while it is roughly $n^3/6$ for the Cholesky method.

## 3. Computational Results

The speed up of an algorithm is defined here

0190-3918/86/0000/0441 $01.00 © 1986 IEEE

as its the sequential CPU time divided by its vectorized CPU time. The normalized CPU time is defined as the CPU time divided by the order of operation counts (here is $N^3$). All programs were tested on a randomly generated positive definite matrix. Table 1 gives the CPU time (in seconds) for the GE, LDLT and PGE programs on the Cyber-205 and the Cray-1 for different values of N. From Table 1 we can see there is not much difference in the CPU time for the scalar PGE program and the scalar GE program(within 10% difference). The speed up values on the Cray-1 never exceed 10 while the speed up values on the Cyber-205 can be greater than 20 (e.g. the GE algorithm for N = 200).

The speed up factors as functions of the matrix size N for all three algorithms on both vector machines are shown in Figure 1. The GE algorithm has larger speed up compare to the other two algorithms on both computers. This may imply that the GE algorithm has more parallelism in the computations. The GE algorithm in the Cyber-205 especially has the largest speed up values (e.g. maximum speed up of 22.3 for N = 200) except when N is very small. On the Cyber-205, the LDLT has better speed up than the PGE algorithm due to the fact that certain percentage of the code in PGE can not be vectorized. The speed up of the LDLT algorithm is about the same as the speed up of the PGE algorithm in the Cray-1( it ranges from 2 to 5 from small N to large N ). This probably is due to the Cray-1 has better scalar process to handle the pivoting. In general, the Cyber-205 has better speed up than the Cray-1 especially for large N. This means there is a substantial difference in program performance as one change from the scalar environment to the vectorized environment on the Cyber-205. The results are reasonalbe since Cyber-205 favors long vector length for peak performance and its scalar codes do not run as fast as their counterparts on the Cray-1.

Table 1. The CPU time (in seconds) for the vectorized and scalar versions of the GE, PGE and LDLT algorithms on both vector computers. The S columns are the speed up values for the particular algorithm.

CYBER-205

| | PGE | | | LDLT | | | GE | | |
|---|---|---|---|---|---|---|---|---|---|
| N | SCALAR | VECTOR | S | SCALAR | VECTOR | S | SCALAR | VECTOR | S |
| 50 | 0.033 | 0.009 | 3.7 | 0.023 | 0.006 | 3.8 | 0.029 | 0.004 | 6.7 |
| 100 | 0.236 | 0.037 | 6.4 | 0.170 | 0.025 | 7.0 | 0.221 | 0.018 | 12.2 |
| 150 | 0.765 | 0.086 | 8.9 | 0.56 | 0.057 | 9.8 | 0.73 | 0.043 | 16.9 |
| 200 | 1.99 | 0.168 | 11.8 | 1.335 | 0.109 | 12.2 | 1.853 | 0.083 | 22.3 |

CRAY-1

| | PGE | | | LDLT | | | GE | | |
|---|---|---|---|---|---|---|---|---|---|
| N | SCALAR | VECTOR | S | SCALAR | VECTOR | S | SCALAR | VECTOR | S |
| 50 | 0.025 | 0.007 | 3.4 | 0.015 | 0.005 | 3.4 | 0.022 | 0.005 | 4.9 |
| 100 | 0.175 | 0.036 | 4.9 | 0.106 | 0.020 | 5.2 | 0.161 | 0.027 | 6.1 |
| 150 | 0.594 | 0.092 | 6.5 | 0.322 | 0.053 | 6.0 | 0.532 | 0.074 | 7.2 |
| 200 | 1.381 | 0.215 | 6.4 | 0.74 | 0.11 | 6.7 | 1.278 | 0.178 | 7.2 |

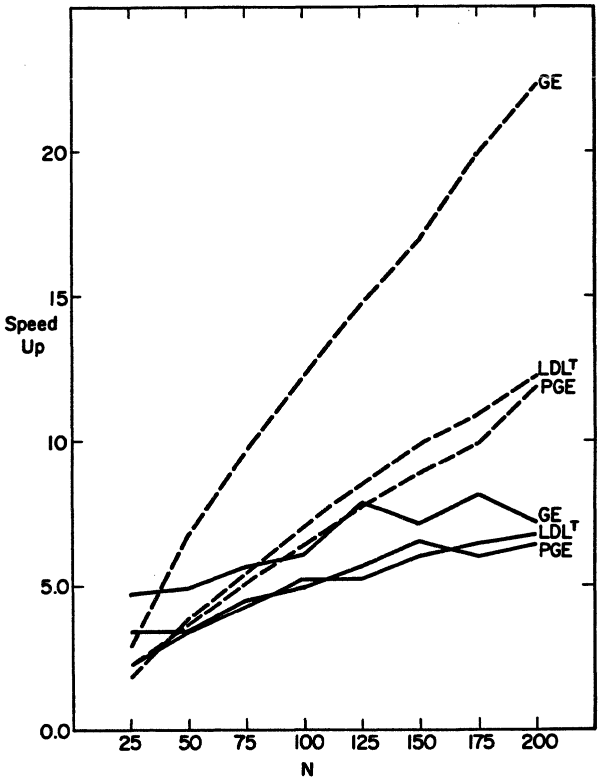


Figure 1. The speed up values as a function of the matrix size N for the GE, PGE and LDLT algorithms on both vector computers. The solid lines are the results from the Cray-1. The dashed lines are the results from the Cyber-205.

Figure 2 illustrtes the normalized CPU time as a function of the matrix size N. The upper four curves are the results of the scalar codes on both computers. The PGE results are so close to the GE results thus not shown here. The scalar LDLT and GE programs run faster on the Cray-1 than on the Cyber-205. The scalar LDLT program runs faster than scalar GE program in both machines. These results are due to better scalar process on the Cray-1 and less operations involed in scalar LDLT program. We also observe from these scalar curves that the Cyber-205 curves reach their asymptotic performance with smaller N compare to the Cray-1 curves. This can be interpreted as the faster reach of Cyber-205's scalar optimal efficiency than the Cray-1.

The bottom six curves the Figure 2 are the results of the normalized CPU time for the vectorized codes on both computers. Non of the testing codes reached its asymptotic performance for the experimental values of N. The vectorized LDLT program runs faster than the vectorized GE program on the Cray-1. On the other hand, the vectorized GE program performs better than vectorized LDLT program on the Cyber-205. Actually the vectorized GE performs best on the Cyber-205 for N greater than 100. The PGE program performs better on Cyber-205 than the PGE program on the Cray-1 only when N is somewhat greater than 100. The bigger size of matrix A provides long length vectors, therefore the performance is increased on the Cyber-205.

In all cases, the computed error norms do not show significance difference from algorithms to algorithms and from machine to machine. If we are to solve a positive definite matrix that is not diagonally dominant, the accuracy then becomes a big concern, and we should use the vectorized PGE program. The pivoting stragtegy is cheap in the scalar environment (shown in Table 1), but is as expensive as the factorization part of the program for larger N in the vectorized environments. The PGE is not recommended to solve a diagonally dominant matrix on the vector machines.

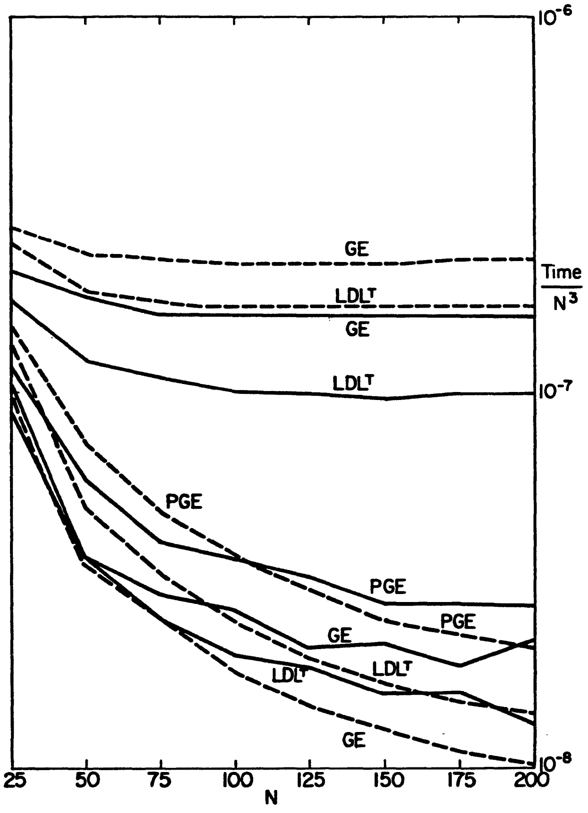


Figure 2. The normalized CPU time (in seconds) as a function of matrix size N on both vector computers. The solid curves are from the Cray-1 while the dashed curves are from the Cyber-205. The upper 4 curves are scalar results while the lower 6 curves are vector results.

## 4. Conclusions

A comparative performance analysis of three different algorithms to solve a positive definite system of linear equations on two vector computers is presented. Our experimental results for vector computers show that the vectorized Gaussian elimination performs best of the three methods on Cyber-205 vector computer, whereas the vectorized Cholesky method is best suited to Cray-1. All the corresponding scalar codes ran faster on Cray-1 than on Cyber-205. On the other hand, the scalar codes reach their optimal asymptotic efficiency with smaller matrix size N on the Cyber-205. In general,the Cyber-205 has better speed up than the Cray-1. The difference in CPU time for scalar PGE and GE programs on both machines were negligible. The price of the pivoting in PGE program can be as expensive as the factorization part of the GE program in the vectorized environment.

Acknowledgements

The work was partially supported by ONR contract # n00014-84-0591, with additional support from the Cooperative Institute for Research in the Atmosphere (CIRA) at CSU. The Cyber-205 computations were performed at Colorado State University and the Cray-1 computations at National Center for Atmospheric Research. The authors wish to thank Drs. Wayne Schubert and Scott Fulton for their useful discussions and comments.

References

[1] Gentzsch, W., Vectorization of Computer Programs with Application to Computational Fluid Dynamics, F. Vieweg and Sons, Braunschweig, 1984.

[2] Hockney, R. W., Parallel Computers, Adam Hilger, Bristol, 1981.

[3] Kascic, M. J., "A Performance Survey of the CYBER-205", High Speed Computation, Springer Verlag, vol.7,1983, pp. 191-210.

[4] Kumar, S. P. and Kowalik, J. S., "Parallel Factorization of a Positive Definite Matrix on an MIMD computer", Proceeding of International Conference on Parallel Processing, Aug. 1984, pp. 410-416.

# Parallel Approximate Algorithms for the 0-1 Knapsack Problem

**P.S. Gopalakrishnan** †
**I.V. Ramakrishnan** ‡
**L.N. Kanal** †
Department of Computer Science
University of Maryland
College Park, MD. 20742

**Abstract**

Several combinatorial optimization problems are known to be NP-complete. As a consequence, fast parallel algorithms for finding optimal solutions for such problems using a polynomial number of processors are unlikely. An important practical approach to solving such problems on parallel machines is to seek approximate algorithms for them. In this paper, we present an approximate algorithm for the 0-1 knapsack problem on the parallel random access machine model. Our algorithm takes an $\epsilon$, $0 < \epsilon < 1$, as an input parameter and finds a solution such that the ratio of its deviation from the optimal solution is at most $\epsilon$. Our algorithm requires $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$ time and uses at most $\frac{n^{2.5}}{\epsilon^{1.5}}$ processors. In contrast to a naive algorithm that requires significantly more processors, our algorithm exploits the relationship among the sets of solutions to the problem generated in the intermediate steps to reduce the number of solutions considered. This in turn results in a reduction in the processor requirements.

## 1. Introduction

There is a growing interest in the design of parallel algorithms for SIMD machines. In particular, the design of SIMD computer algorithms for problems such as sorting, matrix computations, and graph and network computations has received widespread attention. In contrast, very little attention has been paid to designing SIMD computer algorithms for such combinatorially hard problems as knapsack problems, vertex covering, and set covering. In this paper we will describe an approach to the design of parallel algorithms for one such problem, namely, the 0-1 knapsack problem (see Section 2 for definition). This problem is known to be NP-complete [GJ79] and as a consequence, a polynomial time parallel algorithm that finds an optimal solution using a polynomial number of processors is unlikely. An important practical approach to solving such problems has been to design fast algorithms that find approximate instead of optimal solutions. Here we present parallel algorithms for finding approximate solutions to the knapsack problem.

Several approximate algorithms on sequential machines for various NP-complete problems have been reported (see [GJ76] for a bibliography of some early work). Many of these algorithms use a greedy strategy [HS78, Chapter 4]. It has been shown by Anderson and Mayr [AM84] that greedy algorithms for several graph problems are inherently sequential and hence, fast parallel implementations of these sequential algorithms are unlikely. Several approximate algorithms currently known appear to be highly sequential and hence, there is a need to develop new techniques for finding approximate solutions to some of these NP-complete problems on parallel computers.

The model of parallel computation we use here is a parallel random access machine (PRAM) [FW78]. We allow several processors to read from the same location simultaneously. However, simultaneous writing into the same memory location is disallowed. Such a model is usually known as a concurrent read exclusive write (CREW) PRAM. An important goal of research on algorithms for this model has been to design poly-logarithmic time (that is, $O(\log^k n)$ for some k) algorithms. The algorithm reported in this paper finds an approximate solution to the knapsack problem in poly-logarithmic time using a polynomial number of processors. The solution found by our algorithm will be worse than the optimal solution by at most a factor of $\epsilon$, for any given $\epsilon$, $0 < \epsilon < 1$. Our algorithm requires $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$ time and at most $\frac{n^{2.5}}{\epsilon^{1.5}}$ processors. A naive algorithm would generate several partial solutions and merge them to obtain the final solution. As explained in Section 3.3, our algorithm exploits the relationship among these partial solutions in order to reduce the number of such solutions to be generated. This leads to a reduction in the processor requirements of our algorithm. The processor bound is established using an interesting combinatorial result.

The only known parallel algorithm for this problem is by Peters and Rudolph [PR84]. Their algorithms achieve speedups using a limited number of processors by exploiting the explicit parallelism in the inner loop of the sequential algorithm of Lawler [LE79]. The divide-and-conquer algorithm presented there uses a simple merging procedure and hence requires substantially more processors compared to our algorithm. Our attempt has been to explore new properties of the problem that can be exploited in the parallel algorithm.

† Work supported in part by grants from the National Science Foundation to the Machine Intelligence and Pattern Analysis Laboratory.

‡ Present Address - Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY. 11794. The work of the second author supported in part by the National Science Foundation under grant number ECS-84-04399 and by the Office of Naval Research under contract N00014-84-K-0530.

The rest of the paper is organized as follows. In the next section we introduce the knapsack problem. In Section 3, we present a parallel approximate algorithm for the 0-1 knapsack problem. The combinatorial result that is made use of in proving the processor bound is also presented in that section. Some concluding remarks are given in section 4. Several implementation details as well as proofs of some theorems are omitted in this preliminary report for the sake of brevity. Proofs of other theorems are sketched briefly. Complete proofs and other implementation details will appear in a forthcoming paper [GRK86]

## 2. The 0-1 Knapsack Problem

The 0-1 knapsack problem is defined as follows. We are given n elements having positive integer valued profits $p_1, p_2, \cdots, p_n$ and positive integer valued weights $w_1, w_2, \cdots, w_n$. We are also given a positive integer valued knapsack capacity K and the optimization problem is to find a set $S \subseteq \{1, 2, ..., n\}$ that maximizes $\sum_{i \in S} p_i$ subject to the constraint $\sum_{i \in S} w_i \leq K$.

We can assume that all the weights are at most equal to the capacity K. Any solution to the knapsack problem is a set of integers which is a subset of the set $\{ 1, 2, ..., n \}$. We will denote the profit of a solution S by P(S) and accumulated weight by W(S). Thus, $P(S) = \sum_{i \in S} p_i$, and $W(S) = \sum_{i \in S} w_i$. S is a *feasible solution* if $W(S) \leq K$. A feasible solution $S^*$ is an optimal solution if $P(S^*) \geq P(S)$ for any other feasible solution S. We denote the profit of an optimal solution by $P^*$. In the rest of the paper the term solution will stand for a feasible solution unless explicitly stated.

An approximate solution to the 0-1 knapsack problem takes as its input a real number $\epsilon$, $0 < \epsilon < 1$, and finds a solution with profit P such that $P^* - P \leq \epsilon P^*$. Several such algorithms on sequential machines have been reported in the past [IK75, LE79, MO81, SS75]. Our aim in this paper is to design such an approximate algorithm on the PRAM model.

## 3. Parallel Approximate Algorithm

In this section we first present a simple parallel algorithm that finds an optimal solution to the problem. However, this algorithm would require an exponential number of processors to find an optimal solution in polynomial time. We then outline a scheme for obtaining an approximate solution in poly logarithmic time using polynomial number of processors. This approximate algorithm is further refined to obtain a more efficient algorithm with reduced processor requirements.

We will use the following notation in the description of the algorithms below. $R_i$, $S_i$, $T_i$, and $U_i$ ($i = 1, 2, ...$) will denote solutions to the knapsack problem and $G_i$, $H_i$ ($i = 1, 2, ...$) will denote sets of solutions. We will also assume, for the sake of simplicity that n is a power of 2.

All our algorithms use the following dominance relation between solutions in order to discard partial solutions.

*Definition*

Let $S_1$ and $S_2$ be two solutions. Then $S_1 < S_2$ ($S_2$ dominates $S_1$) if $P(S_1) \leq P(S_2)$ and $W(S_1) \geq W(S_2)$.

●

Observe that if $S_i < S_j$ and $S_k$ is a solution that is disjoint from both $S_i$ and $S_j$ then $S_i \cup S_k$ will be dominated by $S_j \cup S_k$. Thus, for every feasible solution that can be obtained from $S_i$ by adding more elements, there is another feasible solution that can be obtained from $S_j$ having possibly greater profit. Hence, given any set of feasible solutions, we can remove all solutions from this set that are dominated by other solutions in the same set.

Note that the dominance relation is transitive. Using this property, it is easy to show the following.

**Lemma 3.1:** Let I be a subset of $\{1, 2, ..., n\}$ and let $G_i$ be the set of all feasible solutions that are subsets of I and are not dominated by any other subset of I. Let $S_j$ be any feasible solution that is also a subset of I. Then, either $S_j \in G_i$ or there is a solution $S_k \in G_i$ such that $S_j < S_k$.

●

Let F(i,j) denote the set of all feasible solutions that are subsets of the set $\{i, i+1, ..., j\}$ and are not dominated by any other feasible solution in the same set.

### 3.1. A Simple Exact Parallel Algorithm

Algorithm 1 below is a simple parallel algorithm for finding an exact solution to the 0-1 knapsack problem.

**Algorithm 1**

```
[1]  Call FEASIBLE(1,n) (given below) to find the set F(1,n).
[2]  Find the solution with the maximum profit from F(1,n) in parallel.
procedure FEASIBLE(i,j)
     (* This procedure returns the set of solutions F(i,j) *)
[3]  If $i = j$ then return the set containing the solutions $\phi$
     and $\{i\}$ with $P(\phi) = W(\phi) = 0$, and $P(\{i\}) = p_i$, $W(\{i\}) = w_i$.
     otherwise
[4]  Compute $G_1 = F(i,(i+j)/2)$ using FEASIBLE(i,(i+j)/2)
     and $G_2 = F((i+j)/2 + 1, j)$ using FEASIBLE((i+j)/2 + 1,j) in parallel.
[5]  For each $S_i \in G_1$ do in parallel -
         For each $S_j \in G_2$ do in parallel -
         Compute $S_i \cup S_j$. Since this is a disjoint union,
         the profit of $S_i \cup S_j$ is $P(S_i) + P(S_j)$ and its weight
         is $W(S_i) + W(S_j)$. If $W(S_i \cup S_j) > K$ then discard
```

this infeasible solution, else add it to the set of solutions $G_3$.

[6] Remove all dominated solutions from $G_3$ and return the set of solutions thus obtained

●

**Analysis**

The correctness of the procedure FEASIBLE(i,j) is easy to establish using induction on j–i. Clearly, F(i,j) is correctly set up when i–j = 0. Assume now that FEASIBLE(i,j) returns the set F(i,j) for all values of i–j < n–1. Now, let i–j = n–1. Consider any solution S that is a subset of {i, i+1, ..., j}. Let the intersection of S with {i,..., (i+j)/2} be $S_1$ and its intersection with {(i+j)/2 + 1, ..., j} be $S_2$. Since F(i, ..., (i+j)/2) and F((i+j)/2 + 1, ..., j) are assumed to be correctly computed by the recursive calls, by lemma 3.1, $S_1$ is either present in F(i, ..., (i+j)/2) or there is some solution $S_i$ in that set that dominates $S_1$. In the latter case S will be dominated by $S_i \cup S_2$. A similar argument can be applied to $S_2$. We can thus show that a feasible solution is in the set returned by FEASIBLE(1,n) if and only if it is not dominated by any other solution.

Observe that if two solutions have the same profit value, one of them will be discarded because it will be dominated by the other. Similarly, if two solutions have the same weight, one will be discarded. Thus, there will be at most one solution for each integer value of the profit and for each value of the weight in the set of solutions returned by procedure FEASIBLE. Therefore, there will be at most $c$ solutions in these sets, where $c = \min\{P^*, K\}$.

The recursive procedure FEASIBLE goes through log n stages of recursion. In each stage, step [5] requires O(1) time and $c^2$ processors as we require one processor for each pair of solutions $S_i$, $S_j$ and there are at most $c^2$ such pairs. Step [6] can be executed in O(log $c$) time using $c^2$ processors as removing the dominated solution involves finding the minimum among at most $c^2$ values (details are omitted here). Since the two recursive calls in procedure FEASIBLE are executed in parallel, at most O(n) sets of solutions will be constructed simultaneously. Hence, the total processor requirement for generating the set F(1,n) is n$c^2$ and the time required is O(log nlog $c$). Finding the maximum profit solution from F(1,n) will require O(log $c$) time and $c$ processors and hence the time complexity of Algorithm 1 is O(log nlog $c$) and the processor requirement is n$c^2$.

## 3.2. Approximation Scheme

Since $c$ can be arbitrarily large, the time complexity of Algorithm 1 is not a true logarithmic function of n and the processor requirement is not a true polynomial of n. However, we can scale down the profits [IK75, LE79, SS75] to obtain a bound on $c$ that is a polynomial in n. We divide all the profit values by a scale factor $\lambda$. For the problem with these scaled down profits, the optimal profit cannot be more than $\frac{P^*}{\lambda}$. The scale factor $\lambda$ is chosen to obtain an upper bound on $\frac{P^*}{\lambda}$ that is a polynomial in n. The scale factor is derived as follows.

Let the profits and weights be arranged such that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$. Now, choose j such that $(w_1 + w_2 + \cdots + w_j) \leq K < (w_1 + w_2 + \cdots + w_j + w_{j+1})$ Let

$$P_0 = \max\{p_1 + p_2 + \cdots + p_j\ ,\ p_m\}, \text{ where } p_m = \max_i\{p_i\}$$

Let $\epsilon$ be the allowed deviation from the optimal solution, $0 < \epsilon < 1$. That is, we are required to find a solution with profit P such that $P^* - P \leq \epsilon P^*$.

**Theorem 3.1:** If the profits are scaled down using a scale factor $\lambda = \epsilon P_0/n$ then an optimal solution to the problem with the scaled profits will be an approximate solution to the original problem satisfying the desired bounds. Moreover, the profit of any solution to the scaled problem will not exceed $2n/\epsilon$.

**Proof:** See [LE79].

A parallel algorithm based on the above derivation is given below.

**Algorithm 2**

[1] Sort the weights and profits into non-increasing order of the ratio $p_i/w_i$.

[2] Compute $P_0$ and scale factor $\lambda$ using the expression given above.

[3] Scale all profits $p_i$ to obtain $r_i = \lfloor p_i/\lambda \rfloor$.

[4] Use Algorithm 1 on the modified problem with profits $(r_1, r_2, \ldots, r_n)$, weights $(w_1, \ldots, w_n)$ and capacity K. Let $P_1$ be the profit of the solution to the modified problem found by Algorithm 1.

[5] The approximate solution to the original problem is the solution found above and its profit P is $\lambda P_1$.

**Complexity**

The sorting step [1] in the above algorithm will require O(log n) time and n processors using the algorithm in [AK83]. To compute $P_0$, we need to find out the integer j such that $w_1 + w_2 + \cdots + w_j \leq K < w_1 + w_2 + \cdots + w_j + w_{j+1}$. Such a j can be found in O(log n) time with n processors. Step [3] requires O(1) time and n processors. Thus the time and processor requirements of Algorithm 2 are dominated by the requirements of step [4]. From the analysis of the complexity of Algorithm 1, it follows that Algorithm 2 requires O(log n.log c) time and n$c^2$ processors. Since $c$ is O(n/$\epsilon$) for the approximate algorithm, the time complexity is $O(\log^2 n + \log n . \log \frac{1}{\epsilon})$ and the number of processors required is $\frac{n^3}{\epsilon^2}$ †.

### 3.3. An Improved Algorithm

We can obtain a more efficient parallel algorithm (in terms of the processor time product) by modifying the merging step used in procedure FEASIBLE of Algorithm 1. This modification, as we shall see later, will increase the time by another factor of (log n) and decrease the worst case processor requirement to $\frac{n^2}{\epsilon}\sqrt{\frac{n}{\epsilon}}$. The processor-time product of the resulting algorithm is significantly smaller. This reduction in the number of processors is significant since the value of $\epsilon$ is usually very small.

Notice that the merging step of procedure FEASIBLE is highly wasteful since, out of the $c^2$ solutions that are generated, at most $c$ are retained. We need not generate several of these solutions if we make effective use of the information from previous stages of the procedure FEASIBLE. Recall that in procedure FEASIBLE(i,j), F(i,j) is computed by first computing F(i,(i+j)/2) and F((i+j)/2 + 1, j) and then merging them. Let $G_1$ denote F(i,(i+j)/2) and let $G_2$ denote F((i+j)/2 + 1, j). Instead of merging $G_1$ and $G_2$ directly by taking the union of every solution from $G_1$ with every solution from $G_2$, we first merge $G_1$ with the two sets that were merged to obtain $G_2$, namely, F((i+j)/2 + 1, 3(i+j)/4) and F(3(i+j)/4 + 1, j). Let $G_3$ and $G_4$ denote F((i+j)/2 + 1, 3(i+j)/4) and F(3(i+j)/4 + 1, j) respectively. Let $H_1$ and $H_2$ denote the result of merging $G_1$ with $G_3$ and $G_4$ respectively. That is, $H_1$ is the set of all feasible solutions that are subsets of the set {i, i+1, ..., (i+j)/2}∪{(i+j)/2 + 1, ..., 3(i+j)/4} and are not dominated by any other feasible solution in the same set. Similarly, $H_2$ is the set of all non-dominated feasible solutions that are subsets of the set {i, i+1, ..., (i+j)/2}∪{3(i+j)/4 + 1, ..., j}. Figure 3.1 shows the relationship among these sets. The reduction in the number of solutions follows from the fact that when $G_1$ is merged with $G_3$ and all dominated solutions are removed, any solution that is discarded will not form part of a non-dominated solution in the set obtained by merging $G_1$ and $G_2$. The same is the case with the solutions that are discarded when merging $G_1$ and $G_4$. This result is proved below.

Each non-empty solution $S_i$ in $G_2$ is either a solution $U_i$ in $G_3$, or a solution $V_j$ from $G_4$, or the solution $U_i \cup V_j$ (where $U_i$ and $V_j$ are non-empty). If $S_i$ is $U_i$, its union with solutions in $G_1$ would already have been determined while computing $H_1$. Similarly, if $S_i$ is $V_j$, its union with the solutions in $G_1$ will be present in $H_2$. So, we need now consider only the solutions $U_i \cup V_j$ in $G_2$.

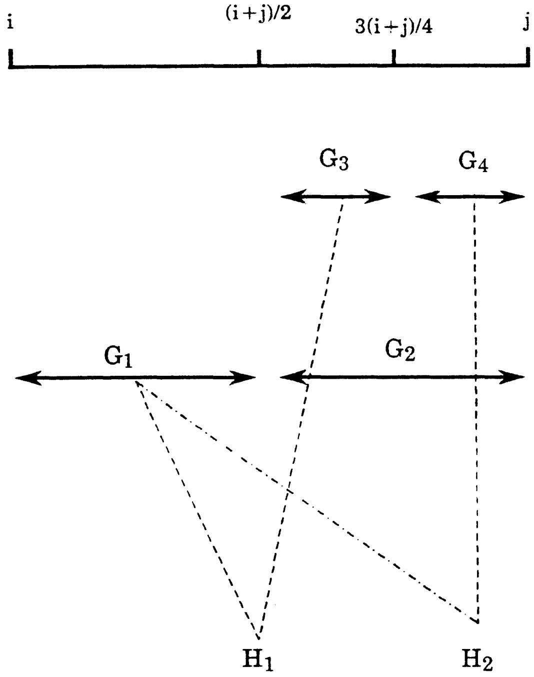


Figure 3.1. Broken lines indicate merging.

Let $f(U_i)$ be the set of solutions from $G_1$ whose union with $U_i$ occurs in $H_1$. That is,

$$f(U_i) = \{R_k \mid R_k \in G_1 \text{ and } U_i \cup R_k \in H_1\} \qquad (3.7)$$

Similarly, let $f(V_j)$ be the set of all solutions from $G_1$ whose union with $V_j$ occurs in $H_2$. That is,

$$f(V_j) = \{R_k \mid R_k \in G_1 \text{ and } V_j \cup R_k \in H_2\} \qquad (3.8)$$

The following lemma is the key to the reduction in the number of solutions generated.

**Lemma 3.2:** Let $S_i$ be $U_i \cup V_j$ and let $R_1$ be a solution in $G_1$ that is not in $f(U_i) \cap f(V_j)$. Then one of the following is true - (1) $S_i \cup R_1 < U_i \cup T_1$ for some $T_1$ in $H_2$, or (2) $S_i \cup R_1 < V_j \cup T_2$ for some $T_2$ in $H_1$.

**Proof:** Let $R_1$ be in $f(U_i)$ and not in $f(V_j)$. This implies that $R_1 \cup V_j$ does not occur in $H_2$. By Lemma 3.1, there is a solution $T_1$ in $H_2$ such that $R_1 \cup V_j < T_1$. Now, $S_i \cup R_1 = U_i \cup V_j \cup R_1 < U_i \cup T_1$, since $T_1$ and $U_i$ are disjoint.

If $S_i \cup R_1$ is a feasible solution then $U_i \cup T_1$ is also a feasible solution. Similarly, if $R_1$ is in $f(V_j)$ and not in $f(U_i)$ then there is a solution $T_2$ in $H_1$ such that $U_i \cup R_1 < T_2$.

†Here we use the property that any algorithm that can be executed in time T(n) using O(p) processor can be executed in O(T(n)) time using p processors.

This implies that $S_i \cup R_1 < V_j \cup T_2$. If $R_1$ belongs to neither $f(U_i)$ nor $f(V_j)$, then both conditions (1) and (2) hold.

●

Lemma 3.2 shows that the union of certain pairs of solutions from $G_1$ and $G_2$ need not be generated. In particular, for every solution $U_i \cup V_j$ in $G_2$, we need generate only its union with every solution in $f(U_i) \cap f(V_j)$. This is implemented by replacing the merging step in procedure FEASIBLE of Algorithm 1 (step [4]) with the following procedure :

```
*procedure* MERGE($G_1$,$G_2$,i,j)
    (* Here $G_2$ is F(i,j) *)
[1] If i=j then merge $G_1$ and $G_2$ by taking the union of
    each element in $G_2$ with every element of $G_1$ in parallel.
    Compute the union of the resulting set with $G_1$ and $G_2$
    and remove all dominated solutions. Return the set thus
    obtained. (* There will be only two solutions in $G_2$ if i=j
    and hence, this step can be executed in O(1) time using
    $c$ processors. *)
    **otherwise**
[2] Let $G_3$ = F(i,(i+j)/2) and $G_4$ = F((i+j)/2 + 1, j). (*
    These are available from the previous stage of procedure
    FEASIBLE. *)
    Call MERGE($G_1$,$G_3$,i,(i+j)/2) to obtain $H_1$
    and in parallel call MERGE($G_1$,$G_4$,(i+j)/2 + 1, j) to
    obtain $H_2$.
[3] For each non-empty solution $S_i \in G_2$ that is of the
    form $U_i \cup V_j$, where $U_i \in G_3$ and $V_j \in G_4$ and $U_i$, $V_j$
    are non-empty, do in parallel -
    Identify the set of solutions $f(U_i) \cap f(V_j)$. Generate the
    solution $S_i \cup R_k$ for each $R_k$ in $f(U_i) \cap f(V_j)$ in parallel.
    If $W(S_i \cup R_k) > K$ then discard this solution, else add
    it to the set of solutions $H_3$
[4] Compute the union of $H_1$, $H_2$, and $H_3$ and remove all
    dominated solutions to obtain $H_4$.
[5] Compute the union of $H_4$, $G_1$, and $G_2$ and remove all
    dominated solutions. Return the resulting set.
```

●

We prove the correctness of the MERGE procedure below.

**Theorem 3.1:** Let $G_1$ be F(k,l) and $G_2$ be F(i,j), where $k \leq l < i \leq j$. Let $H_5$ be the set of solutions generated in step [5] of the procedure call MERGE($G_1$,$G_2$,i,j). Then $H_5$ is the set of all feasible solutions that are subsets of the set $\{k, ..., l\} \cup \{i, ..., j\}$ and are not dominated by any other solution in the same set.

**Sketch of Proof:** We prove the result using induction on (j–i). The base case is correctly handled by step [1]. Let us assume that the recursive calls in step [2] return sets $H_1$ and $H_2$ which contain all the feasible non-dominated solutions that are subsets of the set $\{k, ..., l\} \cup \{i, ..., (i+j)/2\}$ and the set $\{k, ..., l\} \cup \{(i+j)/2 + 1, ..., j\}$ respectively. To show that $H_5$ is the set of all non-dominated solutions that are subsets of $\{k, ..., l\} \cup \{i, ..., j\}$, we have to show that for any feasible solution $S \subseteq \{k, ..., l\} \cup \{i, ..., j\}$, S is in $H_5$ if and only if it is not dominated by some solution $S_1$ in $H_5$.

Since $G_1$ is F(k,l) and $G_2$ is F(i,j), we need concern ourselves only with solutions S of the form $R_1 \cup T_1$, where $R_1$ is in $G_1$ and $T_1$ is in $G_2$.

(I) Sufficiency: We need to show that if $S \notin H_5$ then $S < S_1$ for some $S_1$ in $H_5$.

If $S \notin H_5$ then either S was generated in step [3] and eliminated in steps [4] or [5], or S was not generated in step [3]. If S was generated and eliminated then obviously $S < S_1$ for some $S_1 \in H_5$. Assume now that S was not generated at all. If S is of the form $R_1 \cup T_1$ and $T_1$ is either equal to $U_1$ for some $U_1 \in G_3$, or equal to $V_1$ for some $V_1 \in G_4$ then by the induction hypothesis and lemma 3.1, either S is present in $H_1$ or $H_2$, or $S < S_1$ for some $S_1$ in $H_1$ or $H_2$. Since $H_1$ and $H_2$ are used in step [4], there must be some solution in $H_5$ that dominates S. Now, if S is of the form $T_1 \cup R_1$, and $T_1 = U_1 \cup V_1$ for some $U_1 \in G_3$, $V_1 \in G_4$ then we can show, using Lemma 3.2 that S would be dominated by some solution that is already in $G_2$, $G_1$, $H_1$, or $H_2$, or is generated in step [3].

(II) Necessity: We want to show that if $S < S_1$ for some $S_1$ in $H_5$ then $S \notin H_5$.

This is trivial since all dominated solutions from the solutions that are generated are eliminated in steps [4] and [5].

Thus, the set $H_5$ generated in step [5] is the set of all feasible solutions that are subsets of $\{k, ..., l\} \cup \{i, ..., j\}$ and are not dominated by any other solution in the same set.

●

From Theorem 3.1 it is clear that if procedure MERGE is called with $G_1$ = F(1,n/2) and $G_2$ = F(n/2 + 1, n), then the set of solutions returned would be F(1,n).

**Complexity**

Any call to procedure MERGE will go through (log n) stages at most. Step [1], where the recursion halts, will require O(1) time and $c$ processors. At any instant, at most n/2 simultaneous executions of step [1] may be going on. Thus, this step contributes O(n$c$) to the overall processor requirement. We implement step [3] as follows.

After the sets $H_1$ and $H_2$ have been generated we construct, for each $U_i \in G_3$, the set $f(U_i)$ defined by equation (3.7) using $c$ processors in O(log $c$) time. Also, the total number of solutions in this set is stored along with each $U_i$. A similar computation is done for each $V_j$ in $G_4$. Now, for each $S_i$ in $G_2$ we assign one processor. If $S_i$ is of the form $U_i \cup V_j$ then the processor checks the number of solutions in the sets $f(U_i)$ and $f(V_j)$. The minimum of these two numbers is found and these many processors are assigned to $S_i$. We can use these processors to identify the solutions that are common to $f(U_i)$ and $f(V_j)$ in O(1) time. The solutions $S_i \cup R_k$ are generated for each solution $R_k$ that is in this common set in O(1) time.

The number of processors required for the computation of the solutions $S_i \cup R_k$ is thus $p = \sum_{U_i \cup V_j \in G_2} \min\{$number of solutions in $f(U_i)$, number of solutions in $f(V_j)\}$. Taking into account the number of processors needed for the initial computations on $H_1$ and $H_2$, the total number of processors required for step [3] is $\max(p,c)$ and the time required is $O(\log c)$. Since the two recursive calls in step [2] are executed in parallel, the overall processor requirement of procedure MERGE would be $n(\max\{p,c\})$ and the time required would be $O(\log n . \log c)$.

●

Now, Algorithm 1 is modified by replacing the merging step in procedure FEASIBLE with a call to procedure MERGE. We can now use this modified algorithm to find a solution to the scaled problem and get an approximate solution. The time required by the modified algorithm is given by Theorem 3.2 below.

**Theorem 3.2:** An approximate solution having relative error at most $\epsilon$ can be found using the modified parallel algorithm in time $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$.

**Sketch of Proof:** The procedure FEASIBLE goes through (log n) recursive stages. In each stage, the MERGE procedure requires $O(\log n . \log c)$ time. Thus, the time required by procedure FEASIBLE is $O(\log^2 n \log c)$ and this is greater than the time required for the rest of the steps in Algorithm 2. Since $c$ is $O(\frac{n}{\epsilon})$ for the approximate algorithm, the time required is $O(\log^2 n \log \frac{n}{\epsilon})$ which is $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$.

●

The worst case processor bound for the modified algorithm is obtained using the following combinatorial result.

Let G be an undirected bipartite graph (A,B,E) where A and B are the sets of vertices and E is the set of undirected edges. Let $a_i$ be a non-negative integer weight associated with each vertex i in A and $b_j$ be a non-negative integer weight associated with each vertex j in B. The edges and weights satisfy the following constraints -

$$|E| \leq c \tag{3.9}$$

$$\sum_{i \in A} a_i \leq c \tag{3.10}$$

$$\sum_{j \in B} b_j \leq c \tag{3.11}$$

**Theorem 3.3:** For any bipartite graph G and set of weights satisfying constraints (3.9), (3.10), and (3.11), the sum $M = \sum_{(i,j) \in E} \min\{a_i, b_j\}$ is at most $c\sqrt{c}$.

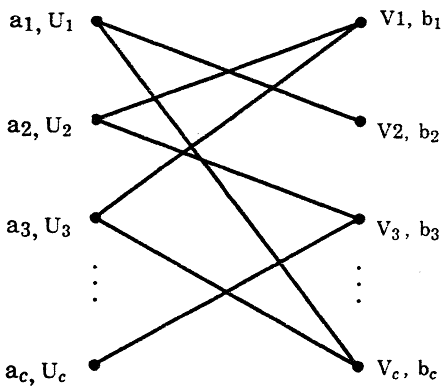
**Sketch of Proof:** The result is proved by showing that given any bipartite graph and set of weights satisfying constraints (3.9) - (3.11), we can transform the graph into a standard graph by applying a series of transformations, each of which increases the value M, or at worst, leaves M unchanged. This standard structure will be an almost complete bipartite graph having close to $\sqrt{c}$ vertices on both sides and nearly equal weights on the vertices. The value of M cannot be increased above the value obtained for this structure. The value of M for this graph can be shown to be at most $c\sqrt{c}$.

●

**Theorem 3.4:** Algorithm 2 using the modified merging procedure requires $\frac{n^{2.5}}{\epsilon^{1.5}}$ processpors.

**Proof:** From the complexity analysis of procedure MERGE, we see that this procedure requires $n.(\max\{p,c\})$ processors, where $p = \sum_{U_i \cup V_j \in G_2} \min\{$number of solutions in $f(U_i)$, number of solutions in $f(V_j)\}$, and $c$ is $O(\frac{n}{\epsilon})$. The processor requirement of Algorithm 2 is dominated by the requirement of procedure MERGE. Theorem 3.3 can be used to determine p. We define a bipartite graph associated with the set $G_2$. The vertices of this graph correspond to each $U_i$ in $G_3$ and each $V_j$ in $G_4$. An edge between vertices $U_i$ and $V_j$ is present in this graph if the solution $U_i \cup V_j$ is present in $G_2$. The weight associated with $U_i$ is the number of solutions in $f(U_i)$, where $f(U_i)$ is defined by equation (3.7). Similarly, the weight associated with $V_j$ is the number of solution in $f(V_j)$, where $f(V_j)$ is defined by equation (3.8) (see Fig 3.2). There are at most $c$ elements in $G_2$ and hence the number of edges in this graph is at most $c$. This satisfies constraint (3.9). The sum of weights on all the $U_i$ is at most equal to the number of solutions in the set $H_1$ which is less than or equal to $c$. This shows that constraint (3.10) is satisfied. The sum of weights on all the $V_j$ is at most equal to the number of solutions in $H_2$ which is at most $c$. Thus all three constraints are satisfied. Applying Theorem 3.3, we see that p, which is $\sum_{U_i \cup V_j \in G_2} \min\{$number of solutions in $f(U_i)$, number of solutions in $f(V_j)\}$, is at most $c\sqrt{c}$.

Thus, the total processor requirement is at most $nc\sqrt{c}$. Since $c$ is at most $2n/\epsilon$ for the approximate algorithm, the total processor requirement is $O(\frac{n^{2.5}}{\epsilon^{1.5}})$. Using the property that any algorithm that runs in time t using O(p) processors can be executed in O(t) time using p processors, we see that the processor requirement for Algorithm 2 is at most $\frac{n^{2.5}}{\epsilon^{1.5}}$.



Edge $(U_i, V_j)$ => $U_i \cup V_j$ occurs in $G_2$

$a_i = \| f(U_i) \|$

$b_j = \| f(V_j) \|$

Fig 3.2. Construction for proof of processor bound.

●

We have thus shown that using a parallel algorithm we can find an approximate solution to the given knapsack problem whose profit P is related to the profit $P^*$ of the optimal solution by the inequality $P^* - P \leq \epsilon P^*$. The time required by the parallel algorithm is $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$ and the number of processors required is at most $\frac{n^{2.5}}{\epsilon^{1.5}}$.

## 4. Conclusions

In this paper, we have presented an approximate algorithm for the 0-1 knapsack problem on the CREW PRAM model. Our algorithm takes an $\epsilon$, $0 < \epsilon < 1$, as an input parameter and finds a solution such that the ratio of its deviation from the optimal solution is at most $\epsilon$. Our algorithm requires $O(\log^3 n + \log^2 n \log \frac{1}{\epsilon})$ time and uses at most $\frac{n^{2.5}}{\epsilon^{1.5}}$ processors. In contrast to a naive algorithm that requires significantly more processors, our algorithm exploits the relationship among the partial solutions to reduce the number of such solutions to be generated. This in turn results in reductions in the processor requirements. (Notice that the divide-and-conquer algorithm presented in [PR84] has, in fact, the same processor and time bounds as the algorithm described in section 3.2 and hence is less efficient compared to the algorithm presented in section 3.3.)

Although the worst case processor requirement of our algorithm is $\frac{n^{2.5}}{\epsilon^{1.5}}$, we strongly conjecture that on an average our algorithm would only require $O(\frac{n^2}{\epsilon})$ processors. It will be interesting to analytically establish an average case processor bound.

The approach we have adopted in our algorithm is general enough to solve several other problems that can be formulated along similar lines as the knapsack problem (details will appear in a forthcoming paper [GRK86]).

### Acknowledgement

The first author would like to thank David Mount for several helpful discussions.

### References

[AK83]
M. Ajtai, J. Komlos, and E. Szemeredi, "An O(nlog n) Sorting Network", Proceedings of the 15th Annual ACM Symposium on the Theory of Computing, 1983, pp.1-9.

[AM84]
R. Anderson and E. Mayr, "Parallelism and Greedy Algorithms", T.R. STAN-CS-84-1003, Computer Science Department, Stanford University, 1984.

[FW78]
S. Fortune and J. Wyllie, "Parallelism in Random Access Machines", Proceedings of the 10th Annual Symposium on the Theory of Computing, 1978, pp.114-118.

[GJ76]
M.R. Garey and D.S. Johnson, "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography", in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub (ed.), Academic Press, N.Y., 1976.

[GJ79]
M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the theory of NP-Completeness*, Freeman, N.Y., 1979.

[GRK86]
P.S. Gopalakrishnan, I.V. Ramakrishnan, and L.N. Kanal, "Parallel Approximate Algorithms for Combinatorially Hard Optimization Problems", Technical Report, Department of Computer Science, University of Maryland, College Park, MD.

[HS78]
E. Horowitz and S. Sahni, *Fundamentals of Algorithms*, Computer Science Press, Rockville, MD., 1978.

[IK75]
O.H. Ibarra and C.E. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems", Journal of the ACM, 22, 4, 1975, pp.463-468.

[JD74]
D.S. Johnson, "Approximate Algorithms for Combinatorial Problems", Journal of Computer and Systems Sciences, 9, 1974, pp.256-278.

[LE79]
E.L. Lawler, "Fast Approximation Algorithms for Knapsack Problems", Mathematics of Operations Research, 4, 1979, pp.339-356.

[MO81]
M.J. Magazine and O. Oguz, "A Fully Polynomial Approximation Algorithm for the 0-1 Knapsack Problem", European Journal of Operations Research, 8, 3, 1981, pp.270-273.

[PR84]
J. Peters and L. Rudolph, "Parallel Approximation Schemes for Subset Sum and Knapsack Problems", 22nd Annual Allerton Conference on Communication, Control, and Computing, 1984, pp.671-680.

[SS75]
S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem", Journal of the ACM, 22, 1, 1975, 115-125.

# PARALLEL PROCESSING
# FOR
# QUADTREE PROBLEMS

Gee-Gwo Mei and Wentai Liu
Department of Electrical & Computer Engineering
North Carolina State University
Raleigh, NC. 27695-7911
(919)-737-2336

## ABSTRACT

Parallel architectures, based on the *two dimensional shuffle exchange (2DSE) network*, to solve a class of quadtree problems are presented. The quadtree problems are those can be solved by using quadtree as data structure. The 2DSE approach enables us to have a spectrum of functionally equivalent configurations, which use different number of processors and have different time complexities. The various configurations give us more choices to best-fit to the constraints of real world applications.

## 1. INTRODUCTION

The quadtree is a well known data structure in the areas of computer vision and image processing. It has been used for the algorithms for computing geometric properties such as areas and moment [1], perimeter [2], Euler number [3], and distance transform [4]. In addition, it has been used in digital image processing applications such as edge enhancement [5], image segmentation [6], smoothing [7], shape approximation [8] *et al.* For a tutorial on quadtree, see [9]. However, the quadtree approach we adopt, is how the quadtree manipulates data rather than a data compression scheme for image storing. Hardware architectures for solving similar problem can be found in the papers [10,11,12,13]. The distinctions between our approach and the previous ones are listed as follows.

(1) Versatility of the architecture: The architectures, which are used to solve the quadtree problems in this paper, can also be applied to tackle a class of two dimensional problems such as 2DFFT. For more details see [14]. The versatility of the architecture give us valuable tools to investigate more complicated vision functions, and construct more advanced vision systems.

(2) Various configurations for real world applications: The 2DSE approach enables us to have a spectrum of functionally equivalent configurations, which use different number of processors and have different time complexities. The various configurations give us more choices to best-fit to the constraints of real world applications.

(3) Common communication lines: All of the algorithms in this paper, plus a class of two dimensional algorithms, utilize the same set of lines among processors. This makes the 2DSE approach more economically feasible.

## 2. 1DSE AND 2DSE NETWORK

One-dimensional shuffle exchange (1DSE) network can perform the shuffle, unshuffle, exchange, and broadcast operations. A shuffle operation moves the data at the address **x** to the output address $\mathbf{S_1(x)}$, where $\mathbf{S_1(x)}$ is defined as follows:

$$\mathbf{S_1(x)} =$$
$$(2x \text{ MOD } N) + 1 \qquad \text{if } x \ge \frac{N}{2}$$
$$2x \text{ MOD } N \qquad \text{if } 0 \le x < \frac{N}{2}.$$

An unshuffle operation $U_1$ perform the function inverse to the shuffle operation. Where $\mathbf{U_1(x)}$ is defined as follows:

$$\mathbf{U_1(x)} =$$
$$\frac{x}{2} \qquad \text{if x is an even number.}$$
$$\frac{x+N-1}{2} \qquad \text{if x is an odd number.}$$

The exchange operation is defined by $\mathbf{E(x)}$ as follows:

$$\mathbf{E_1(x)} = x_{n-1}\,2^{n-1} + \ldots.. + x_1 2 + \bar{x}_0.$$

The broadcast operation is defined by $\mathbf{B_1(x)}$ as a mapping from **x** to two positions $x_{n-1}\,2^{n-1} + \ldots.. + x_1 2 + x_0$ and $x_{n-1}\,2^{n-1} + \ldots.. + x_1 2 + \bar{x}_0$.

A two-dimensional shuffle exchange network can perform the two dimensional shuffle, unshuffle, exchange, and broadcast operations. A two-dimensional (2D) shuffle operation $S_2$ and unshuffle operation $U_2$ move the data at the address **(x,y)** to the output address $\mathbf{S_2(x,y)}$, and $\mathbf{U_2(x,y)}$. Where $S_2(x,y)$ is defined as follows:

$$\mathbf{S_2(x,y)} = (\mathbf{S_1(x)}, \mathbf{S_1(y)}) \tag{2.1}$$

$$\mathbf{U_2(x,y)} = (\mathbf{U_1(x)}, \mathbf{U_1(y)}) \tag{2.2}$$

An 2D exchange element is a switch, with four inputs and four outputs, which can perform all kinds of interchanges of four inputs (*i.e.* it is equivalent to a 4x4 crossbar network).

The 2D broadcast operation is defined by $\mathbf{B_2(x,y)}$ as a mapping from **x** to four positions, which are the same as the inputs of the 2D exchange elements.

## 3. MATHEMATICAL DERIVATION

Due to the space limitation, we use two examples, area and centroid computation, to demonstrate the applicability of 2DSE architectures on the quadtree problems.

### 3.1. Area Calculation

The area of a image is defined as

$$A = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y) \tag{3.1}$$

where f(x,y)=1 if the pixel at (x,y) is black.
=0 otherwise, $0 \le x,y \le$ N-1.

$$x = x_{m-1}2^{m-1} + x_{m-2}2^{m-2} + \ldots\, x_1 2 + x_0$$

$$y = y_{m-1}2^{m-1} + y_{m-2}2^{m-2} + \ldots\, y_1 2 + y_0$$

The equation (3.1) can be denoted by the following binary representation:

$$A = \sum_{x_{m-1}}\sum_{y_{m-1}} \cdots \sum_{x_0}\sum_{y_0} f(x_{m-1}, y_{m-1}, \ldots, x_0, y_0) \tag{3.2}$$

The area can be computed in m steps, where m is equal to logN. The area computation results of the first stage can be calculated by equation (3.3) and saved in buffers $B_1$. $B_i$, which saves the computation results of the *i-th* stage, is a two dimensional array of registers denoted by indices x and y. $B_0$ saves the digitized image at the initial state.

$$B_1(x_{m-1}, y_{m-1}, x_{m-2}, y_{m-2}, \ldots x_1, y_1, 0, 0) =$$

0190-3918/86/0000/0452 $01.00 © 1986 IEEE

$$\sum_{x_0}\sum_{y_0} f(x_{m-1},y_{m-1},\dots,x_0,y_0) \tag{3.3}$$

Similarly, the area in the s-th stage is computed by the equation (3.4) and saved in buffer $B_s$.

$$B_s(x_{m-1},y_{m-1},\dots x_s,y_s,0,0,\dots,0,0,0.0) =$$

$$\sum_{x_{s-1}}\sum_{y_{s-1}} B_{s-1}(x_{m-1},y_{m-1},\dots x_{s\ 1},y_{s\ 1},0,0,\dots,0,0)$$

$$\text{for } s = 2, 3\dots. m. \tag{3.4}$$

The m-th stage area computations, which represent the total area of the image, are saved in the buffer $B_m(0,0)$.

### 3.2. Centroid Computation

The centroid of a image is defined as

$$(x_{center}\hat{\imath}+y_{center}\hat{\jmath}) = \frac{1}{A}\sum_x\sum_y(xf(x,y)\hat{\imath}+yf(x,y)\hat{\jmath}) \tag{3.5}$$

where A is equal to the area of the image.

The results in the s-th stage is computed by the equation (3.6), and saved in buffers, $A_s$, $B_{x,s}$, and $B_{y,s}$.

$$B_{x,s}\ (\ x_{m-1},y_{m-1},\dots x_{s-2},y_{s-2},x_{s-1},y_{s-1},0,0,0,0,\dots,0,0\ ) =$$

$$\sum_{x_{s-1}}\sum_{y_{s-1}} B_{x,s-1}\ (\ x_{m-1},y_{m-1},\dots x_{s-1},y_{s-1},0,0,\dots,0,0\ ) +$$

$$\sum_{x_{s-1}}\sum_{y_{s-1}} x_{s-1}2^{s-1}A_{s-1}\ (\ x_{m-1},y_{m-1},\dots x_{s-1},y_{s-1},0,0,\dots,0,0\ ) \tag{3.6}$$

Where A are functionally the same as B, which are defined in equations (3.3) and (3.4). The $B_{y,s}$ can be obtained by changing the the variable x to y in equation (3.6).

The centroid of an image can be obtained by equation (3.7) and (3.8).

$$x_{center} = \frac{B_{x,m}(0,0)}{A_m(0,0)} \tag{3.7}$$

$$y_{center} = \frac{B_{y,m}(0,0)}{A_m(0,0)} \tag{3.8}$$

## 4. ARCHITECTURES AND ALGORITHMS

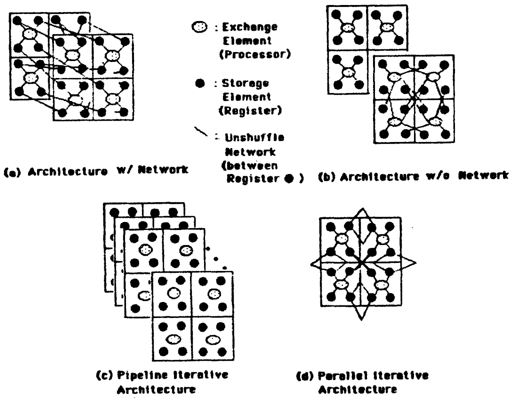

: Exchange Element (Processor)
: Storage Element (Register)
: Unshuffle Network (between Register ● )
(a) Architecture w/ Network
(b) Architecture w/o Network
(c) Pipeline Iterative Architecture
(d) Parallel Iterative Architecture

Figure 1: 2D Architectures

Figure 1 illustrate the architectures that can solve the quadtree problems with maximal parallelism based on 2DSE network. The black nodes in Figure 1 represent storage elements. Use centroid computation as example, these nodes stand for the places where the values of $B_x$, $B_y$, and A are stored. The dotted nodes represent exchange elements (*i. e.* processors) where all the computations are executed. In the case of centroid computation, these nodes executed the operations shown in equation (3.6).

Part (c) and (d) of Figure 1 show the two different ways: iteratively pipeline and iteratively parallel, to integrate processors into a working system.

Architectures in part (a) and (b) are functionally equivalent. The only difference is: in part (a), the window size of exchange elements is a constant one, but the size doubles when stage advances in part (b). Where the window size of exchange elements are defined as the index distance between any accessed data inputed to the exchange elements. The role of the 2DSE network is to unshuffle the data so that the window size of exchange elements can remain the same. Two dimensional input and output is the common characteristics and necessary requirement of the 2D architecture shown in the Figure 1.

The summation mechanism $SUM_2^i$, are exchange elements difined to executed the operations in the equation (3.3), with window size $2^i$, where i is the stage number. The function sums up the value from four subdived quadrants, is used in both the area calculation and centroid computation.

The centroid computation mechanism $CENX_2^i$, and $CENY_2^i$, are exchange elements defined to execute the operations shown in the equation (3.6) with window size $2^i$. Thus, the centroid of the image can be computed with time complexity of O(logN) by the architecture shown in Figure (1.b). Where N is equal to the number $2^m$, which is the number of pixels in the x or y direction.

We have the following algorithm:

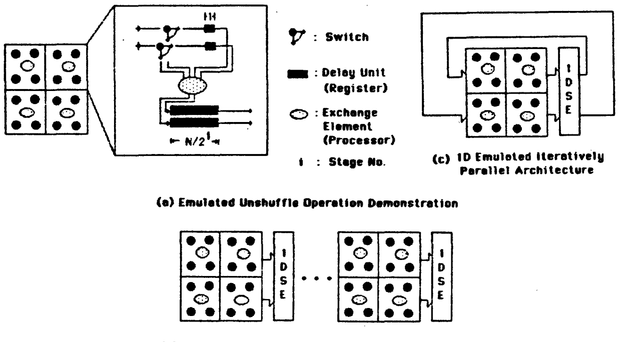
```
line Procedure CENTROID-b (A,$B_x$, $B_y$)
     //centroid computation in architecture (1.b)//
     for i = 1 to logN do
       begin
         for n1, n2 = 0 to N-1
           cobegin
             A[n1,n2] = ($SUM_2^i$A[n1,n2])
             //do summation with window size $2^i$ /
             $B_x$[n1,n2] = ($CENX_2^i B_x$[n1,n2,)
             //do center-x with window size $2^i$/,
             $B_y$[n1,n2] = ($CENY_2^i B_y$[n1,n2])
             //do center-y with window size $2^i$//
           coend
        end.
```

The role of 2DSE network (*i.e. unshuffle*) can be used to access data that have the window size doubled when stage advances. Thus the centroid computation can be mapped into a 2DSE architecture (*i.e.* Figure (1.a)) with time complexity O(logN). The mechanism $SUM_2$, $CENX_2$, and $CENY_2$ are exchange elements with widow size fixed as *two*.

The following procedure CENTROID-a is a parallel algorithm based on the architecture in Figure (1.a).

```
line Procedure CENTROID-a (A,$B_x$, $B_y$)
     //centroid computation in architecture (1.a)//
     for i = 1 to logN do
       begin
         for n1, n2 = 0 to N-1
           cobegin
             A[n1,n2] = {$U_2$($SUM_2$A[n1,n2])}
             //do summation and unshuffle the results//
             $B_x$[n1,n2] = {$U_2$($CENX_2${$B_x$[n1,n2]})}
             //do center-x and unshuffle the results//
             $B_y$[n1,n2] = {$U_2$($CENY_2${$B_y$[n1,n2]})}
             //do center-y and unshuffle the results//
           coend
        end.
```

In 1983, Liu proposed a way to emulate the 2DSE network so that he can implement the 2DFFT computation in VLSI [15]. His scheme takes advantage of the separability property formulated in the equation (2.1) of the 2D shuffle operation. The delay units, in his paper, are used to emulate the x direction of the shuffle operation. By modifying of his scheme, the emulated 2D architectures, which are shown in Figure 2, are obtained.



(c) 1D Emulated Iteratively Parallel Architecture

(a) Emulated Unshuffle Operation Demonstration

(b) 1D Emulated Iteratively Pipeline Architecture

Figure 2: 1D emulated Architectures

Part (a) of Figure 2 illustrates the architecture that emulates the 2D unshuffle operation. Part (b) and (c) show the iteratively pipeline and parallel fashion of the 1D emulated architecture.

Table 1 summarizes the results of the quadtree problems we solved by using 2DSE pipelined (*i.e* part (c) in Figure 1) and 2D emulated architecture (*i.e.* part (b) in Figure 2).

The detail algorithms for all these quadtree problems can be found in [16].

| | ARCHITECTURE 1<br>2DSE PIPELINE | | ARCHTECTURE 2<br>1D PIPELINE | |
|---|---|---|---|---|
| PROBLEM | Processor | Time | Processor | Time |
| Area | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Centroid | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Projection | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Signatures | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Eccentricity | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Contour Following | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Scroll Operations | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Perimeter Computation | O(N$^2$) | O(logN) | O(NlogN) | O(N) |
| Component Labeling | O(N$^2$) | O(log$^2$N) | O(Nlog$^2$N) | O(N) |
| Set Complementation | O(N$^2$) | O(1) | O(N) | O(N) |
| Set Union | O(N$^2$) | O(1) | O(N) | O(N) |
| Set Intersection | O(N$^2$) | O(1) | O(N) | O(N) |

Table 1 Summarized Results

The various functionally equivalent architectures enable us to best-fit to the requirements of real world constraints.

## 5. CONCLUSIONS

In this paper, parallel processing for quadtree problems based on 2DSE network is discussed. Various architectures such as 2D iteratively pipelined, iteratively parallel, 1D pipelined *et al* can be used to solve the quadtree problems with different performance constraints (*i.e.* processor numbers and time complexity). The results of the 2DSE approach toward the quadtree problems are listed in Table 1. It includes many primitive operations of computer vision. Besides the quadtree problems, the 2DSE network had been applied to efficiently solve a class of two dimensional problems such as 2DFFT, 2D Walsh/Hadamard transform, and 2D sorting [14]. Thus, the same architectures can be used for a broad range of applications. This makes the 2DSE approach more economical feasible.

## REFERENCE

1. M. Shneier, "Calculations of geometric properties using quadtrees," *Computer Graphics Image Processing 16,* pp. 296-302 (1981).

2. H. Samet, "Computing perimeters of images represented by quadtrees," *IEEE Trans. Pattern Analysis Machine Intelligence PAMI-3,* pp. 683-687 (1981).

3. C. R. Dyer, "Computing the Euler number of an image from its quadtree," *Computer Graphics Image Processing 13,* pp. 270-276 (1980).

4. H. Samet, "Distance transform for images represented by quadtrees," *IEEE Trans. Pattern Analysis Machine Intelligence PAMI-4,* pp. 298-303 (1982).

5. S. Ranade, "Use of quadtrees for edge enhancement," *IEEE Trans. System, Man, Cybernetics SMC-11,* pp. 370-373 (1981).

6. S. Ranade, A. Rosenfeld, and J. M. S. Prewitt, "Use of Quadtrees for Image Segmentation," *Computer Science TR-878, University of Maryland,* (1980 ).

7. S. Ranade and M. Shneier, "Using quadtrees to smooth images," *IEEE Trans. Systems, Man, Cybernetics SMC-11,* pp. 373-376 (1981).

8. S. Ranade, A. Rosenfeld, and H. Samet, "Shape approximation using quadtrees," *Pattern Recognition 15,* pp. 31-40 (1982).

9. H. Samet, "A Tutorial on Quadtree Research," pp. 212-223 in *Multiresolution Image Processing and Analysis,* ed. A. Rosenfeld,Springer-Verlag (1984).

10. T. Dubitzki, A. Wu, and A. Rosenfeld, "Parallel Region Property Computation by Active Quadtree Networks," *IEEE Trans. Pattern Analysis and Machine Intelligence, PAMI-3,* pp. 626-633 (1981).

11. N. Ahuja and S. Swamy, "Multiprocessor Pyramid Architectures for Bottom-Up Image Analysis," in *Multiresolution Image Processing and Analysis,* ed. A. Rosenfeld,Springer-Verlag (1984).

12. S. Tanimoto, "A pyramidal approach to parallel processing," *Proc. of the 10th Annual International Symposium on Computer Architecture,* pp. 372-378 (June 13-17 1983).

13. S. Tanimoto, "Sorting, Histogramming, and Other Statistical Operations on a Pyramid Machine.," pp. 132-145 in *Multiresolution Image Processing and Analysis,* ed. A. Rosenfeld,Springer-Verlag (1984).

14. G. Mei, "A Class of Two Dimensional Problems: VLSI Algorithms, Architectures, and Synthesis Tools," *M. S. Thesis Electrical and Computer Engineering Dept, NCSU.,* (December 1985 ).

15. W. Liu, "A new pipelined/parallel architecture for two dimensional Fast Fourier Transform," *Proceed. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* pp. 214-219 (Oct. 1983).

16. G. Mei and W. Liu , "Implementation of the Quadtree Problems on 2DSE Architectures," *Internal Report of the PEACE Project,* (Feb. 1986).

# A Feasibility Study and Simulation of the Circulating Context Multiprocessor (CCMP)

Clinton A. Staley    Steven E. Butner

**Department of Electrical & Computer Engineering**
**University of California – Santa Barbara**
**Santa Barbara, CA 93106**

## Abstract

This paper discusses the design and preliminary performance evaluation of the Circulating Context Multiprocessor (CCMP) -- a novel form of tightly coupled multiprocessor that combines significant positive features of modern computer structures and organizations while avoiding many of the negative ones. The CCMP consists of banks of special-purpose processors joined by full interconnection nets with queue-buffering. Each bank contains processors which perform one portion of the traditional Von Neumann cycle. Processes are represented by packets of information circulating among these banks and being modified by them. Many processes are active at once, being executed in an instruction-interleaved fashion. The structure inherently supports load balancing, high degrees of pipelining, efficient context switching, and modular reconfiguration. It appears to the software designer, however, to be simply a parallel computer with many Von Neumann processors sharing a memory.

**Keywords:** MIMD computer architecture, parallel computers, interleaved instruction streams.

## I. Introduction

The circulating context machine organization was first proposed in 1984 [1]. Since that time a great deal of feasibility study, simulation, and further development of the architecture has occurred. This paper reports on our current circulating context multi-processor (CCMP) architecture, its basis, strengths, weaknesses, and simulated performance. Additional material on the CCMP architecture can be found in [2].

The origins of CCMP are based on a collection of existing modern computer structures. The CCMP organization pulls together and adapts known successful concepts and structures from other designs while carefully avoiding most of the observed weaknesses of these concepts. Overall, it is an adaptation of ideas that make sense together. The list below capsulizes the design concepts and their origins.

- **Packets of executable context** --- from the dataflow (activation by availability) model [3]. The use of such packets facilitates streaming-type high throughput ... high-efficiency use of processing resources. Other problematical areas of dataflow, such as the need for a different model of computation, are avoided.

- **Von Neumann model** --- By utilizing this well-known computational model we do not require major investments for development in languages, compilers, linkers, debuggers and the like. Non-Von Neumann systems (such as data flow) will certainly require major development in these areas.

- **Queues with multiple server processors** --- from transaction systems (e.g. Pluribus [4]). The use of first-in, first-out queues as front-end staging for multiple identical servers (processing units) decouples hard synchronization, provides smoothed packet flow, and facilitates higher system availability.

- **Interleaved instruction streams** --- from modern pipelined systems [5,6,7]. By circulating process packets for many independent processes simultaneously, we can ensure (given sufficient workload) that processing units are kept busy, thus achieving very high utilization.

- **Redundancy management and reconfiguration** --- from fault-tolerant systems [8,9]. The presence of multiple copies of identical processing units within a system with full interconnection implemented among them allows immediate replacement of failing units --- giving a very robust and highly available organization.

The architectural concepts above are brought together to form the circulating context multiprocessor. The system makes a deliberate trade-off of increased instruction execution time for each individual process in exchange for overall system throughput and flexibility in adapting the machine to many differing types of parallel processing.

The motivations for the latency-for-throughput trade-off are several. In order to implement Von Neumann style computing, pipelining, and transaction-style dataflow in the same machine, a ring of processing stages is used. When parallel units are added, along with queueing to decouple and smooth packet flow, the trade-off becomes clear. The trade-off is necessary in order to get a clean, unidirectional processing loop structure with scalability, high-availability and other positive properties. The structure and connectivity of the resulting system inherently support "hot sparing", a type of redundancy that is among the quickest and easiest to reconfigure. The CCMP does not really treat "spares" as extra units --- instead such units are actively used to increase throughput. When failures occur and some units start to stand-in for downed companions, the system gracefully degrades in performance.

0190-3918/86/0000/0455 $01.00 © 1986 IEEE

By using queues to stage work packets for each set of processor servers, high utilization of pipelines is possible. The use of instruction stream interleaving guarantees that no dependencies will exist within pipelines and hence that no pipeline breaks will occur. Full pipelines imply maximized throughput. In a heavily pipelined system, throughput is directly linked to system clock rate. With the latency-for-throughput trade-off in effect, we will expand (within reason) the number of pipeline stages that are used to accomplish a given bit of processing if it helps in increasing the main clock rate.

The programming of CCMP is not vastly different from the programming of a conventional uni-processor. The CCMP offers flexible parallelism providing a high-throughput easily-sharable computing resource. For appropriately decomposable problems, the CCMP can take advantage of a significant degree of parallelism, particularly if the problem can be subdivided into co-operating but largely independent processes.

The machine does not have hard architectural size limits as are typical of SIMD machines with fixed vector lengths. The CCMP is first and foremost a throughput engine. The more it is loaded, the more it will deliver. Section 4 presents detailed simulation results which show linear throughput per process until roughly 80% efficiency is achieved. The CCMP can achieve streaming mode execution as is the goal of activation-by-availability (dataflow). However, the CCMP requires no changes to the model of computation.

On the negative side, the trade-off of per-process latency for system throughput implies that single stream performance will never be much better than that of a microcomputer. Thus, tasks will little parallelism will run relatively slowly. This roughly corresponds to non-vectorized segments of programs on SIMD machines. One important difference in the CCMP case, however, is that the unused portion of the machine (corresponding to the wasted vector register lengths in an SIMD machine) is available for use by other simultaneously-executing processes. The CCMP architecture allows full sharing of all resources so that users with low parallelism tasks get fixed, but relatively low performance while highly parallel ones can simultaneously get a significant percentage of system resources.

### Previous Work

As is apparent from the introduction, the ideas upon which the CCMP is based are not new. What is new is the combination and adaptation of those ideas in the present form ... culminating in a feasible, scalable, flexible multiprocessing system. There are several descriptions in the literature that document both proposed and constructed systems using some of the same ideas. In the area of interleaved instruction streams, the work of Miller [10] and of Kaminsky & Davidson [6] must be acknowledged. Other important related work is [5]. The positive effect of queued decoupling between various parts of a processor was discussed in [11].

The closest system to the CCMP is unquestionably the heterogeneous element processor (HEP) [7] [16]. This machine bears a resemblance to the CCMP although some of the CCMP's most important features are not present in the HEP. CCMP processes may visit any execution processor, a property that aids in systemwide load balancing. Interconnection networks in the CCMP are global and buffered, making the organization more homogeneous and thus both more flexible and more available. The CCMP memory is variably interleaved (as seen by the home module). This allows programs to create special memory layouts so that they can efficiently fit special meshes or algorithms to get added parallelism.

## II. Current CCMP Design

Here we discuss a refined CCMP design based on the concepts already discussed.

### Stage Division

The CCMP divides the Von Neumann cycle into portions, with a stage of processors handling each portion. The exact division used is a design parameter. Our current design uses three stages. Each of them is discussed below. Figure 1 provides an overview of the structure of the design, with the three stages linked by several interconnection networks through which packets of process context flow from stage to stage.

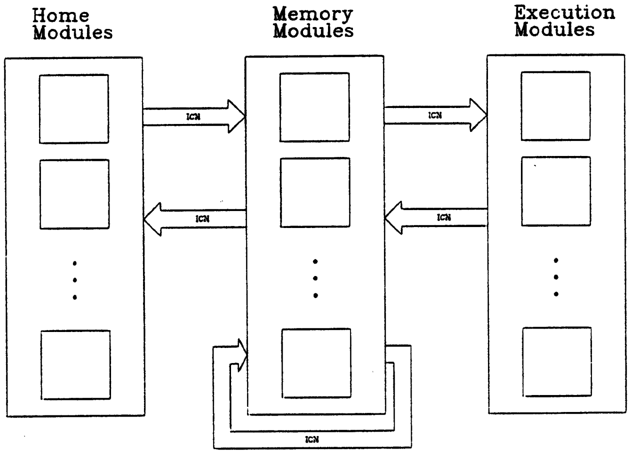


**Figure 1: Overview of the CCMP**

The first stage is the "home" stage which performs process specific functions such as updating the PC, delivering interrupts to a process, performing register fetch/store operations, and using segmentation registers to translate virtual addresses to physical addresses. The home processors in this stage are so called because a given process is always served by the same home processor -- it "returns home" to the same point in this stage. Note that we have departed from the pure CCMP concept here in that a good deal of the process' context remains in the home stage, instead of being transmitted about via the process packet. This was deemed necessary to avoid an unacceptable overhead in transmission costs, especially since most of this data is used only occasionally. The current instruction, the data on which we are immediately working, and a unique process ID number are the only contents of the process packet in our current design. Given the decision to hold some of the process context in a fixed stage, the necessity for that stage to have the "return home" property is obvious, since replicating each process' context in each processor of the stage would restrict the degree of parallelism possible, and result in data concurrency problems among the processors in the stage.

The next stage is the memory stage, "smart memory banks" capable of performing variable length instruction fetches, executing synchronization primitives like test-and-set and possibly maintaining private caches. Note that the memory stage to memory stage interconnect shown in figure 1 allows processes to visit several memory processors in sequence to fetch or store data without going to any other stage in the interim.

The memory stage constitutes a shared global memory accessible to all processes, but has no inherent interleaving policy. Memory segmentation is performed in the home stage, since performing it in the memory stage would require replication of segmentation information for each process in every memory processor. A few extra bits in the home processor segmentation registers can allow variable specification of interleave on a per-segment basis without any action on the part of the memory processors. The memory processors will use a standard physical address, the bits of which can be derived in any manner from the virtual address by the home processor to cause a segment to be low, high or even mid-order interleaved. This allows for more flexibility in load balancing of the memories and is essential to the high availability schemes discussed later. Variable interleaving can cause complex overlap problems between segments in physical memory -- it is assumed that any operating system for the CCMP will either plan for and prevent such overlaps (unless they are desired), or simply use a standard interleave policy for all segments.

The final stage in the design is the execution stage, which performs all arithmetic or logical operations demanded by the instructions. Each processor in the stage is identical -- we chose not to specialize them to avoid problems with tasks that make skewed use of operations and might load only a subset of the processors. Any process, then, may visit any execution processor to have an operation performed on data that it has fetched. Processes are sent to non-overloaded processors in a round robin fashion. Overloaded processors receive no new processes until they become free; then they join the round-robin.

As an example to clarify the above, we trace a process packet through a typical instruction. The instruction begins with the packet at its home processor, where its PC is fetched and translated to a physical address that is placed in the process packet. The packet then goes to the appropriate memory processor where the contents of the instruction are fetched. It returns to the home processor, where any register fetches required by the instruction are performed. If memory data fetches are required, the home processor determines the physical addresses involved and the process packet returns to the memory stage, travelling to the memory(ies) where the data to be fetched resides. Data stores can be done at this point also. For instance, a memory-to-memory data transfer can be done without returning to the home. If arithmetic or logical operations are required, the process travels from the memory stage to the execution stage. If memory stores are required after an execution processor completes the operation, the process returns to the memory stage, and then finally returns home where any register stores needed are performed and the next instruction begins.

## Interconnection Networks

We have said nothing in detail thus far about the interconnection networks. ~~There are a lot of them as the reader~~ can see from Figure 1. This set of networks was derived from an examination of the likely stage-to-stage transfers in a typical instruction set. As one would expect, the design of these interconnects is critical to the success of the CCMP. For large scale parallelism --- more than around 40 processors per stage --- we currently intend to use a single stage cube connection (see [14]). We feel that the cube connection is superior to the shuffle exchange or one of its relatives for this application. The cube connection allows for fewer transfers between nodes on the average. This is generally offset by the cost of a larger number of interconnections per node. In our case, however, we are designing to optimize the use of the hardware, coming as close as feasible to saturating the network with a continuous flow of data, rather than attempting to guarantee some short delivery time to occasional individual messages. For this discussion, let us assume that the clock speeds of sending and receiving modules are the same as that of the interconnection lines, and that bandwidth is measured in bits per clock. Assuming full utilization of bandwidth, (with $n$ the bandwidth per incoming line and $p$ the degree of parallelism) our choice is between shuffle nodes with internode bandwidths of $n \log_2 p$ on each of two internode lines and cube nodes with internode bandwidths of $n$ on each of $\log_2 p$ internode lines. We obtain a 50% reduction in internode bandwidth with the cube interconnect, paying only the price of dividing the internode lines more finely. This comparison is discussed in more detail in [12].

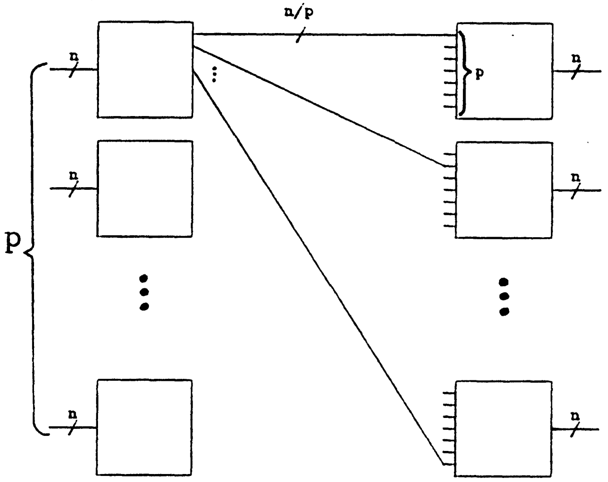


**Figure 2: Modified Crossbar Interconnection Network**

Let us assume that the transmission time of an individual packet is not at issue, and concern ourselves only with meeting the overall bandwidth requirements of the stage-to-stage interconnection with a minimum of physical interconnect. Assuming a packet has $n$ bits and that a fully utilized module delivers one packet per clock, we see that the actual bandwidth required is only $np$ and that both of the above networks use a factor of $\log_2 p$ too much bandwidth. We can remove this factor by using a modified crossbar interconnect as shown in Figure 2. Each sending node has an outgoing bandwidth divided evenly among all receiving nodes (simultaneous transmission is assumed). The same bandwidth can be divided more finely for higher degrees of parallelism, serializing the transmission of individual processes to accommodate the lower individual bandwidths. The cost of such division is that each packet is delivered more slowly, and somewhat more queueing is required to buffer packets prior to delivery. More complex switching is also required in the sending and receiving units. The tradeoffs between this approach and the cube interconnect are shown in the complexity analyses of table 1.

Two terms that may require elaboration are the $O(p^2)$ and $O(\frac{1}{p})$ figures for number of processes and per-process throughput in the bipartite (crossbar) case. Although the overall throughput of the modified crossbar is equal to that of the cube interconnect, the speed of transmission through the interconnect drops for an individual process in inverse

**Table 1: Complexity Analysis of Candidate ICNs for the CCMP**

| Resource | Modified Crossbar Interconnect | Cube Interconnect |
|---|---|---|
| Throughput | $O(p)$ | $O(p)$ |
| Interconnect Bandwidth | $O(p)$ | $O(p\log_2 p)$ |
| Number of Processes | $O(p^2)$ | $O(p\log_2 p)$ |
| Per Process Throughput | $O(\frac{1}{p})$ | $O(\frac{1}{\log_2 p})$ |
| Number of Queues | $O(p^2)$ | $O(p\log_2 p)$ |
| Interconnect Node Complexity | $O(p)$ | $O(log_2 p)$ |

proportion to the degree of parallelism. This is due to the increased serialization needed to maintain the same bandwidth as the number of processors increases. $O(p)$ processes per sending processor will be tied up in transmission across the net. Thus, $O(p^2)$ processes will be needed to maintain an $O(p)$ increase in throughput. The constant factors in the $p^2$ terms for number of processes and queue sizes are expected to be small enough to make the modified crossbar connection preferred for low orders of parallelism. Note that number of processes, or more precisely, degree of parallelism is viewed as a resource here, and the modified crossbar scheme requires a higher order of parallelism to deliver the same performance with all other factors held constant. A number of changes in parameters for the CCMP will result in higher consumption of parallelism but no significant drop in throughput. This is true, for instance, when the number of pipeline stages in the various processors is increased.

**Software and System Details**

The current synchronization primitive in the CCMP simulator is a standard test and set instruction. We expect to change this, however, to an atomic swap operation. This requires about the same extra logic in the memory processors and is more powerful. In particular, it can be made to emulate the full/empty flags that were attached to registers by the HEP [7] designers and which work so elegantly in that architecture. All that is required is a designated "empty" value. The two pieces of code in table 2 demonstrate one use of this. A group of processes running the sending segment can synchronously pass one value at a time through memory location $x$ to a group of processes running the receiving segment.

**Table 2: Synchronization via Atomic Swap**

| Sending Segment | Receiving Segment |
|---|---|
| while data to send {<br>   place data in d;<br>   while d ≠ "empty" value {<br>      atomic swap d and x;<br>   }<br>} | while data to be received {<br>   place "empty" value in d;<br>   while d equals "empty" {<br>      atomic swap d and x;<br>   }<br>} |

Control of process creation and segmentation is through special instructions. There is a set of (privileged) instructions to allow one process to set another's segmentation registers. There is also a "*create*" instruction that allows one process to create another by instructing the new process' home processor to start a packet circulating for that process and to set its PC to a designated value. Home processors keep track of whether a process is currently active (has a packet circulating) or not. If a *create* is done on an already circulating process, the *create* becomes an interrupt, which is implemented by altering the affected process' PC so that it will go elsewhere next time it comes around. This same mechanism allows an I/O device to interrupt a given process.

The I/O system is memory mapped, so connections between it and the CCMP hardware are via the memory processors. Low bandwidth devices are linked to memory processors in a conventional manner, with devices evenly spread across processors to prevent any one processor from being overloaded by process packets attempting to fetch data from devices. Disk or other mass storage devices are somewhat more problematic, since they are required to deliver block transfers in various interleaved fashions to the memory processors. To accommodate this, we place another interconnection network between disks and memories, with buffering at the disks. This allows each disk to deliver data to and from each memory and permits a block load, even to a low-order interleaved segment, to be delivered from a single disk.

A simple approach to delivering interrupts to a process when an I/O operation is complete would be to run a bus through the home processors to allow devices to signal for interrupts. This might become a bottleneck in a highly parallel system, however. As mentioned above, internal interrupts may be delivered to a process by means of a *create* instruction. A more general approach to I/O interrupts, then, would be to allow an I/O device to signal a memory processor to create a dummy packet with a *create* instruction in it, destined for the desired process. The dummy packet would be given a special process ID number that would cause it to vanish after completing its task. In this way, the capacity to deliver interrupts would increase proportionally to the parallelism.

We are currently looking into the design of a distributed microcode system, in which each type of processor would have its own set of micro-operations, and would fetch microinstructions from a private microinstruction memory which would tell it what to do for a packet with a given instruction at a given point in its completion (instruction and state of completion are part of the process' packet). This would allow a more flexible instruction set and simpler hardware, as with other microcoded machines. Dynamic downloading of microcode would, of course, also be a possibility.

One somewhat tricky problem with the CCMP is the possibility of what we refer to as "ring deadlock", a situation in which a ring of modules and interconnect lines in the architecture have become mutually deadlocked. This situation will not abate by itself, and will quickly deadlock the remaining processes as they attempt to enter modules on the deadlocked ring. The solution to this problem is fairly straightforward. As figure 1 shows, the topology of the interconnections is such that the memory processors have a number of choices of where to send packets, and any deadlock loop must include a memory processor. Any time a memory processor becomes blocked, it sends the offending packet to another memory processor with an indication that the packet has been deferred. The receiving processor will try to send the process to its proper destination, or failing that will defer it once again. To obtain ring deadlock in this case it would be necessary to fill all the memory-to-memory queues. This is arranged to be more space than there can be process packets so deadlock becomes impossible. The existence of ring deadlock was predicted and then confirmed by simulation. The solution described has also been tested in simulation and found to be highly effective -- keeping the machine running smoothly when it would quickly deadlock without the avoidance mechanism.

## III. Simulation Results

We have developed a detailed simulation written in 'C' and running on a SUN work-station (or VAX) under UNIX 4.2. The simulation is at the register-transfer level, accounting for single-clock events. We believe that our simulations are conservative and very realistic, modeling virtually all significant aspects of the CCMP.

Several configurations of a particular CCMP machine have been modeled. Because of the detailed nature of our simulator, however, we have not extensively studied *large* configurations. We have instead concentrated on investigating relatively small systems, particularly studying the behavior under heavy process and memory loads. The system loads well (according to our studies so far) and we have reason to expect that because of its load-balanced, bottleneck-free structure, the CCMP will scale-up to large configurations well.

Our simulations assume representative levels of pipelining such as would be achievable and typical in conventional NMOS or CMOS VLSI (refer to table 3). We have assumed 18-25 MHz clock rates† (depending on stage type). Instruction processing may involve as many as several hundred single-clock pipeline stages including queue waiting times. This still would provide per-process instruction speeds of around 100,000 operations per second.

**Table 3: Pipeline Stages per Module Type**
(queue lengths excluded)

| Module Type | Simulated Stages |
|---|---|
| Home module | 16 |
| Memory module | 8 |
| Execution unit | 32 |

In the simulation results that follow, we have modeled a 4/4/2 CCMP system, i.e. one with 4 home modules, 4 memories, and 2 execution units. Note that because of our extensive use of queues these modules need not be synchronized, nor must they execute at similar clock rates. If the clock rates of different banks of resources are not roughly equal, then it will be necessary to balance the overall system by selecting the number of units in each resource bank so that their average throughputs match.

### Matrix Multiply (Saturation Load) — MMSAT

The first test program written for our simulator was a matrix multiply. In this program, we use one process for each entry in the output matrix. Since the matrix size chosen was 15×15, we used 225 processes. In this problem the parallelism is immediate, predictable, and constant. This is shown by the processes versus time plot (figure 3); the corresponding instructions per second versus process count plot (figure 4) shows that linear throughput is delivered as a function of the number of concurrent processes. Instead of terminating execution upon completion of the matrix multiply, we simply continue computing the elements over and over so that the program can be used as a type of system load.

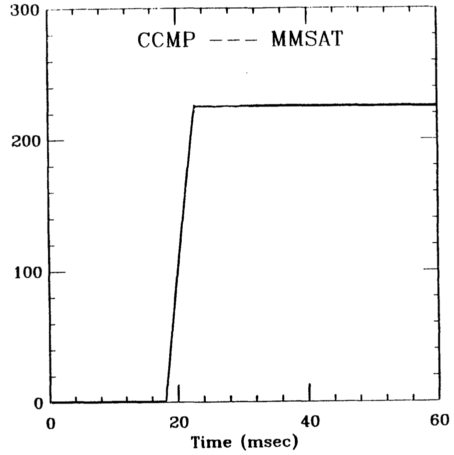


**Figure 3: MMSAT — Processes vs. Time**

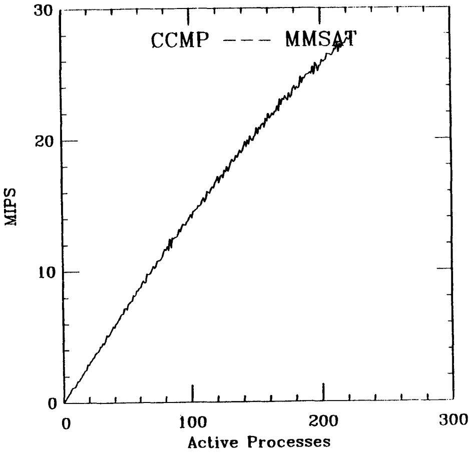


**Figure 4: MMSAT — Throughput vs. Offered Load**

### Recursive Quick Sort — QSORT

The quicksort algorithm [13] has also been coded and run on our simulator. Three distinct experiments were performed with qsort. In the first one, the pivot element was chosen arbitrarily (the first element in the array) as specified in Knuth's description. Figure 5 gives the parallel behavior for a sort of 10,000 integer elements in memory. Note that the maximum parallelism achieved is in the neighborhood of 30 processes. The corresponding throughput graph (figure 6) gives the observed MIPS rate, again very linear with respect to process count.

The second experiment with qsort was an attempt to intelligently choose better pivot points, ones that tend to divide the remaining elements into more or less equal groups. To do this, a simple pivot-choosing loop was added. The resulting behavior is shown in Figures 7 and 8. In this case parallelism of over 200 processes was observed, but at

† Note that we may freely trade additional pipeline stages in exchange for higher clock rates since an overall per-process latency trade-off has already been made.

the cost of a long-running initial pivot choice. The algorithm used for choosing the pivot point samples every $32^{nd}$ data element throughout the entire array. For the full 10,000 element array (the first pivot point chosen), this requires the inspection of over 300 items. Clearly, we could adjust this number and get vastly improved behavior with a much smaller number of probes.

The final experiment with qsort involved running the previous case in the presence of the matrix multiply saturation load. This was done to see how the two parallel programs interacted (i.e. whether they interfered with one another) on the machine. As expected, the CCMP allowed smooth sharing of resources. Neither program significantly impacted the resource utilization of the other. Figures 9 and 10 show the observed behavior from the mmsat + qsort experiment.

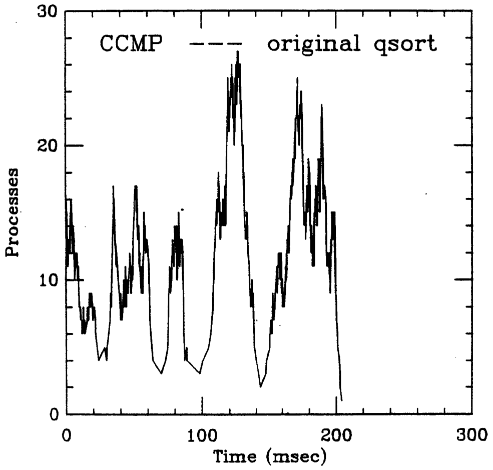


**Figure 5: QSORT — Processes vs. Time**

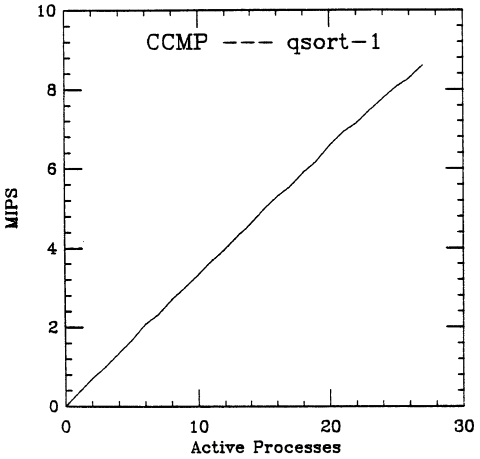


**Figure 6: QSORT — Throughput vs. Offered Load**

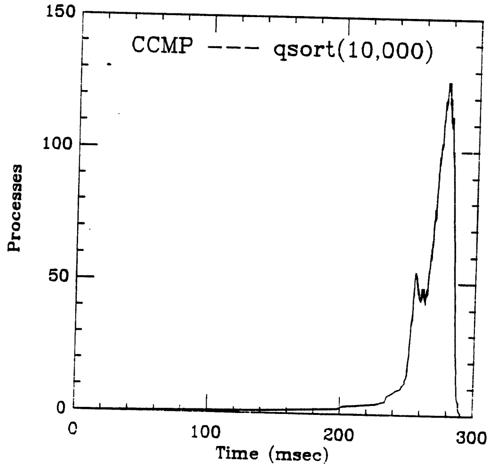


**Figure 7: QSORT w/ Pivot Selection — Processes vs. Time**

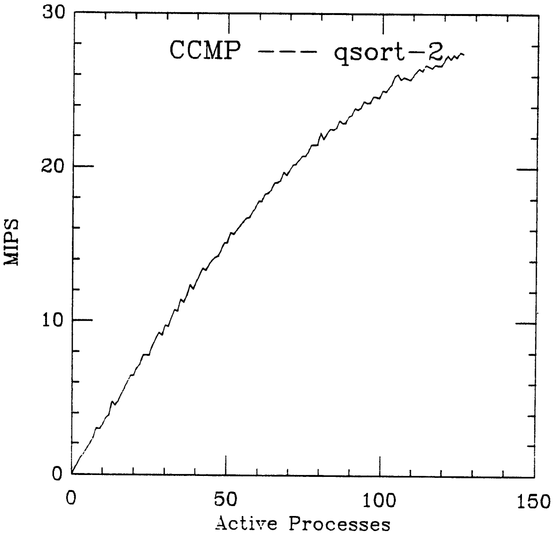


**Figure 8: QSORT w/ Pivot Selection — Throughput vs. Load**

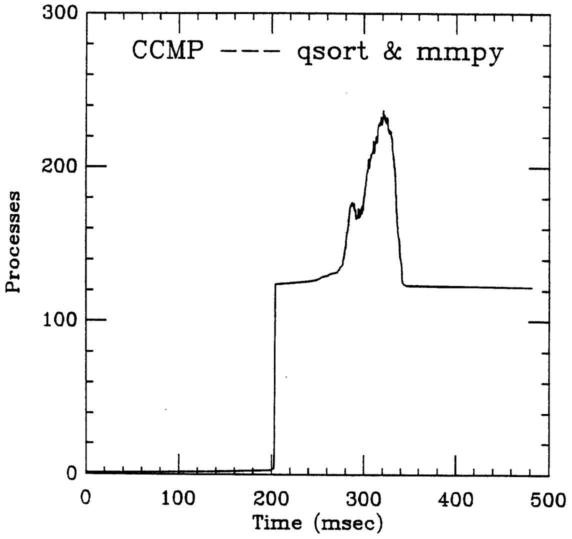


**Figure 9: QSORT w/ MMSAT — Processes vs. Time**

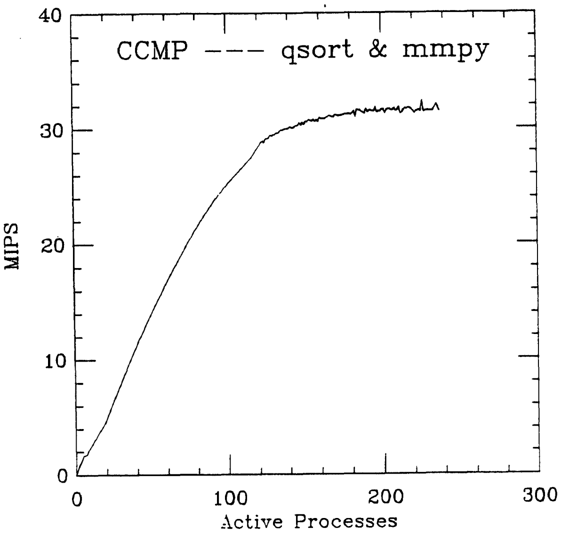


**Figure 10: QSORT w/ MMSAT — Throughput vs. Offered Load**

**Relaxation Algorithm — RL**

This program implements a simple 2-dimensional relaxation algorithm. We used a fast process generation technique, with each process creating four others, to rapidly build parallelism. Even though the program begins with only a single process, it is clear from figure 11 that after a segment set-up time, included for ease of coding, the full parallelism is acheived almost instantaneously. The simulation uses 256 cells (each with its own process) without synchronization. Each cell accesses the data elements associated with its neighboring cells, i.e. neighboring cells' data areas overlap. Like the matrix multiply (saturation) test, this program never terminates. It is the fully-loaded behavior of the CCMP that we wish to study. Figures 11 and 12 present the observed performance. The throughput graph appears jagged because of transient effects during the rapid increase in parallelism. The final dip in performance is caused by transient contentions for the same instructions by many newly-created processes. As can be seen from the two figures, this cluster of processes rapidly spreads out to permit 77% utilization of the machine (maximum performance achievable in this configuration is 36.36 MIPS). The relaxation algorithm provides so much parallelism that very little time is spent (hence, relatively few sample points) with less than the full number of processes.

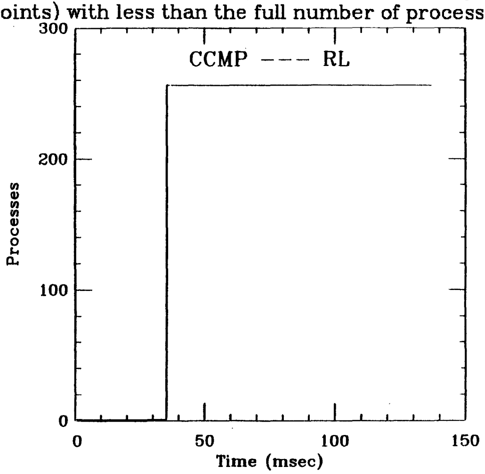


**Figure 11: RL — Processes vs. Time**

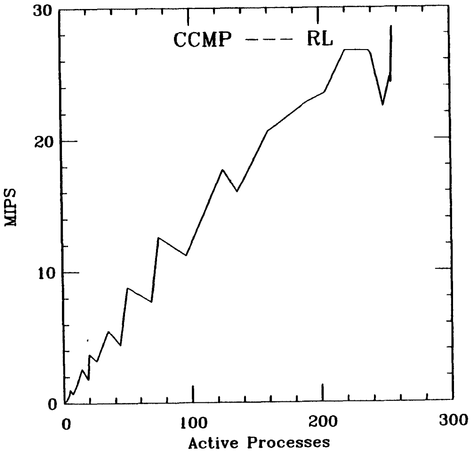


**Figure 12: RL — Throughput vs. Offered Load**

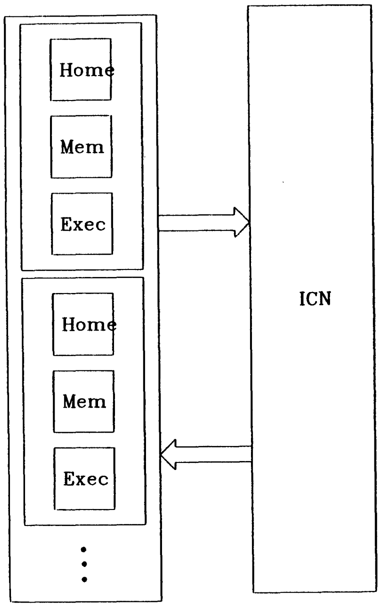


**Figure 13: Grouped Module CCMP Configuration (not yet simulated)**

## IV. Ongoing Work

We are currently engaged in simulation of larger models of the CCMP. Indications have so far shown no serious problems with expanding the degree of parallelism. One difficulty is that small, highly iterative loops in a low-order interleaved segment should be unrolled to span the entire set of memory processors to avoid overloading only a few processors with instruction fetches.

We are currently planning a major re-structuring of the simulator to allow us to test the effects of different instruction sets, varying numbers of registers, various queue designs, and other CCMP configurations, especially that shown in figure 13, where the various stages have been stacked one atop the other and share a common, higher-bandwidth interconnection network.

Implementation of high availability for the CCMP is another area for further research. Coding can be added to the process packets to allow detection of faults in networks or processors. Once a faulty processor has been detected it can be set offline while the others continue functioning. In the case of the execution units this is trivial since they are all identical. A faulty home can be offlined by not using the process ID's that report to it. Offlining a memory processor can be accomplished by using the flexible interleaving capability described earlier to interleave segments across subsets of the memory processors, avoiding the faulty one(s). Offlining a network link in the case of the crossbar arrangment can be accomplished by offlining the processors at either end of it or by using the deferral mechanism already installed for deadlock avoidance. All stages can send to the memory stage, and a bad link can be circumvented by sending the process, with an indication that it has been deferred, to an accessible memory processor which can then send the packet to the desired location.

In the case of a cube interconnect, there are well-established means for avoiding a bad link. A bad node, however, would require the offlining of the two processors to which it is connected.

## V. Conclusions

We have presented a novel multiprocessor architecture and the results of a recent study of its feasibility and performance. The CCMP machine is a general MIMD computer that trades individual process performance for high overall system throughput. For maximum speed on a given (parallel) problem, the goal is to express the overall computation in terms of many (perhaps thousands) of co-operating, but largely independent, processes.

We have carefully modeled a particular instruction set at essentially the register-transfer level and studied the performance on several important and typical problems. The system can deliver 80% efficiency quite routinely (i.e. without significant optimization) and can get near-100% utilization on selected problems.

Since the system in pipelined and uses instruction stream interleaving to preclude pipeline breaks, pipelines can be longer than typical. This can allow higher than typical clock rates, since large-delay circuitry can be broken down into multiple simpler (i.e. faster) pipeline stages. The CCMP makes sense for VLSI, particularly as technology moves toward wafer-level interconnect [15] and multi-chip hybrids.

Another strength of the architecture (which has not been stressed in this report) is its excellent support for error-detection, reconfiguration, and re-try. With adjustments made to the home modules (primarily in the number and type of registers) and with the addition of error detection (at either the register-transfer, single module, or single instruction loop level), the CCMP could be made to support fault-tolerant applications.

## VI. References

[1] S. E. Butner, "The Circulating Context Multiprocessor, An Architecture for Reliable Systems," International Symposium on Mini & Microcomputers (ISMM), June 1984, pp. 50-52.

[2] S. E. Butner & C. A. Staley, "A RISC Multiprocessor Based on Circulating Context," IEEE Phoenix Conference on Computers and Communications, March 1986.

[3] J. Dennis, "Data Flow Supercomputers," **IEEE Computer**, Nov 1980.

[4] D. Katsuki et. al., "Pluribus - An Operational Fault-Tolerant Multiprocessor," **Proceedings of the IEEE**, vol. 66 # 10, pp. 1146-1159, Oct 1978.

[5] M. J. Flynn and A. Podvin, "Shared Resource Multiprocessing," **IEEE Computer**, March 1972, pp. 20-28.

[6] W. Kaminsky & E. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," **IEEE Computer**, pp. 66-76, Dec 1979.

[7] Denelcor Inc., *Heterogeneous Element Processor: Principles of Operation*, April, 1981.

[8] J. Sklaroff, "Redundancy Management Technique for Space Shuttle Computers," **IBM Journal for Research & Development**, 20-27, Jan 1976.

[9] J. Stiffler, "Architectural Design for Near-100% Fault Coverage," IEEE Fault-Tolerant Computing Symposium (FTCS-6), 134-137, 1976.

[10] E. F. Miller, "A Multiple-Stream Registerless Shared Resource Processor," **IEEE Transactions on Computers**, C-23 #3, Mar 1974, 277-285.

[11] James E. Smith, "Decoupled Access/Execute Computer Architectures," **ACM Transactions on Computer Systems**, Vol 2 #4, Nov 1984, 289-308.

[12] P. Y. Chen, D. H. Lawrie, P-C. Yew, D. A. Padua, "Interconnection Networks Using Shuffles," **IEEE Computer**, vol 14 #12, Dec. 1981, p. 55-63.

[13] D. E. Knuth, **The Art of Computer Programming**, vol. 3, Addison-Wesley, 1973.

[14] H. Sullivan & T. Bashkow, "A Large Scale Homogeneous Machine I," IEEE Symposium on Computer Architecture, 1977, pp. 105-117.

[15] B. Donlan, et.al., "The Wafer Transmission Module," VLSI Systems Design, vol VII, no. 1, Jan 1986, pp. 54-58.

[16] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," Proceedings of the 1978 Conference on Parallel Processing, pp. 6-8, 1978.

# Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study

Thomas J. LeBlanc
Computer Science Department
University of Rochester

**Abstract** - The BBN Butterfly Parallel Processor can support a user model of computation based on either shared memory or message-passing. In this paper we describe the results of our experiments with the message-passing model. The goal of the experiments was to analyze the tradeoffs between the shared memory model, as exemplified by the BBN Uniform System package, and a simple message-passing model. We compare the two models with respect to performance, scalability, and ease of programming. We conclude that the particular model of computation used is less important than how well it is matched to the application.

## Introduction

The hardware architecture of the BBN Butterfly™ Parallel Processor supports message-passing between processors using an FFT network of 4x4 switch elements. The memory architecture, implemented by the operating system in conjunction with a micro-coded co-processor, provides the illusion of shared memory. As yet, there are no additional software levels that distort this illusion, so the user-level view of the architecture is that of a shared memory multiprocessor. This paper describes a series of experiments designed to explore the ramifications of a user-level view of the Butterfly architecture based on message-passing.

Both shared memory and message-passing models have been advocated for parallel computation. Shared memory is usually found in tightly-coupled architectures that support remote memory operations in hardware or firmware. Message-passing is typically employed in loosely-coupled systems (*e.g.*, local-area networks). Early work on the Butterfly, the Voice Funnel application in particular, used message-passing. Recently, Butterfly applications, including finite element analysis and computer vision algorithms, have assumed the shared memory model of computation. The only programming environment available on the Butterfly that masks the low-level details from the programmer is the Uniform System package from BBN, which implements a shared memory model. Programmers find it easier to use the Uniform System than to build the program from scratch, *regardless* of how well the application fits the shared memory model.

Before proceeding with our implementation of an environment supporting message-passing [3], we conducted a series of experiments using Gaussian elimination as a sample application. The goal of the experiments was to explore the tradeoffs between the shared memory model, as exemplified by the Uniform System package, and a simple message-passing model based on asynchronous send and receive. Important factors to be considered were ease of programming, performance, and scalability. In this paper, we present the results of our experiments.

## The BBN Butterfly Parallel Processor

The Butterfly multiprocessor at the University of Rochester consists of 128 processing nodes connected by a switching network. Each processor is an 8 MHz MC68000 with 24 bit virtual addresses. A 2901-based bit-slice co-processor interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. All of the memory in the system resides on individual nodes, but any processor can address any memory through the switch. A remote memory reference (read) takes about 3.75 *us.*, roughly 6 times as long as a local reference.

Each switch node in the switching network is a 4-input, 4-output crossbar switch with a bandwidth of 32 megabits/sec. An N processor system uses $(N \log_4 N)/4$ switches arranged in $\log_4 N$ columns. The 128-node Butterfly contains 256 switch nodes; the extra switch capacity is used to provide an alternate communication path between all processors for reliability and improved efficiency. The alternate paths can be enabled or disabled by the user, making it possible to indirectly measure the effect of switch contention.

## Case Study: Gaussian Elimination

The sample application chosen for the experiments was the solution of a set of linear equations using Gaussian elimination (without pivoting). The reason for this choice was that several experiments using shared memory to implement Gaussian elimination had been performed at BBN Laboratories [2,5]. Our experiments with message-passing were designed for comparison with their results. In addition, Gaussian elimination is representative of a large class of problems in finite element analysis. While it is fairly easy to develop parallel algorithms for Gaussian elimination, the problem does have important synchronization constraints.

In solving a set of linear equations using Gaussian elimination, the coefficient matrix M is diagonalized, producing a modified vector of unknowns, and the unknowns are determined using backsubstitution. (Since backsubstitution is a small percentage of the total time required to solve the equations, it is not performed in any of the experiments.) To eliminate an entry M[i,j], we replace row M[i] with M[i] - (M[j] * M[i,j]/M[j,j]), where M[j] is known as the pivot row. However, this operation cannot be performed until row M[j] has stabilized, *i.e.*, $M[j,k] = 0, \forall k < j$. In addition, all previous entries in row i must already be eliminated, *i.e.*, $M[i,k] = 0, \forall k < j$. These two synchronization constraints limit the amount of parallelism that we can expect to achieve.

Floating point arithmetic is required to solve a set of linear equations. Unfortunately, our Butterfly does not have floating point hardware. All floating point arithmetic is performed by costly subroutines. This makes it difficult to analyze the communication costs of an application because the execution time is dominated by floating point software. To alleviate this problem, our experiments were performed with *simulated floating point* arithmetic, using the same approach as that used in the shared memory implementation [2]. All floating point variables were replaced with integer variables, and addition and subtraction were used in place of multiplication and division. Such a computation does not give the correct answers, but does accurately simulate the performance of the computation on a Butterfly with floating point hardware. Each experiment was performed using software floating point to ensure the correctness of the results, but all reported performance figures refer to the results of experiments using simulated floating point. By using simulated floating point, we not only can make predictions about the performance of the Butterfly with floating point hardware, we also reduce the execution time of the computation and thereby increase the significance of communication costs.

0190-3918/86/0000/0463 $01.00 © 1986 IEEE

### Shared Memory Implementation

The shared memory version of Gaussian elimination was implemented at BBN Laboratories using the Uniform System (US) package [1]. US provides a set of calls to create a global shared memory, accessible to all US processes. Within the shared memory, pointers may be shared between processors. US also creates a manager process on each processor that is responsible for allocating the processor to a series of tasks, light-weight processes that operate on the shared memory. Usually, a task is some small procedure to be applied to a subset of the shared memory. A task, therefore, can be represented as simply an index, or a range of indices, into the shared memory and an operation to be performed on that memory. Atomic operations in micro-code are used to efficiently allocate tasks to processors.

The strength of the Uniform System is that it supports a user-level view of the architecture consisting of light-weight tasks and shared memory. To reduce memory contention, US encourages a globally-shared memory and the scattering of data uniformly throughout the machine. (This may even include putting data in memory associated with a processor that is not involved in the computation.) To reduce process management overhead, only one real process is allocated per node; all tasks are light-weight.

Several experiments were performed at BBN Laboratories to see how well applications that use US, including Gaussian elimination, perform on large Butterfly configurations [2, 5]. In these experiments, the problem matrix was uniformly distributed throughout available memory. A task was created to eliminate a single entry in the matrix, M[i,j]. Both row M[i] and row M[j] were transferred to local memory before performing the computation, rather than perform a remote reference for each individual entry in a row. (Block transfers amortize overhead associated with remote references and can significantly improve performance.) Since M[i] is modified by the operation, an additional transfer of the modified row back to the shared memory location for M[i] was also necessary.

The main result of these experiments was to show that the 128-node Butterfly could achieve nearly linear speedup when performing Gaussian elimination using US. Both switch contention and memory contention were shown to be insignificant (2% and 3%, respectively). However, those results used an early version of the program which had not been tuned. An optimized version was later developed, which greatly reduced the time spent in the inner loop of the program. Tuning the inner loop not only lowers the total time required for the computation, but increases both memory and switch contention during the computation. Therefore, we performed several experiments to examine the effects of contention on the optimized shared memory implementation.

The first experiment we performed was designed to examine the effect of memory contention on the performance of the optimized shared memory implementation. In the US implementation, all memories in the system are used to store the problem matrix, even when only a few processors are involved in the computation. This has the effect of improving the overall performance by decreasing memory contention. Since the message-passing implementation can not take advantage of memories in unused processors, it is important to quantify this effect before comparing the two implementations.

In order to determine the effect extra memories have on the performance of a computation (and to indirectly measure memory contention), the optimized shared memory implementation of Gaussian elimination was executed on various numbers of processors and memories. The test results show that the extra memories can have a substantial impact on performance, which in turn suggests that memory contention can be a significant factor. On a problem matrix of size 400x400, a Butterfly configuration with 4 columns of switches, 16 processors, and 96 memories performed 15% better than 16 processors and 16 memories. On a problem matrix of size 200x200 and a Butterfly configuration with 2 columns of switches, 4 processors and 16 memories performed 30% better than 4 processors and 4 memories. The greatest effect occurs when roughly 1/4 to 1/2 of the total number of processors are in use. When a larger fraction of processors are performing computation, most of the memory is already in use, *i.e.*, there is no other extra memory. When too few processors are used, they are insufficient to generate enough load on the memory to make memory contention significant.

The second experiment we performed was designed to examine the effect of switch contention on the performance of the optimized shared memory implementation. The 128-node machine has a switch configuration capable of supporting 256 nodes. The excess switch capacity increases the total amount of potential bandwidth in the switch, which will improve the overall performance of a computation whenever there are enough processors involved to cause switch contention.

The results of this experiment show that the effect of alternate switch paths can be dramatic (as a percentage of total execution time) when a large number of processors are in use. A 35% improvement in execution speed was attained using alternate paths for 94 processors on a problem of size 400x400; 64 processors achieved a speedup of more than 20% on the same problem using alternate paths. Since the shared memory implementation does not exploit locality of data (two problem rows must be copied into local memory before an operation can proceed), it is not surprising that the shared memory implementation introduces significant switch contention once the inner loop has been tuned to the point where the program is communication intensive.

### Message-Passing Implementation

In order to explore fully the potential of message-passing in the context of the case study, it was important to provide a general-purpose message-passing system. Communication primitives too closely associated with the application would demonstrate little about the message-passing model. Therefore, the following primitives were provided:

**Send**(destination, buffer address, byte count);
**Broadcast**(buffer address, byte count);
**Receive**(source) : buffer address:

Communication is asynchronous between a sender and receiver. **Send** is nonblocking; the sender may continue to execute before the destination receives the message. However, the implementation may block the sending process for a short period of time necessary to buffer the message. **Receive** will block until a message from the source arrives. A wildcard value is available so that messages from any source may be received. **Broadcast** is included because it can be implemented more efficiently by the system than if it is simulated using point-to-point communication. These primitives are very general since minor variants have been used as the basis for communication in distributed operating systems.

Several versions of the message-passing system were implemented in an attempt to explore efficient implementation techniques. In the first, and simplest, implementation, an object was created for each message, using the object management facilities of Chrysalis, the Butterfly operating system [4]. Each time a broadcast took place, the message system would create a remote object for each recipient and copy the message into the object. This approach yielded a very clean implementation, but the overhead of using the object management system for each message was too great. In addition, remote object creation requires the cooperation of a remote daemon process, which introduces contention on the remote processor. Later versions used preallocated objects for buffers and placed responsibility for copying a message with the receiver rather than the sender. The final implementation has the following properties: (1) buffers are preallocated on each processor and are eventually reused, (2) each sender copies the message from local memory into a local buffer object and updates the local ring buffer pointers, (3) each broadcast recipient copies a message from a remote buffer into local memory when a corresponding **Receive** is issued, and decrements a use count in the remote buffer using an atomic operation, and (4) synchronization for access to buffers uses primitive atomic operations implemented in micro-code.

In the implementation of Gaussian elimination, a coordinator process is responsible for creating worker processes on each processor. All workers initialize the local message handler, which requires global synchronization, and then create a local partition of the problem. When N processors are used, each processor P is assigned the task of diagonalizing all rows R for which R mod N = P. To do so, each processor must request each row of the problem matrix in sequence and use it to eliminate entries in the corresponding column of the local partition. A percentage of these requests, 1/P, will be satisfied locally and, therefore, do not require a message. Local rows are broadcast to the other processors when they have been diagonalized. When all local rows are completely diagonalized, the worker process signals the coordinator, which is responsible for termination of the computation.

### Implementation Comparison

In this section, we compare the two implementations of Gaussian elimination with respect to performance, scalability, and ease of programming. Where possible, quirks in the implementation of a model are ignored, in order to concentrate on the methodology suggested by the model.

Figure 1 shows the performance of the two implementations on a problem matrix of size 800x800. With four processors, the message-passing implementation is about 30% faster than the shared memory implementation. The relative performance is consistent across different problem sizes and, in each case, the message-passing implementation is more efficient. The disparity decreases as additional processors are used until the "knee" in the performance curve of the message-passing system is reached.

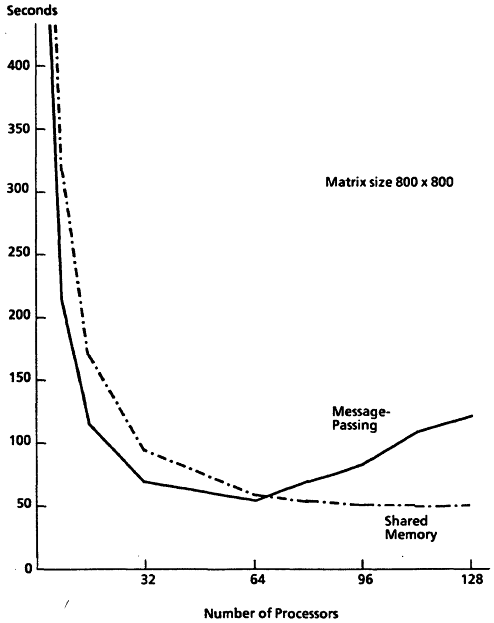
The improved performance of the message-passing implementation can be directly attributed to data locality. Nearly every data access in the shared memory implementation requires a remote reference. Even when the data physically resides in local memory, all data conceptually resides in shared memory and must be copied into the local workspace. Very few remote references are needed in the message-passing implementation (none, if only one processor is in use), and a copy takes place only when a remote reference is made.

Exploiting locality of data also minimizes switch contention. Unlike the shared memory implementation, the message-passing implementation for Gaussian elimination showed no significant switch contention. A problem of size 800x800 was able to execute on 32 processors without alternate paths at roughly the same speed as with the alternate paths. Over the entire range of experiments, the effect of alternate communication paths improved the execution time of the message-passing implementation by at most 3%.

Another measure of an application's performance is how well the implementation is able to use additional processors. Ideally, if a problem of size NxN using P processors requires time T, the same problem can be solved on 2P processors in time T/2. The extent to which the ideal is realized depends, in part, on the number of sequential operations in the implementation that depend on P.

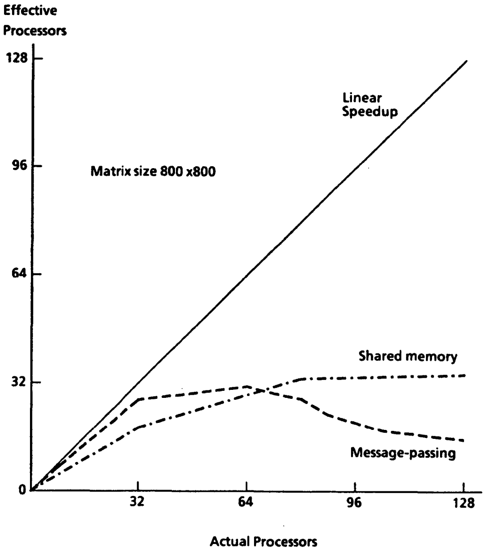
Figure 2 illustrates the scalability of the two implementations. The computation time for a program designed for a single processor is divided by the time required by the parallel version to yield the number of *effective* processors. With linear speedup, P actual processors yields P effective processors. As Figure 2 demonstrates, the message-passing implementation has nearly linear speedup with 16 processors, and a slight deviation from linear speedup with 32 processors. Speedup reaches a turning-point at 64 processors (31 effective processors) and additional processors only serve to *increase the execution time of the computation.*

The shared memory implementation, on the other hand, has a speedup factor that is less than linear, even for a small number of processors. For example, 4 real processors shows a speedup of 2.8 and 16 processors shows a speedup of 10.6. Speedup reaches a plateau when 64 processors are in use (30 effective processors). Additional processors do not have much impact and, at best, were able to yield 34 effective processors with 116 actual processors.



Shared Memory vs. Message-Passing: Performance

Figure 1



Shared Memory vs. Message-Passing: Speedup

Figure 2

In order to explain the "knee" in the performance curve for message-passing, we have to look at the implementation for operations that depend on P, the number of processors involved. Each broadcast operation requires time O(P). Since all data is local to some process, each process is responsible for modifying its local rows and broadcasting the results. Each processor that is to receive a broadcast must copy the message into local memory, so a broadcast requires P-1 processors to sequentially access a single remote memory. A total of N messages are broadcast to P-1 processors. Since each message is buffered locally before it is copied, the total number of copies is P*N. As P increases and N remains fixed, we reach a point of diminishing returns. Eventually, the time to copy an additional row in the presence of broadcast contention is not justified by the gain in parallelism.

The shared memory implementation does not exhibit a "knee" because, after initialization, there are no operations in the shared memory implementation that depend on P. The Uniform System encourages the programmer to avoid memory contention by distributing the data uniformly throughout the available memory. Tasks are allocated to processors independently of the data to be accessed. This means that while contention for a particular memory is reduced, overall switch contention is increased because practically all data references are remote. $N^2$ remote data copies, involving the non-zero portion of a whole row, are required, so the total number of copies is independent of the number of processors. Thus, if P is significantly smaller than N, as it should be to achieve high processor efficiency, the message-passing implementation will make many fewer copies than the shared memory implementation, although each copy (message) will take longer. Fewer copies will reduce both switch and memory contention, and improve performance, until the "knee" in the curve is reached. Beyond that point, additional copy operations cannot be justified by a significant gain in parallelism.

Our third criterion for comparison is ease of programming. Programming models are employed so that each application can be written without the knowledge of too many underlying details. Thus, an important criterion of any model is that it facilitate program construction. One concrete measure is the amount of user code that needs to be written. For Gaussian elimination, the amount of user code is comparable in both implementations. The Uniform System implementation contains 426 lines of user code, including comments, compared with 368 lines of user code in the message-passing implementation.

A related measure is the amount of user code unrelated to the specific application. That is, to what extent does the user code have to deal with the details of communication and synchronization? In the message-passing implementation, there is one call to **Broadcast** and another to **Receive**. There is no explicit synchronization in the user code. The shared memory implementation uses primitive atomic operations for synchronization and several calls to transfer blocks of memory. There are also spin locks that are sensitive to the amount of time delay between attempts to set the lock.

## Conclusions

The original intention of this work was to show that message-passing could play an important role in the design of a programming environment for a tightly-coupled multiprocessor. This exercise was designed to provide some empirical data that suggests how best to structure communication between processes in a tightly-coupled multiprocessor. We offer the following conclusions.

*The particular model of computation in use is less important than how well it is matched to the application.* Gaussian elimination is an appropriate application for a multiprocessor, however, this does not mean that the shared memory model is the best fit for this application. Since Gaussian elimination is essentially *value-oriented* (no addresses are communicated and rows are not used until they have stabilized), message-passing is a better model of the communication that takes place. There are other applications for which message-passing would not be appropriate. Any programming environment that offers a single model of communication will not be well-matched to a large class of applications.

*A high-level interface, efficiently implemented, leaves the programmer fewer opportunities to introduce inefficiency.* Of course this assumes that the high-level interface provides a *useful* abstraction. The Uniform System offers no high-level synchronization primitives; the two primitives provided are a busy wait LOCK and an UNLOCK primitive. Each programmer must use these to implement the appropriate synchronization. Spin locks are commonly used, but the resultant program can be especially sensitive to the amount of time spent between attempts to set the lock [5]. The only process synchronization in the message-passing implementation occurs during access to message buffers, which is implemented very efficiently using micro-coded atomic operations. The user does not need to be concerned with low-level synchronization; it is implicit in the message-passing primitives.

*The performance of an application depends not only on the efficiency of communication, but also on the extent to which the underlying model of computation encourages or discourages communication.* The Uniform System model does not encourage the programmer to exploit locality. Data is assumed to reside in one large, uniform address space and the boundaries between processors are ignored. Thus, in Gaussian elimination, each task must copy two rows into local memory to eliminate a single entry and then copy the result back. (Pivot rows were cached to avoid one of these copies.) This is not the case with the message-passing model, since all data is local to some process in the computation. Locality makes it possible to avoid copying any row to be modified (since only local rows are ever modified) and also to avoid copying any pivot row that happens to be local. Thus, an implementation based on very efficient communication (*e.g.*, shared memory) may perform worse than one based on a less efficient mechanism (*e.g.*, message-passing), if such efficiency encourages too much communication.

## Acknowledgements

I would like to thank Lawrence Crowl, Neal Gafter, and Thomas Olson for their help in performing the experiments. Liud Bukys, Stuart Friedberg, Walter Milliken (BBN), and Bob Thomas (BBN) provided technical assistance and several helpful suggestions. This work was supported by the National Science Foundation under grant number DCR-8320136 and DARPA/ETL under grant number DACA76-85-C-0001.

## References

1. BBN Laboratories, The Uniform System Approach To Programming the Butterfly Parallel Processor, Oct 1985.
2. W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken and T. Blackadar, Performance Measurements on a 128-Node Butterfly Parallel Processor, *Proceedings of the International Conference on Parallel Processing,* August 1985, 531-540.
3. T. J. LeBlanc, SMP: A System for Structured Message-Passing in the BBN Butterfly Multiprocessor, Computer Science Department, University of Rochester, Nov 1985.
4. W. Milliken et al, Chrysalis Programmer's Manual, Version 2.2, BBN Laboratories, June 1985.
5. R. Thomas, Using the Butterfly to Solve Simultaneous Linear Equations, Butterfly Working Group Note 4, Mar 1985.

# A COMMUNICATION MODEL FOR OPTIMIZING HIERARCHICAL MULTIPROCESSOR SYSTEMS

Santosh G. Abraham and Edward S. Davidson

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
305 Talbot Laboratory, 104 S. Wright St.
Urbana, IL 61801

## ABSTRACT

Experimental hierarchical multiprocessor systems, such as the Cm* [1],[2], EGPA [3] machine and the Cedar [4], show that hierarchical systems encompass a range of processor configurations and interprocessor communication links. Cost and performance models are developed for the components of a hierarchical system where all processors share a global memory and processors are grouped into clusters that share a cluster memory. With these models, it is shown that a unique combination of processor speeds and numbers best achieves a given performance goal. Given the distribution of interprocessor communication, the cluster size is chosen to minimize average communication delay. When the patterns of communication correspond directly to some parallel architecture, the analysis indicates the effectiveness of the hierarchical system in emulating that architecture. Our primary objective is to compare various processor configurations in large systems. Thus, the models we have developed ignore factors that tend to have a similar effect on a wide range of configurations.

## 1. INTRODUCTION

The shared memory model, in which all $p$ processors access a common memory in constant time, has been widely used in parallel processing research. Crossbar switches are too expensive to construct for large $p$. Multi-stage interconnection networks have order $lg\ p$ stages and a direct realization of the shared memory model results in an order $lg\ p$ increase in delay as $p$ increases. Local memory for each processor can decrease memory latency to some extent because a large fraction of all references are to variables that are private to each processor. An extension of this approach is to organize processors in groups (*clusters*) with a local *cluster memory* shared by each group of processors. Communication between processors in the same cluster is done through the cluster memory. Access to *global memories* is required primarily to communicate among processors in different clusters. Such a two-level hierarchical system may be extended to several levels. In this paper, we focus on two-level hierarchies.

As in uniprocessor systems, a large proportion of all references are to a small set of storage locations. The development of methods to ensure that the highest level in the hierarchy is populated mainly by this set has led to the success of cache-based systems. Furthermore, for many applications, it is possible to assign tasks so that interprocessor communication occurs mainly within small groups of processors. It is for these applications that providing a faster set of communication links within groups of processors results in improved performance. The models that we use here are similar in some respects to those in Welch [5] and in Vrsalovic et al [6].

We briefly describe three hierarchical systems. The Cm* is a two-level hierarchical multiprocessor system [1]. The Cm* hardware is made up of 50 processor-memory pairs called Computer Modules or Cm's [2]. These are grouped to form clusters containing up to 14 Cm's. Communication within the cluster is via a parallel bus controlled by a *Kmap*, which is an address mapping processor. There are five clusters and these communicate via intercluster buses. The Cm* is extensible, either by adding processors to each cluster (up to 14) or by increasing the number of clusters. The Cedar [4] uses a crossbar interconnection between the processors within a cluster and the cluster memory they share, and a multistage interconnection network between all processors and a global memory shared among all clusters. At present there are up to 8 processors per cluster and the number of clusters is extensible.

The Erlangen general purpose array, EGPA [3], is extensible only in the number of levels. Each processor except for the ones at the lowest level is connected to the memories of four subordinate processors. Data transfer is done on the EGPA via the common memories. Processors and memories in the same level are connected by additional links. Although hierarchical and tree are often used interchangeably, we assume that a ***hierarchical machine*** has processors at the leaf nodes and communicating nodes or memories at the internal nodes, whereas a ***tree machine*** has processors at internal as well as leaf nodes. We thus classify the EGPA as a tree machine. The models we develop are directly applicable only to hierarchical systems.

Several design decisions involve specifying gross system parameters. For instance, a mesh architecture can be specified by the size of the mesh, the speed of the individual processors, and the communication bandwidth and delay of the links. In a hierarchical (cluster) system, parameters include the number of processors and their speed, the number of levels in the hierarchy, the type of interconnection at each level, and the number of clusters. We have developed cost and performance

This work was supported in part by the National Science Foundation under Grant Nos. US NSF DCR84-06916 and US NSF DCR84-10110, and the US Department of Energy under Grant No. US DOE DE FG02-85ER25001.

0190-3918/86/0000/0467 $01.00 © 1986 IEEE

models and techniques that use them to make some of these decisions, given knowledge of the workload as input.

The efficient utilization of the resources in a multiprocessor system requires a match between the architecture and the application. Inter-task and interprocessor communication are determined by the algorithms used. Structured programming with hierarchical design may improve the mapping to a hierarchical multiprocessor. The importance of reducing interprocessor communication is well described in [7]. In order to examine the general suitability of hierarchical systems, we evaluate their ability to emulate the patterns of communication encountered in typical code for some important applications. We develop some methods to choose a processor configuration that is best suited for the patterns considered.

We present models for the components of a hierarchical system in section 2. In section 3, we show how these models help to obtain an optimum distribution of costs among the components. In section 4, we are concerned with the choice of cluster size. For some patterns of communication commonly encountered, we show that the cluster size can be optimized with respect to average communication delay. In section 5, we determine the cluster sizes that will minimize interprocessor communication delay for emulating certain processor networks.

## 2. COST AND PERFORMANCE MODELS

### 2.1. Parameters

We list some of the parameters that characterize a two-level hierarchical system (Fig. 2.1).

$p$ - number of processors in the system.
$s$ - speed of a single processor.
$c$ - number of processors in a cluster.
$p/c$ - number of clusters in a two-level system.

For a single application with a uniform pattern of communication, a uniform cluster size, $c$, is the best choice.

### 2.2. Cost and Access Times

We propose models for the three major components in a two-level hierarchical system, viz. the processors, the cluster interconnection, the global interconnection. These models are simple since the primary purpose is to develop quantitative methods to obtain a better understanding of some tradeoffs in large systems [8]. For example, we show a tradeoff between processor cost and communication structure cost. Other models can be substituted for the ones used here, but the basic analysis technique remains unchanged.

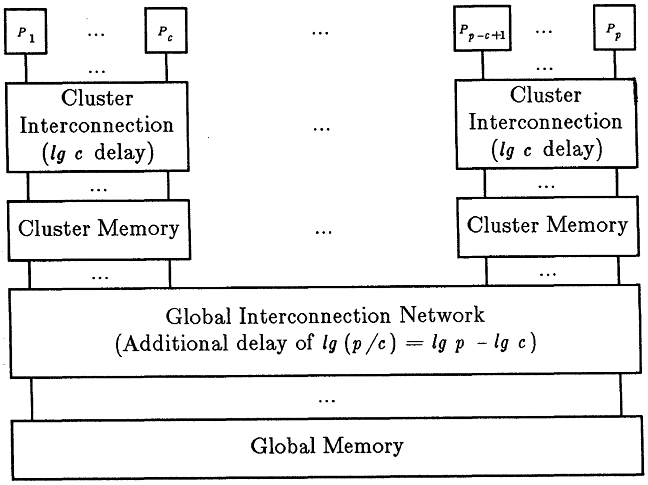

$P_1$
...
$P_c$
...
$P_{p-c+1}$
...
$P_p$
Cluster Interconnection ($lg\ c$ delay)
Cluster Interconnection ($lg\ c$ delay)
Cluster Memory
Cluster Memory
Global Interconnection Network
(Additional delay of $lg\ (p/c) = lg\ p\ - lg\ c$)
Global Memory

Fig. 2.1. Model of Communication Delay for a Hierarchical Multiprocessor.

### 2.2.1. Processors

In order to focus on the configuration issues of concern here, a processor is characterized by only a single parameter, speed. All processors are required to have the same speed.

One model for the individual processor cost as a function of speed is the power function $c(s) = s_0 . s^{\alpha}$. With this model, multiprocessing is a cost-effective option only if $\alpha \geq 1$. If $\alpha < 1$, the performance of a uniprocessor system can be improved at sublinear cost, while a multiprocessor system with additional processors can result in at most linear speedup. Such a power function is fairly commonly used [5] and is mathematically tractable. The boundary beyond which multiprocessing is undesirable can be stated concisely.

### 2.2.2. Global Interconnection

There are several strategies for interconnecting a large number of processors [9]. The ring and the mesh networks (Figs. 5.1 and 5.2) are good for certain applications, but they have a large diameter. For general purpose multiprocessors, multi-stage packet switched networks are usually most suitable for global interconnection and we model such networks here.

In such a network, there are $p$ requesters and a similar number of memory banks serving these requesters. The memory banks together constitute the global memory (Fig. 2.1). The number of switches in each stage is proportional to $p$. There are $lg\ p$ stages in the network. The total number of switches and the number of wires connecting switches are thus proportional to $p\ lg\ p$ and the cost of the global interconnection is modeled as $p\ lg\ p$. We assume that each switch operates in unit time. Processor speed, $s$, may thus be viewed as being relative to switch speed.

Since the number of switches that have to be traversed to reach the server is $lg\ p$, the model access time is proportional to $lg\ p$. Another rationale for modeling the access time or communication delay as $lg\ p$ is the following. Since the number of possible destinations is $p$ and the switches are binary in nature, the data should be switched through at least $lg\ p$ switches. Accordingly, the model delay of a link in a mesh is two, since there are four possible destinations. To obtain a fair comparison, it is necessary to penalize, with a larger delay, those parallel architectures with many communication links per processor.

### 2.2.3. Cluster interconnection

At the cluster level, a wider range of interconnection schemes is possible, since the number of processors is small [10]. We show that a crossbar for the cluster interconnection is not feasible if the cluster size is to be larger than $lg\ p$ and we may be constrained to multi-stage interconnection networks within such a large cluster.

A desirable if not essential characteristic of a large multiprocessor architecture is scalability. Scalability as it pertains to cost requires that the cost of a system should not increase much faster than the number of processors. We have seen that the cost of the global interconnection increases as $p\ lg\ p$. We require that the total cost of all cluster interconnections be similarly bounded.

Consider the choice of a crossbar for the cluster interconnection. Since there are $c^2$ switches in a crossbar of size $c$, the cost of the crossbar increases as $c^2$. Since there are $p/c$ crossbars, the total cost of the cluster crossbars is proportional to $p.c$. Since this cost must be bounded by $p\ lg\ p$, the size of a cluster, $c$, should not increase faster than $lg\ p$, as $p$ increases.

As we shall see for some cluster communication patterns, cluster size should increase faster than $lg\ p$ to ensure minimum communication delay. A multi-stage interconnection network within the cluster may then be more suitable. We assume a delay of $lg\ c$ for a cluster interconnection.

## 3. COST TRADEOFFS BETWEEN PROCESSORS AND THE COMMUNICATION NETWORK

The wide disparity in the cost of microprocessors and large scientific computers often evokes the suggestion of generating supercomputing power from a multi-microprocessor system. In a large multiprocessor, the interprocessor communication links can account for a significant fraction of the total cost. In our model, the cost of the interconnection network increases faster than the number of processors. Even if the choice of cheaper and slower, but more processors to attain a required system performance reduces the total processor cost, the larger interconnection network might increase total cost. Additionally, as the number of processors increases the utilization of the processors decreases, where utilization is the fraction of the total number of processors that can be effectively used. As a result of these factors, higher levels of performance are best achieved by using sufficiently fast processors as system components. In this section, we develop quantitative methods for evaluating such tradeoffs, e.g. to determine the number and speed of the processors.

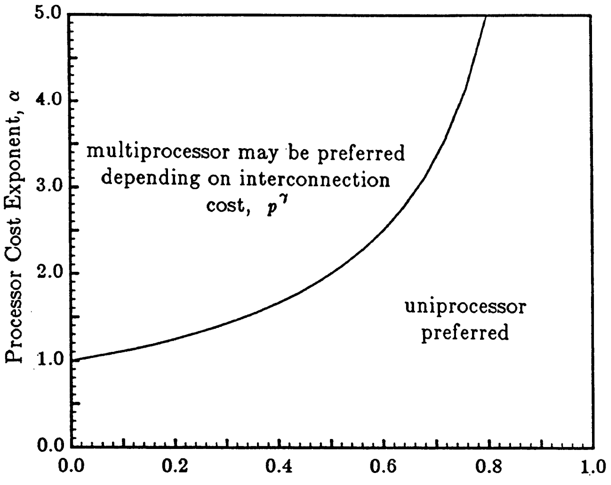


Fig. 3.1. Region in which a Uniprocessor is Cost-Effective. ($\alpha(1 - \beta) < 1$)

A combined analysis of the factors affecting cluster size, $c$, and system size (the number of processors), $p$, is difficult. Although the cluster size can affect the cluster interconnection cost, we approximate the total cluster interconnection cost to be a function of system size, $p$. This enables us to find the system size, $p$ without a knowledge of the cluster size. Further, we approximate the interconnection cost to be $p^\gamma$. Here $\gamma$ is chosen so that $p^\gamma$ approximates the model cost of $p\ lg\ p$ for the range of system sizes under consideration. This enables us to develop closed form expressions. Iterative solutions can be obtained without this additional assumption.

The parameters of the model are listed below.

$P_0$ - desired performance level.
$s$ - individual processor speed.
$s_0 s^\alpha$ - individual processor cost.
$p^\gamma$ - communication structure cost.
$p^{-\beta}$ - processor utilization factor.
$C$ - total cost of the system.

We model the utilization factor as $p^{-\beta}$, because the utilization factor is generally observed to decrease as the number of processors increases. If we obtain linear speedup, $\beta=0$ and the processor utilization factor is one. If there is no speedup, $\beta=1$ and this factor is $1/p$. Ordinarily, $\beta$ lies between 0 and 1, where smaller values imply better utilization.

The objective is to attain performance $P_0$ with minimum total cost, $C$. The total processing capacity is $s.p$, of which a fraction $p^{-\beta}$ is used. Therefore,

$$P_0 = s.p.p^{-\beta} = s.p^{(1-\beta)} \qquad (3.1)$$

The cost $C$ is the sum of processor and the communication structure costs, namely

$$C = p.s_0.s^\alpha + p^\gamma \qquad (3.2)$$

From (3.1)

$$s = P_0/p^{(1-\beta)} \qquad (3.3)$$

Substituting (3.3) in (3.2),

$$C = s_0.P_0^\alpha.p^{1-\alpha(1-\beta)} + p^\gamma \qquad (3.4)$$

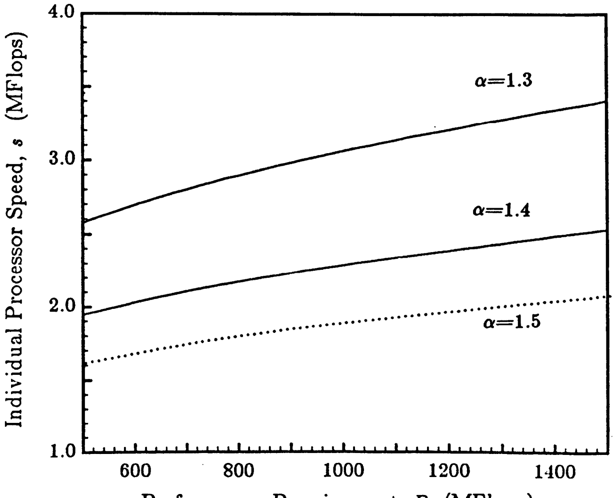


Fig. 3.2. Plot of Individual Processor Speed, $s$ vs. Performance Requirement, $P_0$. (from (3.7) in (3.3) with $\beta$=0.1, $\gamma$=1.4, $s_0$=25.0)

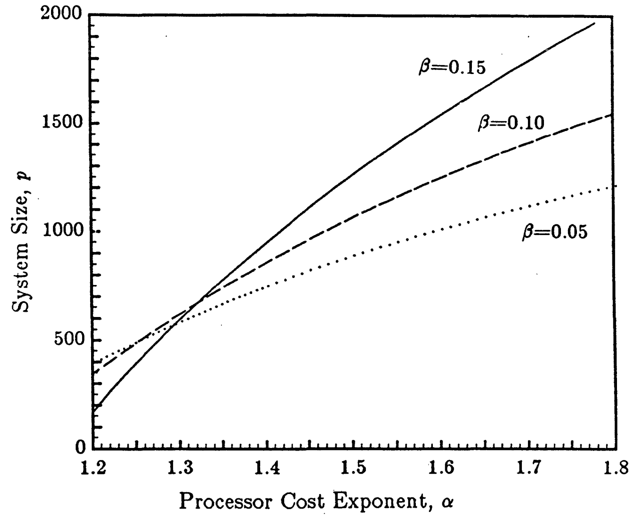


Fig. 3.3. Plot of System Size, $p$, vs. Processor Cost Exponent, $\alpha$. (from (3.7) with $\gamma$=1.4, $s_0$=25.0, $P_0$=1000)

Differentiating (3.4),

$$\frac{dC}{dp} = [1 - \alpha(1 - \beta)].s_0.P_0^{\alpha}.p^{-\alpha(1 - \beta)} + \gamma.p^{\gamma-1} \tag{3.5}$$

Setting (3.5) to zero, we note that there is a solution only if

$$\alpha(1-\beta) \geq 1 \tag{3.6}$$

In Fig. 3.1 a uniprocessor solution is best in the region below the curve. Within this region, if a uniprocessor system with cost $C$ achieves performance $P_1$ and a two-processor system with individual processor cost $C/2$ achieves performance $P_2$, then $P_1 > P_2$. This can be shown by substituting for $s$ from (3.2) in (3.1), assuming that the interconnection cost is zero.

If (3.6) holds, we can set (3.5) to zero and solve for $p$. We obtain the following closed form solution for $p$.

$$p = [P_0^{\alpha} s_0 (\frac{\alpha(1 - \beta) - 1}{\gamma})]^{\frac{1}{[\gamma - 1 + \alpha(1 - \beta)]}} \tag{3.7}$$

From (3.3) we can now find the processor speeds (Fig. 3.2).

In Fig. 3.2, we see that higher performance is best achieved by increasing processor speeds in addition to increasing the number of processors. For higher processor cost exponents, $\alpha$, corresponding to rapidly increasing processor cost versus speed, the same performance is best achieved by a greater number of slower processors (Figs. 3.2 and 3.3). If the cost of interconnecting processors is larger (larger $\gamma$), it is more economical to use fewer and therefore faster, more expensive processors to reduce the interconnection cost at the expense of processor cost (Fig. 3.4).

## 4. CLUSTER SIZES FOR VARIOUS APPLICATIONS

In the previous section, we developed techniques to determine the number of processors and their speeds. In a two-level cluster system, specifying the cluster size completes the processor configuration. If the cluster interconnection has nearly a linear cost-size relationship, the choice of cluster size has little effect on the cost of the system. The choice of cluster size does, however, have a significant effect on interprocessor communication delay. If the cluster interconnection is a multi-stage interconnection network, a cluster size is best determined from performance considerations. Though otherwise identical two-level systems with different cluster sizes may not appreciably differ in cost, there may be a significant cost difference between such a system and a similar system, like the NYU Ultracomputer [11], with a single global memory and no cluster memories. In this section, we propose some methods for determining the best cluster size.

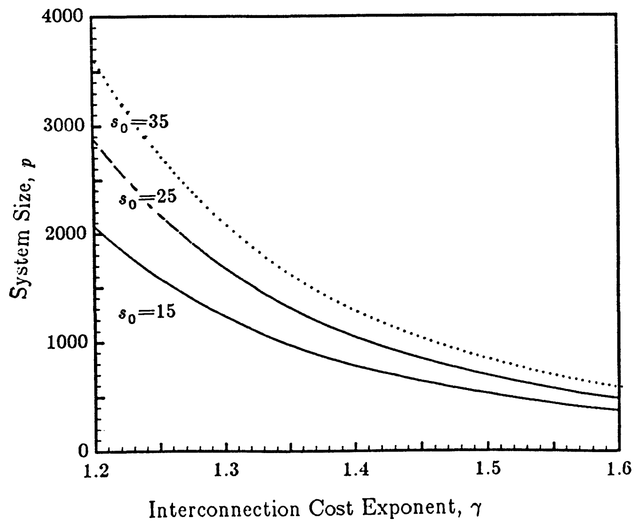


Fig. 3.4. Plot of System Size, $p$, vs. Interconnection Cost Exponent, $\gamma$. (from (3.7) with $\alpha$=1.5, $\beta$=0.1, $P_0$=1000)

### 4.1. Cluster Size from Probability Distribution

We first propose a model for the distribution of references between processors. We assume that all references to cluster and global memory are to communicate between processors in the same cluster and in distinct clusters, respectively. This assumption is not perfectly correct since the cluster and even the global memory may be used to avoid duplication of variables among local memories, which are private to each processor. We choose any processor $p_j$ and determine the distribution of interprocessor communication $c_j(p_i)$ between processor $p_j$ and the other processors $p_i$ in the system. The $c_j(p_i)$ are scaled so that $\sum_i c_j(p_i) = 1$. We assign an integer rank $r_j(p_k)$ to a processor $p_k$ so that $r_j(p_k) = l$ if there are $l$ processors, $p_i$, with $c_j(p_i) \geq c_j(p_k)$. We plot the communication $c_j(p_i)$ versus the rank $r_j(c_i)$ of each processor. We call this the *communication distribution curve* and by definition it is a monotonically non-increasing curve. We assume that such plots for all the processors are identical, i.e. all processors have similar patterns of communication. Additionally, we assume that if we choose a cluster size of $c$, we can obtain a *compatible assignment* of processors to disjoint clusters. In a compatible assignment, for each processor, $p_j$, processors, $p_i$ with rank, $r_j(p_i)$ less than $c$ belong to the cluster containing $p_j$. With such a model, we show that there is an optimum cluster size which minimizes average communication delay.

*Example :* Consider a system of four processors, $p_1$ through $p_4$. Let the fraction of communication between any two processors be specified by the following table.

| **Communication between Processors** (total communication of each processor normalized to 1) | | | | |
|---|---|---|---|---|
| | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
| $p_1$ | | 0.7 | 0.2 | 0.1 |
| $p_2$ | 0.7 | | 0.1 | 0.2 |
| $p_3$ | 0.2 | 0.1 | | 0.7 |
| $p_4$ | 0.1 | 0.2 | 0.7 | |

Table 4.1. Distribution of Interprocessor Communication.

The first step involves selecting a processor, say $p_3$. The next step involves ranking the remaining processors. Since the processor communicating the most with $p_3$ is $p_4$, the rank of $p_4$, $r_3(p_4)$, is 1. The ranks $r_3(p_1)$, $r_3(p_2)$ of $p_1$ and $p_2$ are 2 and 3 respectively. We can now plot the curve of fraction of communication versus rank through the points (1,0.7), (2,0.2) and (3,0.1). Such curves for the other processors are identical to the one for $p_3$. Choosing a cluster size of 2 and assigning $\{p_1, p_2\}$ to one cluster and $\{p_3, p_4\}$ to the second cluster forms a compatible assignment. We note that, by our model, the average communication delay with no cluster memory is $lg\ 4 = 2$. With a cluster size of 2, the average communication delay is $0.7\ lg\ 2 + (0.2 + 0.1)\ lg\ 4 = 0.7 + 0.6 = 1.3$.

Given a continuous communication distribution curve $f(q)$, we can find an optimum cluster size. With cluster size, $c$, the average communication delay, $D_c$ is given by

$$D_c = lg\ c. \int_0^{c-1} f(q).dq + lg\ p. \int_c^{p-1} f(q).dq \qquad (4.1)$$

From the definition of the communication distribution curve, it follows that

$$\int_0^{p-1} f(q).dq = 1 \qquad (4.2)$$

Eqn. (4.1) can thus be rewritten as

$$D_c = lg\ p - (lg\ p\ - lg\ c). \int_0^{c-1} f(q).dq \qquad (4.3)$$

The original objective of $\underset{c}{Min}\ D_c$, given $p$, is equivalent to

$$\underset{c}{Max}\ [(lg\ p\ - lg\ c). \int_0^{c-1} f(q).dq] \qquad (4.4)$$

If $f(q)$ is known, we can determine the cluster size that minimizes average communication delay from (4.4).

We examine the changes in optimum cluster size for a class of communication distribution curves of the form $f(q) = f_0.2^{-\lambda q}$ where $\lambda$ is a parameter and $f_0$ is chosen to normalize $f(q)$. Solving

$$\int_0^{p-1} f_0.2^{-\lambda q}\ dq = 1 \qquad (4.5)$$

we obtain

$$f_0 = \frac{\lambda\, ln\ 2}{1 - 2^{-\lambda(p\ -1)}} \qquad (4.6)$$

Substituting for $f(q)$ in (4.4), the objective is to maximize

$$(lg\ p\ - lg\ c)(\frac{\lambda \ln 2}{1 - 2^{-\lambda(p\ -1)}} \int_0^{c-1} 2^{-\lambda q} dq) \qquad (4.7)$$

Differentiating (4.7) with respect to c, simplifying and setting to zero we obtain the following implicit relationship.

$$lg\ c + \frac{1}{\lambda c. \ln^2 2}(2^{\lambda(c-1)} - 1) = lg\ p \qquad (4.8)$$

Once $p$ and $\lambda$ are known, we can solve (4.8) iteratively for the optimum cluster size, $c$.

Since the choice of cluster size, $c$ is ordinarily constrained to powers of 2, we plot the logarithm of cluster size, $lg\ c$ against the communication exponent, $\lambda$ (Fig. 4.1). When the communication exponent, $\lambda$ is large, a large proportion of the interprocessor communication is to a few processors. In this case, it is best to choose a small cluster size, $c$. A larger cluster size will slow down all references and will not sufficiently increase the fraction of total interprocessor communication satisfied directly by the clusters.

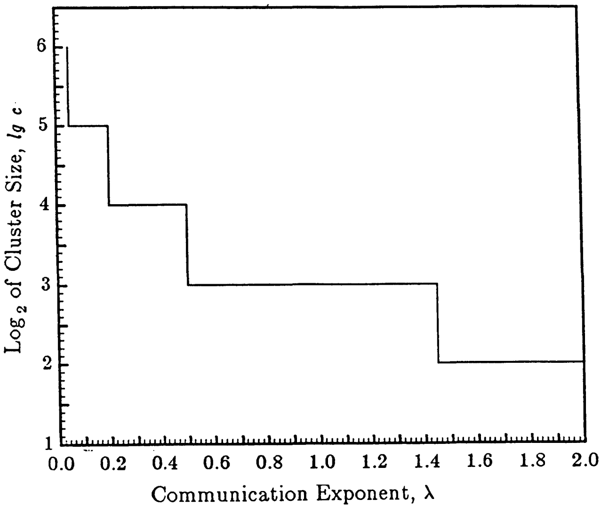


Fig. 4.1. Plot of Cluster Size, $c$ vs. Communication Exponent, $\lambda$ (p = 512).

### 4.2. Cluster Sizes for Fixed Degree Networks

A hierarchical multiprocessor has been proposed as a general purpose multiprocessor, with respectable efficiency over a range of applications. We consider a class of applications, which is frequently encountered, namely problems having a pattern of communication in which each node communicates to a fixed number of neighboring nodes. The communication graph for this class of problems is regular with a fixed degree, $d$. Additionally, we require that a network of size $p$ be reducible to a network of size $p/c$, so that $c$ nodes in the original network are mapped onto a single node in the reduced network and the reduced network belongs to the same class of networks. In [12], the computation factor, $f_c$, is defined to be the number of nodes mapped onto a single node and the exchange factor, $f_e$, is defined to be the number of links that map onto a single link. For the restricted class of networks and reductions under consideration, the computation factor, $f_c$ is always $c$ since we map networks of size $p$ to $p/c$ clusters, each with $c$ processors. For fixed degree networks, the exchange factor, $f_e$, lies between 1 and $c$. This technique is described further in [12].

Many common networks satisfy these requirements. For instance, each node in a mesh (Fig. 5.2) has four neighbors regardless of the size of the mesh. A mesh can be reduced to a smaller mesh, by coalescing square meshes of size $\sqrt{c}$ by $\sqrt{c}$. The computation factor $f_c$ is $c$ and the exchange factor $f_e$ is $\sqrt{c}$. For a mesh, the degree of reduction, $c$, changes the compute/communication balance at the global level by a factor of $f_e/f_e = \sqrt{c}$. In general, the improvement of a hierarchical system over a multiprocessor system with a single global memory depends on this ratio for the application. There are many applications and algorithms having a mesh-like pattern of communication. It may be best to execute such an application on a mesh, but a mesh may not be adequate for other algorithms that have other patterns of communication. A single hierarchical multiprocessor may adequately support a broad range of other patterns.

A multiprocessor system with a single global memory has a communication delay of $lg\ p$. Adding a cluster interconnection to such a system for faster communication between processors within a cluster involves an additional cost which must be justified by performance improvement. We require that a hierarchical system with the same number of processors have an average delay of $(lg\ p)/k$, where the $k$ divisor is intended to compensate for the additional cost of the cluster interconnection and memory. With such a requirement, we show that some range of cluster sizes can be excluded from further consideration for the class of applications described in the first paragraph of this section. We are able to exclude both very small and very large cluster sizes, by making a few assumptions on the workload.

We can now measure average communication delay for a hierarchical system with clusters of size $c$ as

$$D_c = \frac{a_c.lg\ c + a_g.lg\ p}{a_c + a_g} \tag{4.9}$$

where $a_c$ and $a_g$ are the number of cluster and global accesses respectively. We require

$$D_c \leq (lg\ p)/k \tag{4.10}$$

The nodes within a cluster of size $c$ make a total of $d.c$ accesses, since the communication graph has degree $d$. Since the reduced communication graph has degree $d$ with $f_e$ links of the original graph mapped onto each link in the reduced graph, the mapping of nodes onto clusters causes each cluster to make $d.f_e$ global accesses to other clusters. The number of accesses that are served within a particular cluster is $d.c - d.f_e$. From (4.9) and (4.10),

$$D_c = \frac{(d(c - f_e).lg\ c + d.f_e.lg\ p)}{d.c} \leq \frac{lg\ p}{k} \tag{4.11}$$

Solving for $lg\ c$ in (4.11),

$$lg\ c \leq (\frac{c - k.f_e}{k(c - f_e)})lg\ p \tag{4.12}$$

This relationship is always satisfied if $k = 1$, because the average communication delay can be no worse than $lg\ p$. For $k > 1$, it cannot be satisfied when $c > p^{1/k}$. With a cluster size larger than $p^{1/k}$, an average delay greater than $(lg\ p)/k$ is to be expected, because even cluster reference delay is more than $(lg\ p)/k$. Since $c > f_e$, if $c < k.f_e$ the right hand side in (4.12) is negative, requiring the cluster size to be less than one (an impossibility). So a lower bound on cluster size is $k.f_e$. In general, $f_e$ is a function of $c$. For a mesh, $f_e$ is $\sqrt{c}$ and the lower bound on cluster size is $k^2$. These results can be summarized as

$$k.f_e \leq c < p^{1/k} \tag{4.13}$$

Tighter upper bounds on the cluster size, $c$ can be developed if the exchange factor is known. In Fig. 4.2, we plot upper and lower bounds on $c$ obtained by an iterative solution of (4.11) with $f_e = \sqrt{c}$ and $k = 1.5$. For system sizes smaller than 64, no choice of cluster size can yield the desired improvement factor.

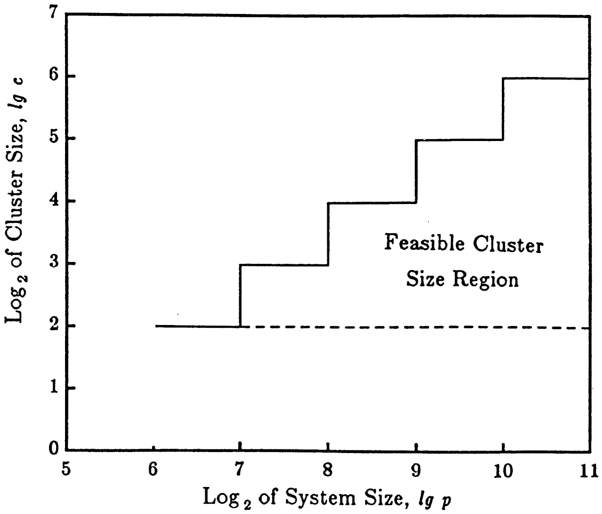


Fig. 4.2. Feasible Cluster Size Region vs. System Size, $p$ for a Mesh with Improvement Factor, $k = 1.5$.

We note that these bounds are independent of the degree of the network. In this section, we have been concerned with the efficiency with which a hierarchical multiprocessor can support the communication requirements of algorithms and the extent to which this can be improved by using clusters. A related problem is to determine how well such a system can emulate the communication links of other processor networks. The results in section 4.2 are equally applicable to processor networks and essentially state that very small and very large cluster sizes can be excluded from consideration.

## 5. EMULATION OF PROCESSOR NETWORKS

In this section, we consider the emulation of some common processor interconnections on a hierarchical system. We show that the average communication delay is minimized by an appropriate cluster size.

We outline the general technique used in the subsequent subsections on specific processor networks. In each case, we consider a processor network with $p$ processors and a two-level hierarchical system of the same size. For some scheme of mapping the processors in the network onto the clusters in the two-level system, we find the ratio of cluster to global accesses as a function of the cluster size. We then use (4.9) to find the best cluster size and its average communication delay. In general, it is difficult to find the best mapping from a network to a two-level system. However, since the networks considered are regular, the best quotient mappings are obvious.

### 5.1. Ring Network

Consider the emulation of a ring of size $p$ on a two-level hierarchy. We divide a ring into $p/c$ segments each containing $c$ processors. We map the processors in each segment of length $c$ onto the processors in a single cluster of size $c$. Consider a typical communication pattern in which each processor in the ring accesses its left neighbor. There are then $c$-1 cluster accesses for each global access (Fig. 5.1). It is easy to see that no other mapping scheme can result in a larger ratio of cluster to global accesses.

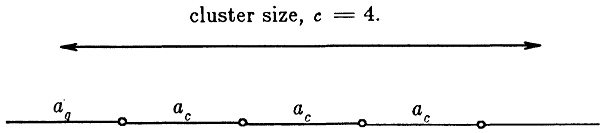


Fig. 5.1. Mapping a Ring Segment onto a Cluster of Size, $c$ = 4.

Substituting for cluster and global accesses in (4.9),

$$D_c = \frac{(c - 1)lg\ c + lg\ p}{c} \qquad (5.1)$$

Differentiating (5.1) with respect to $c$, setting to zero, and solving for $lg\ p$,

$$lg\ c + \frac{c - 1}{\ln 2} = lg\ p \qquad (5.2)$$

We can solve (5.3) iteratively for $c$, or if $c >> 1$ and $c >> lg\ c$, we can approximate (5.2) as

$$c_{opt} = \ln 2.lg\ p \qquad (5.3)$$

With a cluster size of $\ln 2.lg\ p$, the average communication delay is

$$D_{c_{opt}} = (lg\ lg\ p + \frac{\ln\ln 2}{\ln 2})(1 - \frac{1}{\ln 2.lg\ p}) + \frac{1}{\ln 2} \qquad (5.4)$$

For large systems, this can be approximated as

$$D_{c_{opt}} \approx lg\ lg\ p(1 - \frac{1}{\ln 2.lg\ p}) \qquad (5.5)$$

The delay here is a logarithmic factor better than in a system with no clusters and a single global memory.

### 5.2. Mesh Network

Consider the emulation of a toroidal mesh with dimensions $\sqrt{p}$ by $\sqrt{p}$ on a two-level hierarchical system. We map a sub-mesh of size $\sqrt{c}$ by $\sqrt{c}$ onto a cluster of size $c$. Assume (without loss of generality, by symmetry) that all the processors in the mesh access their left neighbors (Fig. 5.2). The number of cluster accesses, $a_c$, is $\sqrt{c}(\sqrt{c} - 1)$ and the number of global accesses, $a_g$, by processors in the cluster is $\sqrt{c}$. Substituting in (4.9),

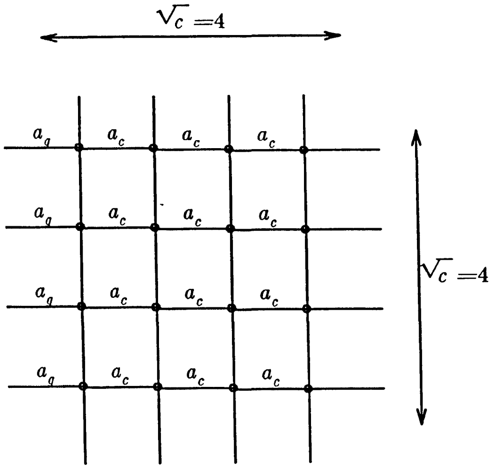


Fig. 5.2. Mapping a Sub-mesh onto a Cluster of Size, $c$ = 16.

$$D_c = \frac{\sqrt{c}(\sqrt{c} - 1)lg\ c + \sqrt{c}lg\ p}{c} \qquad (5.6)$$

Differentiating with respect to $c$, setting to zero, and solving for $lg\ p$,

$$lg\ c + \frac{2(\sqrt{c} - 1)}{\ln 2} = lg\ p \qquad (5.7)$$

We can solve (5.9) iteratively for c or if $\sqrt{c} >> 1$ and $\sqrt{c} >> lg\ c$, (5.7) simplifies to

$$c_{opt} \approx (\frac{\ln 2}{2})^2 lg^2 p \qquad (5.8)$$

For large systems, the average delay ignoring small constant terms is

$$D_{c_{opt}} = lg\ lg\ p(1 - \frac{2}{\ln 2.lg\ p}) \qquad (5.9)$$

In order to obtain a logarithmic improvement in communication delay, the cluster size has to be quite large compared to the results for a ring.

### 5.3. Hypercube Network

Consider emulating a $lg\ p$-dimensional hypercube with $p$ processors on a hierarchical system. The interconnection scheme in a hypercube is best described with a binary numbering of the processor nodes. Processors whose binary numbers differ in exactly one digit are connected by direct links. Under our model, the average communication delay of a multiprocessor interconnection strategy is the $\log_2$ of the number of links per processor. Since each processor node is connected to $lg\ p$ processors by direct links, the hypercube has an average communication delay of $lg\ lg\ p$.

We map the processors in a hypercube onto a two-level system in the following manner. A $lg\, p$-dimensional hypercube can be split into two smaller hypercubes of dimension $lg\, p - 1$, by removing the links connecting processors differing in a particular digit (for instance, all links connecting processors whose numberings differ in the leftmost digit). Since $c$ is usually a power of 2, the hypercube can be divided into $p/c$ smaller hypercubes of dimension $lg\, c$ and each of these can be assigned to a cluster of size $c$. In order to analyze the average communication delay, we assume that a hypercube is equally likely to use any of its links. Of the $lg\, p$ links available to each processor, $lg\, c$ are mapped within the cluster. Hence, the average communication delay from (4.9) is

$$D_c = \frac{lg\, c . lg\, c + (lg\, p - lg\, c) lg\, p}{lg\, p} \qquad (5.10)$$

By analysis as above, the best choice of cluster size is

$$c_{opt} = p^{1/2} \qquad (5.11)$$

The delay for this cluster size is

$$D_{c_{opt}} = \frac{3}{4} lg\, p \qquad (5.12)$$

This delay is not substantially superior to the $lg\, p$ delay for a global memory multiprocessor with no clusters and it is factor of $\frac{3 lg\, p}{4 lg\, lg\, p}$ worse than the hypercube itself.

## 6. CONCLUSION

We have developed simple models for the components of a hierarchical multiprocessor system. A two-level hierarchical system can be specified by the total number of processors, the speed of individual processors, and the size of the cluster. Based on these models, we developed expressions for the optimum number of processors and their speed. Optimum cluster size is application dependent. If the communication probability distribution curve for an application is available, we can find the cluster size which will minimize interprocessor communication delay. For a broad range of problems, we find that we can classify certain ranges of cluster sizes as unsuitable. The choice of a suitable cluster size can minimize the average communication delay when emulating certain processor networks.

While we could use more elaborate models in our analysis, we would not have relatively simple expressions as we have now. Some tradeoffs are quantified easily using the current models: the tradeoff between the cost of the processors and the cost of the communication structure, and the tradeoff between having a large cluster size that contains more references within it and having a smaller cluster size with less communication delay for each intracluster reference. We feel that even though the models are simple they do not unduly favor particular processor configurations.

Average communication delay is a useful metric in choosing a cluster size. However, if the average communication delay is $lg\, lg\, p$, it does not necessarily indicate that the execution times will be degraded by that factor. Frequently a processor can issue a read request and then perform useful computation while the request is being satisfied by the network.

## REFERENCES

[1] R. J. Swan, A. Bechtolsheim, K. -W. Lai. J. K. Ousterhout, "The implementation of the Cm* multi-microprocessor," *Proc. National Computer Conference,* 1977, pp. 645-654.

[2] E. F. Gehringer, A. K. Jones and Z. Z. Segall, "The Cm* testbed," *IEEE Computer,* Oct. 1982, pp. 40-53.

[3] H. Fromm et al, "Experiences with performance measurement and modeling of a processor array," *IEEE Trans. on Computers,* Vol. C-32, Jan. 1983, pp. 15-31.

[4] D. L. Kuck, E. S. Davidson, D. H. Lawrie and A. H. Sameh, "Parallel supercomputing today and the Cedar approach," *Science,* Vol. 231, 28 Feb. 1986, pp. 967-974.

[5] T. A. Welch, "Memory hierarchy configuration analysis," *IEEE Trans. on Computers,* Vol. C-27, May 1978, pp. 408-413.

[6] D. Vrsalovic, E. F. Gehringer, Z. Z. Segall and D. P. Siewiorek, "The influence of parallel decomposition strategies on the performance of multiprocessor systems," *Proc. 12th Int. Symp. on Computer Architecture,* 1985, pp. 396-405.

[7] B. Lint and T. Agerwala, "Communication issues in the design and analysis of parallel algorithms," *IEEE Trans. on Software Eng.,* Vol. SE-7, No. 2, Mar. 1981, pp. 174-188.

[8] B. Kumar and E. S. Davidson, "Computer system design using a hierarchical approach to performance evaluation," *Comm. of the ACM,* Vol. 23, No. 9, Sept. 1980, pp. 511-521.

[9] T. Feng "A survey of interconnection networks," *Computer,* Dec. 1981, pp. 12-27.

[10] D. P. Agrawal and I. E. O. Mahgoub, "Performance analysis of cluster-based supersystems," *Proc. 1st Int. Conf. on Supercomputing Systems,* 1985, pp. 593-602.

[11] A. Gottlieb et al, "The NYU Ultracomputer - Designing a MIMD, shared memory parallel machine," *IEEE Trans. on Computers,* Vol. C-32, Feb. 1983, pp. 175-189.

[12] J. P. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Trans. on Computers,* Vol. C-31, April 1982, pp. 288-295.

# CAMP: A PROGRAMMING AIDE FOR MULTIPROCESSORS[a]

Jih-Kwon Peir[b]
Daniel D. Gajski

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

## Abstract

The major issue in multiprocessing is not the construction of a multiprocessor system, but its efficient use in real applications. This paper describes CAMP, a program-development tool that helps a programmer in coding problems for multiprocessors. CAMP can partition programs, insert synchronization primitives, and simulate program performance. This way, a programmer may experiment with different algorithms for solving the same problem and select the one that fits his multiprocessor system the best.

## 1. Introduction

In the past several years we have seen many proposed and commercial multiprocessor architectures, all aimed at increasing machine performance by an order of magnitude. Although, faster hardware is easy to build these days, there is no agreement on how to achieve this performance increase for realistic applications.

There are basically four schools of thought as to what is the most important factor in obtaining higher performance in a particular machine. The first school of thought believes in faster circuit technology, which will allow us to retain present architectures, possibly augmented with a mechanism for synchronizing parallel processes. The second school puts priority on optimizing or vectorizing compilers, possibly interactive, that will detect parallelism and restructure sequential programs into their parallel format. The third school that believes in a dramatic increase in performance from new parallel algorithms supports development of new languages that will allow easy conversion of algorithms into programs. The fourth school supports new models of computation, such as data flow models, that will allow dramatic increases in parallelism and can be easily exploited by a multiprocessor architecture.

Although each school deals with one part of the solution, none of them shows how to optimally combine application requirements with the capabilities of VLSI technology. The first school uses the least risky approach by retaining old programming and architectural models. It takes advantage only of the speed that the new technology offers, not of its density. The second and third schools retain old architectural models, which may no longer be cost effective. Furthermore, the second school retains programming models developed for pre-VLSI architectures. The fourth school, although the most progressive, does not take into account technological limitations. For this reason, machines based on new models of computation do not exhibit impressive performance. Instead, they create new problems on their own.

The current VLSI technology allows building multiprocessors at low cost. However, This multiprocessor approach introduces three new requirements not encountered in the single processor environment [1]. First, each program must be partitioned into tasks executable on one or more processors. If a task is executed on more than one processor, it must be further partitioned into processes. Second, every task and process must be scheduled for execution on a particular processor or processors. Third, synchronization must be performed between concurrently executing tasks and processes to maintain data dependences in the program.

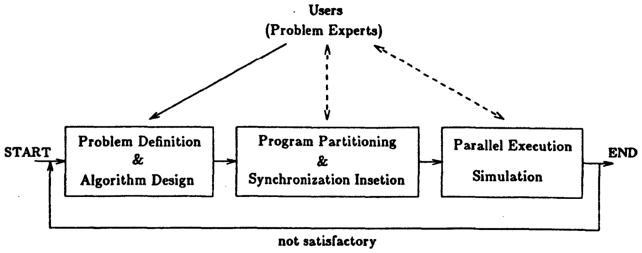
We support a fifth school of thought, which believes that only efficient solutions of the above problems will bring an order-of-magnitude improvement in multiprocessor performance. There are two extreme approaches in achieving this goal. In one, the application programmers (or the problem specialists) are supposed to solve the problem of partitioning, scheduling, and synchronization [2] [3]. Since parallelism is not a natural to human thinking process, this approach may be too cumbersome for all but very simple problems. On the other hand, we may use a restructuring compiler [4] [5] for partitioning, scheduling and synchronization of sequential programs. This approach assumes that such a compiler has good knowledge over wide range of application domains and that the algorithms used in the programs are optimal over different multiprocessor architectures.

We advocate an approach in the middle of the above two extremes. We assume that the programmer is an expert in his application domain. In the mean time,

[a] This work was supported in part by the **Hughes Aircraft Company** under Contract No. 1-5-37368, and the **IBM** graduate fellowship program.

[b] The author's current address: T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

we also realize the shortcomings of human thinking in a parallel environment. In our approach, the user extracts different segments of the program while a program-development tool helps in partitioning these program segments into a number of processes, and inserting synchronization primitives where needed. The simulator incorporated in the tool estimates the performance for different partitioning and synchronization strategies. This way, the programmer may use the tool repeatedly to explore several algorithms of solving the same problem and select one that fits his multiprocessor architecture. The essence of the computer-aided multiprocessor programming tool (CAMP) is shown in Figure 1. Note that scheduling each process to a physical processor is a runtime procedure. However, partitioning a task into a certain number of processes, each of which is associated with a virtual processor, is performed by the tool.



Users
(Problem Experts)
START
Problem Definition & Algorithm Design
Program Partitioning & Synchronization Insetion
Parallel Execution Simulation
END
not satisfactory

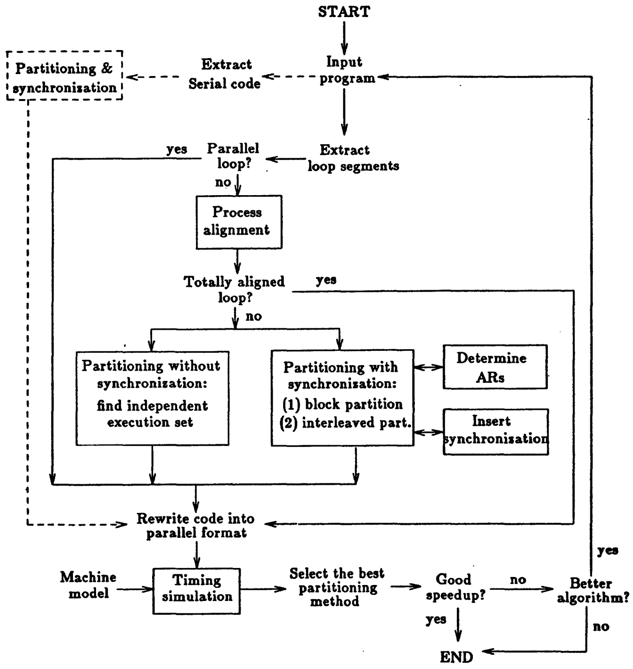
**Figure 1. The program-development tool**

In this paper, we select a loop example to demonstrate the program-development methodology with CAMP. We describe different partitioning and synchronization methods which have been implemented in CAMP for program loops. Most of the parallelism in scientific computations comes from loop statements. In the parallel execution of a loop, each iteration is usually treated as the smallest unit of execution and assigned to a processor in the lexicographical order of its index set. This method is very effective when all iterations are independent. However, when data dependences exist between different iterations, a proper partitioning and synchronization mechanism must be used to assure fast and correct execution.

The layout of this paper is as follows. In section 2, we will describe the basics of CAMP. Then, in the subsequent sections, we will discuss several methods which have been implemented in CAMP. We will describe the process alignment for eliminating synchronization between concurrent processes, a synchronization method, and different partitioning strategies for program loops in sections 3, 4, and 5 respectively. Finally, we will present results of a walk-through example and propose future research in section 6.

## 2. CAMP: A Program-development Tool

Figure 2 shows the flowchart of CAMP. Each step in a box is performed by CAMP and those steps without boxes are programmers' jobs. All the jobs for the programmers involve either a decision making or the extraction and rewriting of program codes, which we believe human can do better than a machine. Note that in this paper we described only the parts used for developing parallel code for program loops.



START
Input program
Extract Serial code
Partitioning & synchronization
Extract loop segments
yes
Parallel loop?
no
Process alignment
Totally aligned loop?
yes
no
Partitioning without synchronization: find independent execution set
Partitioning with synchronization: (1) block partition (2) interleaved part.
Determine ARs
Insert synchronization
Rewrite code into parallel format
yes
Machine model
Timing simulation
Select the best partitioning method
Good speedup?
no
Better algorithm?
yes
no
END

**Figure 2. The flowchart of tasks in CAMP**

Initially, a programmer extracts a loop segment from the original program. He determines whether the extracted loop has independent iterations (called a parallel loop). If so, the programmer can transform the loop into parallel format easily and send the parallel code to the simulator. If the loop is not a parallel loop, CAMP will attempt to align the loop to eliminate synchronization between iterations. If this step is successful, the programmer must rewrite the code according to the alignment results before using the simulator.

If the loop alignment is not applicable, CAMP can take two alternative approaches. First, it may partition the loop iteration space into independent execution sets. This method uses the minimum dependence distance in each dimension to divide the iteration space. The detail description of this method is given in [6]. Second, it may partition the loop with a selected synchronization method. In this approach, CAMP generates the partitioning ratio and the synchronization information, which helps the programmer in rewriting the code into parallel execution format.

A simulator has been developed for evaluating different partitioning methods. It generates the simulated execution time in helping the programmer to select the best partitioning strategy. If the original algorithm does not contain enough parallelism, the programmer may want to write a different algorithm. In the following sections, we will describe the process alignment, the bit–map synchronization scheme, and the partitioning of a loop with the bit–map method. All these mechanisms have been implemented in CAMP.

### 3. Process Alignment

The alignment of a program loop was first introduced by Padua for the purpose of increasing the number of independent partitions of a loop [7]. In this section, we will demonstrate the essence of this method by an example for minimizing the synchronization between iterations of a loop.

In the example of Figure 3 (a), there is a *constant dependence vector* [1,–1] between the second and the first statement. This constant dependence vector is due to the occurrence of A(i,j) in $S_2$ (producing a data) and A(i–1,j+1) in $S_1$ (consuming the data). We can adjust the subscripts of either occurrence of array A. Figure 3 (b) shows the adjustment of the first statement by transforming A(i–1,j+1) into A(i,j), and Figure 3 (c) shows the adjustment of the second statement by transforming A(i,j) into A(i–1,j+1). This code adjustment, called the ***process alignment***, has three significant changes:

(1) transformed subscripts in the adjusted statement;

(2) an additional IF test in each statement, and the extended loop bounds for the boundary iterations;

(3) the statement rearrangement to satisfy data dependences.

The details of an improved alignment algorithm is given in [6].

```
     DO i = 1, N
        DO j = 1, N
S1:         B(i,j) = A(i-1,j+1) × C(i) + D
S2:         A(i,j) = E(i,j) / F(i,j)
        ENDO
     ENDO
```

(a)

```
     DOALL i = 0, N
        DOALL j = 1, N+1
S2:         IF (1≤i≤N) AND (1≤j≤N) THEN
               A(i,j) = E(i,j) / F(i,j)
S1:         IF (1≤i+1≤N) AND (1≤j-1≤N) THEN
               B(i+1,j-1) = A(i,j) × C(i+1) + D
        ENDO
     ENDO
```

(b)

```
     DOALL i = 1, N+1
        DOALL j = 0, N
S2:         IF (1≤i-1≤N) AND (1≤j+1≤N) THEN
               A(i-1,j+1) = E(i-1,j+1) / F(i-1,j+1)
S1:         IF (1≤i≤N) AND (1≤j≤N) THEN
               B(i,j) = A(i-1,j+1) × C(i) + D
        ENDO
     ENDO
```

(c)

**Figure 3. A loop example and its alignment code**

After the process alignment, the dependence vector became [0,0]. Since the production and consumption of the data is now in the same iteration, the loop can be partitioned by iterations without synchronization between any partitioned set.

### 4. The Bit–map Synchronization Method

Synchronization of concurrent processes can be implemented through shared–variable or through message–passing [8]. There are different shared–variable synchronization methods such as the full/empty bit of the HEP [9], the fetch&add of the Ultracomputer [10], and the synchronization key of the Cedar [11]. In this section, we will describe a shared–variable method called the *bit–map* [12]. In the bit–map method, each synchronization data element has an attached *sync* field, and each memory operation contains a *mask* value. The data can be accessed only when the mask matches the sync. This way a proper referencing order to each data element can be maintained.

The bit–map synchronization method will be explained by using the QD–algorithm for calculating the distribution of the eigenvalues of a large positive definite tridiagonal matrix [13]. (Figure 4) This example contains several constant–dependence vectors: [1,0] for the dependences Q(i,j+1) → Q(i–1,j+1), and E(i,j) → E(i–1,j), [0,1] for Q(i,j+1) → Q(i,j), and [1,–1] for E(i,j) → E(i–1,j+1). We omit the dependence vector [0,0] since it indicates a dependence in the same iteration, which is always satisfied.

The data dependences must be preserved during a parallel execution. They can be preserved if and only if the proper order of memory operations to each memory location is maintained. The sequence of read and write operations to the same memory location, called a *referencing pattern* or RP, is denoted by $(R/W)_n$, ....., $(R/W)_i$, ....., $(R/W)_2$, $(R/W)_1$, where R/W indicates a read or a write operation and subscripts indicate the

```
     DO i = 2, N
        DO j = 0, N-1
S1:         Q(i,j+1) = Q(i-1,j+1) + E(i-1,j+1) - E(i,j)
S2:         E(i,j) = Q(i-1,j+1) × E(i-1,j) ÷ Q(i,j)
        ENDO
     ENDO
```

**Figure 4. The QD–algorithm**

sequence number. This sequence of memory operations is sorted by the indices of the multiple nested loops in which the memory location is referenced. If the memory location is referenced more than once in the same iteration, the referencing pattern will be ordered by the statement number first and inside each statement from right to left. In other word, the referencing pattern is defined by the sequential execution of the loop.

In our loop example, the RP of each read/write variable can be determined by sorting the constants in the first subscript of all the occurrences of that variable in a decreasing order. If more than one occurrence has the same constant in the first subscript, we sort them by the second subscript, and so on. If the constants in all subscripts of several occurrences are equal, then we sort the occurrences based on their sequential execution order. However, for the boundary data elements, some of the operations in a RP may not be performed.

The bit–map synchronization method preserves the RPs even if different processors in a multiprocessor system reference the same memory location out of order. Each synchronized variable has a *Sync* field and a *Data* field. The sync field contains a number of Full/Empty (F/E) bits. Each bit is used independently in the same way as the F/E bit in the HEP machine. The sync field and the data field can be implemented as two separate words in the memory to eliminate extra hardware cost. Every synchronization instruction has a *Mask* field which contains information for both *testing and modifying* bits in the sync field. We define two distinct synchronization instructions: SREAD and SWRITE, which can be expressed as:

$SREAD/SWRITE, Address, Mask.

The Address indicates the shared memory location where the accessed data and its associated sync value can be found. These instructions are implemented by the following *indivisible* sequence of microoperations, where *n* represents the length of the Mask and Sync fields and registers represent temporary storages in each processor.

For the SREAD instruction:

$$\bigwedge_{i=1}^{n} (\text{Mask}_i \text{ OR Sync}_i): \quad \text{Register} \leftarrow \text{Memory(Address)}$$
$$\text{Sync} \leftarrow \text{Mask AND Sync}$$

For the SWRITE instruction:

$$\neg\,(\bigvee_{i=1}^{n} (\text{Mask}_i \text{ AND Sync}_i)): \quad \text{Memory(Address)} \leftarrow \text{Register}$$
$$\text{Sync} \leftarrow \text{Mask OR Sync}$$

Generally speaking, the SREAD instruction tests whether certain bits of the sync field are '1' by ORing the sync field with the mask field and fetching data if the result is all '1's. After data is fetched, tested bits are cleared by ANDing the sync field with the mask. On the other hand, the SWRITE instruction tests whether one or more bits of the sync field are '0' by ANDing the sync field with the mask field and storing data only if the result is all '0's. After data is stored, tested bits are set by ORing the sync field with the mask. Only logical operations are needed in both instructions. Other types of synchronization instructions such as test without modifying the sync, can be easily implemented.

Using SREAD and SWRITE to enforce RPs allows consecutive reads to proceed in any order. In general, each read or write needs one bit in the sync field. If that bit is '1', memory location can be read but not written into, and vice versa. However, since any order of consecutive reads is allowed, the write following these reads needs a number of bits to detect if all the reads have been finished. The number of bits is equal to the number of reads ahead of the write. Whenever a read is successful, its corresponding bit is updated. Note that if there is only one write in the RP, this write can proceed when all the bits for reads become '0'.

For loops with constant dependence vectors, the number of operations in a RP of a read/write variable is equal to the number of occurrences of that variable. In general, each operation requires a bit in the sync field. Therefore, using a separate word for the sync field should be sufficient to synchronize any constant–dependence loop. Figure 5 (a) shows the parallel code of

```
DOALL i =2, N
   DOALL j = 0, N–1
S1:   $Q(i,j+1)/1111 = $Q(i–1,j+1)/1011 +
                  $E(i–1,j+1)/1011 – $E(i,j)/1100
S2:   $E(i,j)/1111 = $Q(i–1,j+1)/0111 ×
                  $E(i–1,j)/0111 ÷ $Q(i,j)/1101
   ENDO
ENDO
```

(a)

| Arrays | RPs | | | | Initial Syncs |
|---|---|---|---|---|---|
| Q(1,1..64) | $R_4$ | $R_3$ | – | – | 1 1 0 1 |
| Q(2..64,0) | – | – | $R_2$ | – | 0 0 1 1 |
| Q(2..63,1..63) | $R_4$ | $R_3$ | $R_2$ | $W_1$ | 0 0 0 0 |
| Q(2..63,64) | $R_4$ | $R_3$ | – | $W_1$ | 0 0 0 0 |
| Q(64,1..63) | – | – | $R_2$ | $W_1$ | 0 0 0 0 |
| Q(64,64) | – | – | – | $W_1$ | 0 0 0 0 |
| E(1,0) | $R_4$ | – | – | – | 1 0 1 0 |
| E(1,1..63) | $R_4$ | $R_3$ | – | – | 1 1 1 0 |
| E(1..63,64) | – | $R_3$ | – | – | 0 1 1 0 |
| E(2..63,0) | $R_4$ | – | $W_2$ | $R_1$ | 0 0 1 1 |
| E(2..63,1..63) | $R_4$ | $R_3$ | $W_2$ | $R_1$ | 0 0 1 1 |
| E(64,0..63) | – | – | $W_2$ | $R_1$ | 0 0 1 1 |

(b)

**Figure 5. (a) The parallel QD–algorithm**
**(b) The RPs and syncs**

the QD–algorithm using the bit–map method. With N = 64, the RPs and syncs of Q and E arrays are shown in Figure 5 (b).

The rule of selecting the initial mask and sync values for this example is very simple. The mask value for a read operation has '0's on the corresponding position for this read as well as on the position for the following write operation. On the other hand, the mask value for a write operation is equal to all '1's. The initial sync value of each memory location can be determined by its RP. The bit position for the first (the rightmost) operation, such as $R_2$ of Q(2..64,0) is enable, i.e. '1' for a read, '0' for write. All other operations are disable. If there are consecutive reads from the first operation, then all these reads are enable. The algorithm of finding the masks and syncs for general cases was given in [12].

Thus, the bit–map synchronization transforms a sequential loop into a parallel one by removing the control dependence and preserving the data dependence. All iterations can be executed in parallel like in any other data–flow model of computation.

## 5. Program Partitioning on Multiprocessors

Using the bit–map method, the QD–algorithm can be executed in parallel. The next issue is how to run this parallel code on a limited number of processors. Two factors must be considered when partitioning a loop into processes: *amount of parallelism* and *memory access and synchronization overhead.* The best partition should exploit the maximum parallelism with the lowest overhead. However, this ideal case is usually impossible to achieve. The partitioning strategy is a tradeoff between these two contradictory goals.

In partitioning program loops, we make two restrictions. First, each iteration of the loop is an indivisible unit of execution, executed sequentially by one processor. Second, the loop can only be partitioned into regular patterns, such as by row and/or by column in a two–dimensional space. Although partitioning at arithmetic operation level and/or partitioning along a wave–front [14] may exploit more parallelism, it introduces unacceptable coding complexity and performance overhead. We propose two ways of partitioning the loop: *block* and *interleaved* partitioning.

We now define a program loop and our partitioning schemes. A *program loop* is denoted by $L = (I_1, I_2, \ldots, I_n)\ (S_1, S_2, \ldots, S_s)$, where $I_i$ is an index variable of the i–th nested level of the loop, $1 \leq i \leq n$; and $S_j$ is an assignment or a conditional statement, $1 \leq j \leq s$. Each index variable consists of three attributes $I_i(\tilde{a}_i, \tilde{b}_i, \tilde{c}_i)$, where $\tilde{a}_i$ is the initial value, $\tilde{b}_i$ is the boundary value, and $\tilde{c}_i$ is increment value of $I_i$. In a loop $L = (I_1, I_2, \ldots, I_n)\ (S_1, S_2, \ldots, S_s)$, assume that dimension $I_i$ is partitioned into p sets. The j–th set of the *block* partition is defined to have:

the initial value $\tilde{a}_{i,j} = \tilde{a}_i + \left\lfloor x_i \times (j-1)/p \right\rfloor \times \tilde{c}_i$
the boundary value $\tilde{b}_{i,j} = \tilde{a}_i + (\left\lfloor x_i \times j/p \right\rfloor - 1) \times \tilde{c}_i$
the increment value $\tilde{c}_{i,j} = \tilde{c}_i$
where: $x_i = \left\lfloor (\tilde{b}_i - \tilde{a}_i + \tilde{c}_i)/\tilde{c}_i \right\rfloor$

while the j–th set of the *interleaved* partition has:

the initial value $\tilde{a}_{i,j} = \tilde{a}_i + (j - 1) \times \tilde{c}_i$
the boundary value $\tilde{b}_{i,j} = \tilde{b}_i$
the increment value $\tilde{c}_{i,j} = p \times \tilde{c}_i$

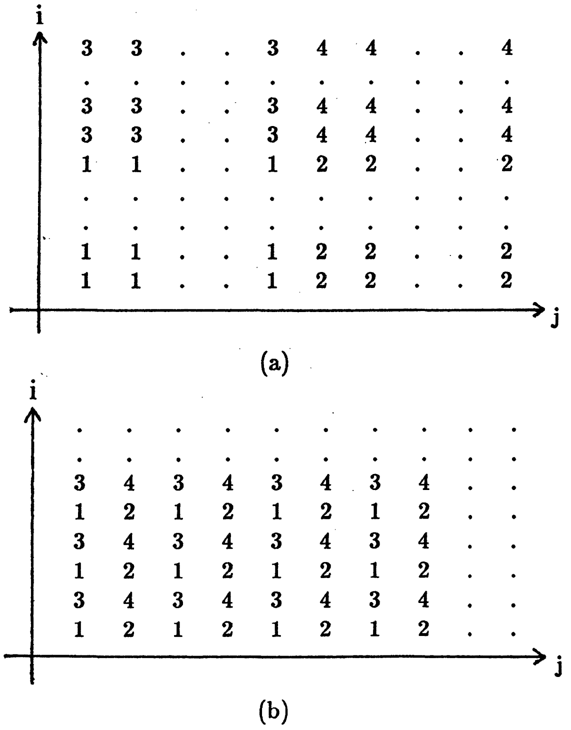
The block and the interleaved partitioning schemes are initiated by two different considerations. The former tries to minimize memory access and synchronization overhead since synchronization is only perform at the block boundary. The latter tries to maximize the parallelism by interleaving iterations among processors so that all processors can start approximately at the same time.

### 5.1. Partitioning with Different Aspect Ratios

The best way of exploiting parallelism of the recurrence loop example is to execute the loop according to the wave–front direction. On the other hand, the easiest way of partitioning the loop is based on the lexicographical order of the index set. The former method introduces unacceptable overhead while the latter method may not utilize all the parallelism in the loop [6].

We use a general partitioning method which divides each level $i$ of a nested set of loops into different number of partitions, $n_i$. Either the block or the interleaved partitioning scheme can be used in dividing the loop. The proportion of the number of partitions on each level is called the *aspect ratio* (AR). If 4 processors are assigned to execute the QD–algorithm, then three ARs ($n_1$:$n_2$) are possible: 4:1, 2:2, and 1:4. Figure 6 (a) shows the block partition and the processor assignment of the QD–algorithm with AR = 2:2, while Figure 6 (b) shows the interleaved partition, where 1, 2, 3, and 4 indicate four different processors. Note that the AR of 1:4 with the interleaved partition is equivalent to partitioning with the lexicographical order.

There is no extra coding complexity using the aspect ratio of the block and the interleaved partitioning methods. However, the total combination of ARs for a given number of processors $p$ grows with the nested levels of the loop. In a doubly–nested loop, the number of ARs is equal to $(\log_2 p + 1)$, while in a triply–nested loop, the number of ARs is equal to $((\log_2 p + 1)^2 / 2 + (\log_2 p + 1) / 2)$, where we assume the number of processors and the number of partitions are all powers of 2, and $n_i \leq p$.



```
3 3 . . 3 4 4 . . 4
. . . . . . . . . .
3 3 . . 3 4 4 . . 4
3 3 . . 3 4 4 . . 4
1 1 . . 1 2 2 . . 2
. . . . . . . . . .
. . . . . . . . . .
1 1 . . 1 2 2 . . 2
1 1 . . 1 2 2 . . 2
```

(a)

```
. . . . . . . . . .
. . . . . . . . . .
3 4 3 4 3 4 3 4 . .
1 2 1 2 1 2 1 2 . .
3 4 3 4 3 4 3 4 . .
1 2 1 2 1 2 1 2 . .
3 4 3 4 3 4 3 4 . .
1 2 1 2 1 2 1 2 . .
```

(b)

**Figure 6. (a) The block and (b) The interleaved partitioning with AR = 2:2**

### 5.2. Data Storage Scheme

Parallelism imposed by the data dependence is crucial to the program partitioning. In order to select the best AR, we need to consider another factor: memory access and synchronization overhead. Three types of memory accesses: *local access, local synchronized access,* and *network synchronized access* can be specified for testing the QD–algorithm in a distributed–shared memory multiprocessor system (Figure 7). The local access provides the fastest access time. Data with the exception of read–only data can be localized only if it is not accessed by any other processor. The local memory coherence problem is thus avoided. Data, read and written by more than one processor, can only be accessed through the synchronization instructions described in section 4.

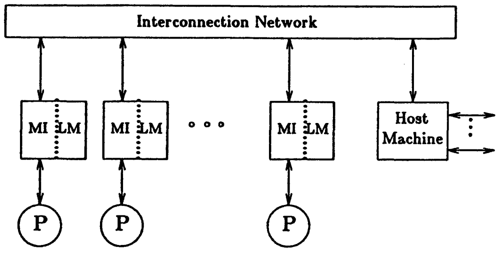


**Figure 7. The model of a multiprocessor system**

Because of the delay in the interconnection network, the synchronized access to any other memory module takes much longer than the synchronized access to the local memory. The data storage scheme is very important in reducing the number of network accesses. For instance, if 4 processors execute the QD–algorithm, with AR = 1:4, and the interleaved partition is used, then $P_2$ will execute iterations (*, 1) and (*, 5), where * stands for all iterations of the corresponding level of the loop. In each iteration, $P_2$ reads the Q array three times (Q(i–1,j+1), Q(i–1,j+1), and Q(i,j)), and writes once (Q(i,j+1)). If we allocate Q(*,1) and Q(*,5) to the local memory of $P_2$, then, only one access to the Q array will be to the local memory. On the other hand, if we allocate Q(*,2) and Q(*,6) to the local memory of $P_2$, then three out of four accesses to the Q array will access to the local memory. Thus, data should be allocated to the shared memory modules according to the allocation of iterations (i.e. the AR) and the subscript expressions of the accessed variable.

In the block partitioning, the type of memory access for each variable may vary from one iteration to another. If the iteration is not located on the block boundary, all the memory accesses in that iteration are local. Otherwise, the memory access can be either a local synchronized or a network synchronized access. The data storage scheme for the interleaved partition is applicable to the block partition.

### 5.3. Aspect Ratio Selection

Because the number of ARs in a three dimensional loop is proportional to $(\log_2 p)^2$, it is very time–consuming to simulate all the ARs for any given number of processors. A heuristic can be used for selecting the best ARs.

The sequential execution time of each iteration of a loop can be computed directly. For example, the starting and ending time for each memory access and arithmetic operation of the QD–algorithm are shown in Figure 8. In it we assume every memory access takes the same amount of time (M), and every arithmetic operation takes time (A). The *time–distance* for each dependence is computed as $t_g - t_n$, where $t_g$ is the data generation time, and $t_n$ is the time that the data is needed. In this example, the time–distance for Q(i,j+1) → Q(i–1,j+1) is 4M+2A; Q(i,j+1) → Q(i–1,j+1) is 0; E(i,j) → E(i–1,j) is 3M+2A; Q(i,j+1) → Q(i,j) is -(2M+A); and E(i,j) → E(i–1,j+1) is 7M+4A. The maximum value of the time–distance is selected for each dependence vector. Those are 4M+2A for [1,0], -(2M+A) for [0,1], and 7M+4A for [1,–1]. If the time distance is less than or equal to 0, it indicates there is no delay caused by the dependence. Therefore, the dependence vector can be omitted. Generally speaking, the time–distance indicates

| Operations | Starting Time | Ending Time |
|---|---|---|
| Fetch Q(i-1,j+1) | 0 | M |
| Fetch E(i-1,j+1) | M | 2M |
| Add | 2M | 2M+A |
| Fetch E(i,j) | 2M+A | 3M+A |
| Subtract | 3M+A | 3M+2A |
| Store Q(i,j+1) | 3M+2A | 4M+2A |
| Fetch Q(i-1,j+1) | 4M+2A | 5M+2A |
| Fetch E(i-1,j) | 5M+2A | 6M+2A |
| Multiply | 6M+2A | 6M+3A |
| Fetch Q(i,j) | 6M+3A | 7M+4A |
| Divide | 7M+3A | 7M+4A |
| Store E(i,j) | 7M+4A | 8M+4A |

M = Memory access time  A = Arithmetic operation time

**Figure 8. The sequential execution of an iteration of the QD–algorithm**

the 'delay' between two iterations with the data dependence between them. We discussed several rules in [6] for reducing this time–distance.

We now describe the algorithm which uses the time–distance for estimating the execution time of a particular AR. The algorithm is demand–driven. First, the execution time of the last iteration of the loop is demanded. This triggers evaluation of the execution time of two groups of iterations. The first group contains those iterations which produce data for the last iteration. The second group contains the iteration which must be executed right before the last iteration because of the sequential execution imposed in the same processor. The first group of iterations can be calculated from the data dependence vectors while the iteration of the second group is determined by the AR and the partitioning scheme. These triggered iterations, in turn, require the execution time of other iterations and so on. This process continues until the iteration space is exhausted or the iteration space boundary is reached [6].

In this heuristic, every memory operation has a constant access time. This assumption is valid for the interleaved partitioning method, since each occurrence of a variable has the same type of memory access for every iteration and can be determined according to the data storage scheme. However, the type of memory access varies for each variable in the block partition. The evaluation algorithm assumes all the memory accesses are local for the block partition. The results from this evaluation indicate the parallelism exploited by different ARs. Next, the number of different types of memory accesses for every AR can be computed. These numbers determine the memory latency. Two ARs are selected for the block partition. One exploits the best parallelism and the other has the lowest memory latency. Both selected ARs along with the best AR of the interleaved partition will run through the simulator to determine the best partitioning strategy.

## 6. Conclusion

We exercised the QD–algorithm with loop bound N = 64 through CAMP. Our simulator has two options for estimating the delay for the synchronization accesses [6]. In option 1, local synchronized access takes 2 units of time and network synchronized access takes 6 units of time. In option 2, local and network synchronized accesses take 3 and 9 units of time respectively.

The QD–algorithm has only one independent execution set and the alignment method is not applicable. Therefore, the partitioning with the bit–map method was selected. The ARs for different number of processors were determined by the evaluation heuristic described in section 5. The simulation results along with the selected ARs are shown in Table 1 and illustrated in Figure 9 in which INT represents the interleaved partition, BKM represents the block partition with minimum memory latency, and BKP represents the block partition with maximum parallelism. We can see that when the number of processors is small, the block partition which maximizes parallelism provides the fastest execution time. However, when the number of processors is greater than 16, the interleaved partition gives the best performance. The block partition which minimizes memory

The sequential execution time = 45056

| Number of Processors | | | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| Low overhead | INT | Time | 29441 | 14765 | 7391 | 3696 | 1841 |
| | | ARs | 1:4 | 1:8 | 1:16 | 1:32 | 1:64 |
| | BKM | Time | 36686 | 19848 | 19906 | 11450 | 11455 |
| | | ARs | 2:2 | 2:4 | 4:4 | 4:8 | 8:8 |
| | BKP | Time | 17282 | 9846 | 6103 | 4194 | 1914 |
| | | ARs | 1:4 | 1:8 | 1:16 | 1:32 | 1:64 |
| High overhead | INT | Time | 41595 | 20842 | 10434 | 5216 | 2595 |
| | | ARs | 1:4 | 1:8 | 1:16 | 1:32 | 1:64 |
| | BKM | Time | 39251 | 22245 | 23273 | 14707 | 15249 |
| | | ARs | 2:2 | 2:4 | 4:4 | 4:8 | 8:8 |
| | BKP | Time | 18567 | 11162 | 7503 | 5759 | 2673 |
| | | ARs | 1:4 | 1:8 | 1:16 | 1:32 | 1:64 |

**Table 1. The simulated execution time of the QD–algorithm**

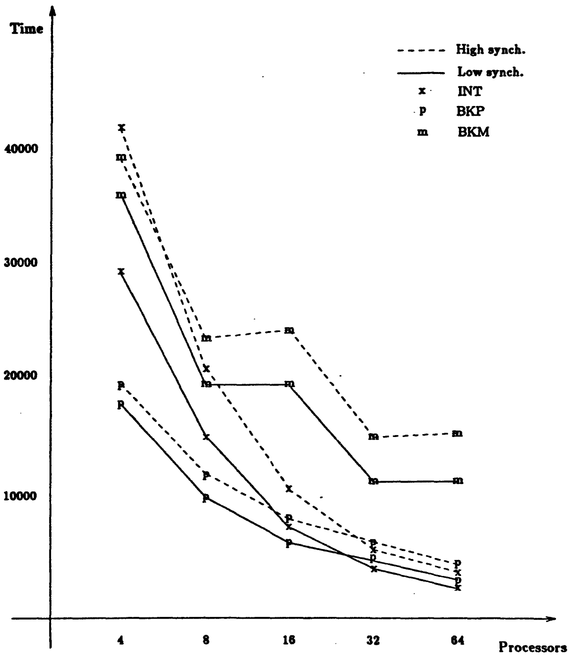


**Figure 9. The plot of the execution time**

and synchronization overhead has the worst performance.

CAMP is written in Pascal and runs on a VAX-11/780 under 4.2 BSD UNIX[a] operating system. Presently, the program-development tool uses only a limited number of partitioning algorithms and synchronization primitives. The future plans include additional partitioning schemes especially for the partitioning of the program segments without loops, shown as the dashed boxes in Figure 2. Other useful synchronization primitives such as test&set, full/empty bit etc. could be added to CAMP. Other future work should also include the partitioning and synchronization of programs for message-passage multiprocessors [3].

## 7. References

[1] D. D. Gajski, and J.-K. Peir, "The Essential Issues in Multiprocessor Systems," *IEEE Computer*, Vol. 18, No. 6, June 1985, pp. 9–27.

[2] P. Wilson, "OCCAM Architecture Eases System Design – Part I," *Computer Design*, Nov. 1983, pp. 107–115. 2NP, UK, 1983.

[3] C. L. Seitz, "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22–33.

[4] D. A. Padua, D. J. Kuck, and D. L. Lawrie, "High Speed Multiprocessor and Compilation Techniques," *IEEE Trans. Computers*, Vol. C–29, No. 9, Sep. 1980, pp. 763–776.

[5] Ptran Project at IBM Research, Yorktown Heights, NY, 1986. (Personal communication with R. Cytron.)

[6] J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," Ph. D. Thesis, Univ. of Illinois at Urb.-Champ., Rept. No. UIUCDCS-R-86-1259, March 1986.

[7] D. A. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph. D. Thesis, Univ. of Illinois at Urb.-Champ., Rept. No. UIUCDCS-R-79-990, Nov. 1979.

[8] G. R. Andrews, and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol. 15, No. 1, Mar. 1983, pp. 3–43.

[9] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," Society of Photo-Optical Instrumentation Engineers, Vol. 298, Real-time Signal Processing IV, Aug. 1981, pp. 241–248.

[10] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, "The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, Vol. C–32, No. 2, Feb. 1983, pp. 175–189.

[11] C-Q Zhu and P-C Yew, "A Synchronization Scheme and Its Applications for Large Scale Multiprocessor Systems," *Proc. Conf. on Distributed Computing Systems*, San Francisco, May, 1984, pp. 486–491.

[12] J.-K. Peir, and D. D. Gajski, "Data Flow Execution of Fortran Loops," *Proc. 1st Int'l Conf. on Supercomputing Systems*, St. Petersburg, FL, Dec. 16–20, 1985.

[13] H. Rutishauser, "Stabile Sonderfalle des Quotienten-Differengen Algorithmus," *Numer. Math.*, Vol. 5, 1963, pp. 95–112.

[14] J. A. Fortes, and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," *Proc. 1984 Int'l Conf. on Parallel Processing*, Aug. 1984, pp. 322–328.

[a] UNIX is a trade mark of Bell Laboratories.

# THE DESIGN OF A QUEUE-BASED VECTOR SUPERCOMPUTER

Honesty C. Young[1]
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

**Abstract-- To meet increasing demands for computing power, systems must exploit parallelism at all levels. A balance between the processing speeds of a machine's subsystems is critical for overall system performance. In this paper, we describe a queue-based vector supercomputer (QVC) which reduces the fraction of work that the scalar mode must process.**

**The proposed queue-based vector supercomputer has the following characteristics: (1) two level control is used to support out-of-order instruction initiation, (2) multiple classes of registers is provided, (3) one valid bit per vector element is included to exploit flexible chaining, (4) a very short vector startup time (one clock period) is achieved, (5) branch instructions are used only for implementing high level language control structures, (6) elements of a queue may be read repeatedly, and (7) vectorization has been generalized.**

**Several hard-to-vectorize loops from the Livermore Loops are used as the benchmark programs. The preliminary study suggests that the proposed queue-based vector architecture is a cost-effective way to implement a supercomputer for array-oriented programs.**

## 1. Introduction

Queues have been included in many architectures [3,5,20,23,24], intended primarily for high performance numerical applications in scalar mode. Most commercial supercomputers [18,21,7] have some kind of vector buffers (registers) (the Cyber 205 [15] is one exception: vector operations are carried out in memory-memory mode). In this paper, we describe a queue-based architecture for which the notion of a vector register is simply a special case.

This paper is organized as follows. Section 2 motivates the vector processors. Section 3 describes the desired features for a vector processor and presents the proposed queue-based vector supercomputer. Section 4 compares the proposed architecture with the Cray X-MP [7] using some hard-to-vectorized Livermore Loops [17,21] as the benchmark programs. Section 5 has the conclusions.

## 2. Motivations of Vector Supercomputers

It has been repeatedly observed [2,4,11,22,24,28] that for a system with two processing speed modes, the effective execution speed is limited by the slower mode, unless the fraction of workload that has to be done in this mode is nominal. Put differently, a good balance between different processing speeds is critical to the overall system performance. In the numerical supercomputer environment, the *high-speed mode* corresponds to vector operations, the *low-speed mode* to scalar instructions. We must push the limits on both ends to meet ever-increasing computing requirements. An ideal high performance system must not only have a fast scalar mode, but also a vector mode that reduces the fraction of work that must be processed in the scalar mode. The latter can be achieved by architectural innovations which facilitate the vectorization of most array operations.

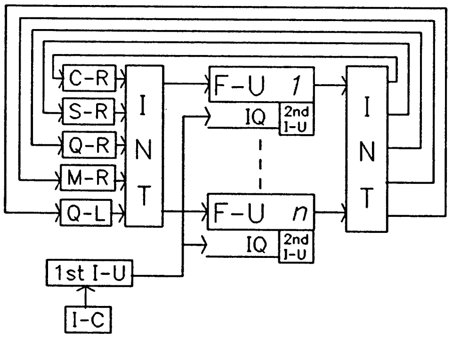
Recently, several queue-based processors [3,5,11,20,24] have been proposed to achieve fast scalar mode. The advantages of a queue-based scalar system have been publicized elsewhere [24,11, 29]. Many array-oriented programs cannot be vectorized, partly because the underlying vector computers do not provide the adequate primitive instructions to generate vector code for many commonly seen vector operations, such as vector inner product. The major goal of the proposed two level control, queue-based vector supercomputer (QVC) is to ease the task of the vectorization of array-oriented programs. Thus, QVC reduces the fraction of work that must be processed in the scalar mode.

The major considerations behind the vector instructions are the following:

1. Flynn bottleneck [9]. The performance of a computer system is sometimes limited by the instruction initiation rate. An issue unit is capable of initiating at most a fixed number (normally, one) of instructions every clock period. If each instruction does too simple an operation, this Flynn limit may become a bottleneck. A decoupled architecture [5,11,20,24] does increase the bandwidth of the instruction issue capability by including multiple (two in all the cases listed above) issue units. Another way of increasing the effective issue bandwidth is to include powerful instructions. Each powerful instruction is able to do many operations. Hence, issuing one powerful instruction can achieve the same effect as issuing multiple relatively simple ones. The issue condition of a powerful instruction, however, does not have to be complex. In the QVC, each powerful instruction (vector instruction) is just a repetition of a simple scalar operation. These vector instructions still enjoy the simple issue condition property of its scalar counterpart.
2. Out-of-order execution. The floating point unit of the IBM 360/91 [26] and the scoreboard of the CDC 6600 [25] are examples of hardware methods to achieve out-of-order execution. Out-of-order execution is an essential part of vector instructions with different vector length and the simultaneous execution of vector and scalar instructions. The vector supercomputer proposed in this study, however, provides out-of-order initiation of scalar instructions (almost) free, in the sense that a scalar instruction is just a special case of vector instruction, *i.e.*, a scalar is just a vector of length one.

## 3. Proposed Architecture

In this paper, we will emphasize the design of the vector mode operation of the QVC. Many functional units are included in the QVC. Each of them is implemented by a linear (fully segmented) pipeline [13] which eliminates the necessity of doing pipeline reorganization and avoids most structural hazards (collisions). Each function unit may either perform a single task (*e.g.*, floating point addition) or be capable of executing several operations (*e.g.*, logical operations). There is an issue unit associated with each functional unit, which checks for the issue conditions. Except for the Load/ Store instructions, all operations are carried out in a register-register fashion. The availability of the operands determines the issue conditions of the register-register instructions. For performance measurement purposes, we compare the QVC with a single CPU of a Cray X-MP having an identical set of function units (see section 4).



The terms used in the figure are explained below:
C-R: Constant Registers
S-R: Scalar Registers
Q-R: Queue Registers
M-R: Mask Registers
Q-L: Queue Length Registers
I-C: Instruction Cache
INT: Interconnection Network
1st I-U: First Level Issue Unit
2nd I-U: Second Level Issue Unit
IQ: Instruction Queue
F-U *i*: the *i*-th Functional Unit

**Figure 1: The Organization of a Processor with *n* Functional Units.**

[1] The work described here was done while this author was at the University of Wisconsin-Madison.

0190-3918/86/0000/0483 $01.00 © 1986 IEEE

The desired vector properties include flexible memory-referencing instructions, arithmetic/logic operations, vector editing instructions and the easily-vectorizable properties. By easily-vectorizable properties, we mean that vector operations are applicable to the programs which are not vectorizable on the current supercomputers. Regular arrays should be treated effectively, while sparse arrays must be handled with reasonable efficiency. We suggest that all the registers be shared by all functional units. We also want to be able to intermix vector instructions with scalar ones. For example, vector A is loaded into a queue-register by a vector load instruction, but entries of A are consumed by different scalar instructions. Figure 1 illustrates the QVC with *n* functional units.

### 3.1. Vector Instruction Format

A vector instruction has the following format:

```
repeat $m$ $n$
    inst$_1$;
    ...
    inst$_m$;
```

The semantic of the above repeat statement is to execute the instruction group, inst$_1$ to inst$_m$, *n* times, where each *inst* represents a simple scalar instruction and all instructions within an instruction group use the same functional unit. The size of an instruction group (*i.e.*, *m*) is a compile time constant, while the length of a vector (*i.e.*, number of iterations) is specified by one of the queue length (QL) registers and the "repeat" instruction selects the appropriate queue length register. If the length of the vector is known to be 1, at compile time, the repeat statement is not needed. This is how we use the scalar mode of this architecture.

### 3.2. Operand Register (Queue) Sets

In addition to the status registers (*e.g.*, the queue length register just mentioned), four sets of operand registers are included:

1. Constant scalar registers. Rationale for these registers is provided elsewhere [29]
2. Scalar registers. They are just like general purpose registers in most computer architectures.
3. Queue registers. Queue registers are a set of registers, each of which is a queue. They are rationalized in the following section.
4. Mask register. Mask registers are similar to the queue registers in that they are also implemented as queues. The major difference is that each element of a mask register has only one bit.

Any one of the scalar registers or the queue registers can be used as a destination/source register of memory accessing instructions. The memory interface is treated as one functional unit.

The mechanism to handle different data dependency hazards (RAW, WAR, WAW) for queue/mask registers is detailed elsewhere [29].

### 3.3. Queue Registers

. Queue registers are a set of registers, each of which is a queue. Only limited elements (*i.e.*, the head and tail of the queue) are accessible from outside. A queue register is the hardware implementation of a "stream". Thus operands in a queue register must be accessed in a predefined order. Each queue register is implemented as an array of registers. There is a *valid bit* associated with each element of the queue register to indicate the data availability of the designated element. Two pointers are associated with each queue register. They point to the head and the tail of that particular queue. A queue may have three possible states--Full, Empty, and Normal--which can be checked by examining the appropriate valid bits. When a queue is full, an attempt to put an operand onto that queue will be blocked.

Similarly, reading from an empty queue is also blocked. One of the advantages in having one valid bit per element is that a queue register can temporarily hold a vector which is longer than the queue size, as long as there are other instructions that take operands out of the same queue register. This valid bit per element scheme also makes chaining more flexible, because the speed of the consumer and the producer does not have to be identical, *i.e.*, chaining can be performed in an asynchronous fashion. A similar scheme is used by the Hitachi S-810 [19] to allow flexible chaining. The QVC has the flavor of dataflow machines [8] except that the issue unit checks for data availability, rather than the data availability "firing" the operation. Thus, function units can operate in parallel as much as possible. We call this organization a *control-driven dataflow* computer architecture. This control-driven dataflow scheme is more flexible than that of most traditional supercomputers in the sense that data availability is checked on an element by element basis, even for vector operands. Hence, the operation of chaining is more flexibility than in the case where data availability is checked at the vector level. On the other hand, the overhead of the dataflow approach [10] is avoided.

One of the characteristics of queues is that an element of a queue is discarded after it has been read. There are cases in which we want to use an operand (a scalar or a vector) more than once. Therefore, we include three access modes in reading from a queue: (a) *destructive* mode; (b) *non-destructive* mode; and (c) *circular* mode. In destructive mode, a read operation removes the first element from the queue register. In non-destructive mode, the first element of a queue remains after a read operation, *i.e.*, a queue can be (non-destructively) read many times while its contents are not changed. In circular mode, the entire queue remains unchanged after each element has been accessed. Sometimes, the access mode is determined at run time based on the (partial) result of an instruction (*e.g.*, compare two queue registers). Since a queue register is implemented as an array of elements with two pointers, a different access mode simply suggests a slightly different interpretation in updating the pointers.

### 3.4. Two-Level Control

A two-level instruction initiating scheme is adapted in this design. The global first level issue unit decides branch outcomes and sends non-branch instructions to the proper instruction queues which are associated with the second level issue unit. For each function unit, there is an instruction queue and a second level issue unit. Each second level instruction issue unit initiates instructions in the order they were sent to the instruction queue (*i.e.*, the original machine code textual order). Instructions in different second level instruction queues, however, may be initiated in a different order than that in which they passed the first level issue unit. The second level issue unit checks for data availability and issues instructions accordingly. The implication of this two-level initiating scheme is that it takes two clock periods to issue a non-branch instruction, *i.e.*, the work of instruction initiating is divided into two stages of the pipeline. However, this scheme does not introduce extensive delay. The delay of the additional clock period in the decoding logic slows down a section of straight line code by at most one clock period. Since the branch decision is made by the first level issue unit, there is no execution time penalty even across basic blocks, as long as the expressions that participate in the branch decision are evaluated earlier, which can normally be done by properly scheduling the code (see, for example, [29]). Thus, in the best case, the penalty introduced by this two-level control scheme is one clock period per program. The worst case penalty, however, is one clock period per basic block. On the other hand, the first level issue unit sends a partially decoded instruction to the instruction queue associated with the appropriate function unit. In other words, the decoding is done in two stages (the first level issue unit and the second level issue unit). If the instruction decoding time of the original design determines the clock period, the two level decoding scheme may imply a faster clock rate, because the instruction decoding is done in two pipeline stages. The startup time of a vector instruction is just the time for the first level issue unit to send a "repeat" statement to the appropriate second level issue unit, which normally takes one clock period.

The first level issue unit is blocked only because of either (a) a branch instruction, which can be alleviated by the prepare-to-branch [11] scheme, or (b) a full instruction queue of the corresponding functional unit. The functionality of this proposed two level instruction initiation scheme is similar to Tomasulo's algorithm [26] except that instructions that use the same functional unit are executed in program order. In [29] a comparison was made to compare the performance of the two-level scheme with that of Tomasulo's algorithm using the example given by Weiss and Smith [27] in studying various instruction initiation schemes. In this example, the two-level scheme and the Tomasulo's algorithm perform equally well. The former, however, avoids the potentially expensive associative search used in the Tomasulo's algorithm.

### 3.5. Vector Load/Store

Studies [4,16] have shown that the relative ratio of vector load/store with unit stride, non-unit stride, and random access is about 70% : 20% : 10%. Thus, the vector load/store instruction has to support accessing random elements of an array. A repeated load/store with autoincrement can be used to access elements separated by a constant stride. A random access is normally represented by an additional level of indirection, *i.e.*, the addresses of the needed elements are put in another array. This random access is supported by the following two vector instructions: (a) put addresses of needed elements onto a queue register (call it %Q*q*); and (b) do a vector load using addresses in %Q*q*.

Simultaneous vector load/store operations may cause undesirable memory overlap hazard conditions (that is, read before write or write before read). One solution for eliminating such hazards is to have a smart memory system detect the conflicts. Another way is to have the software determine the cases where the hazards may occur and assure sequential execution whenever necessary.

### 3.6. Comparison Instructions

The contents of a mask register can be loaded by doing an element-by-element vector comparison. Given two vectors **A** and **B** of length *n*, the result of the vector comparison is stored in **M**. The semantics are that $M_i$ gets the comparison result of $A_i$ and $B_i$. The result of a comparison, then, is treated as an ordinary operand. We term such comparison a *logical* one in the sense the result of the comparison, rather than one of the operands, is returned. This scheme is used by some supercomputers (*e.g.*, Cray-1) to set up a mask register.

The scalar logical comparison is useful in evaluating Boolean expressions without using branches. *i.e.*, logical instructions such as AND, rather than branches, are used to evaluate complex Boolean expressions. Branches are needed only to construct high level program structures.

There are cases where one of the operands involved in the comparison, rather than the logical comparison result, is needed. We propose another set of comparisons called *contents* comparisons. This corresponds to the if-expression in some programming languages. The (vector) contents comparison is generally useful in coping with sorting-related problems.

### 3.7. Vector Editing Instructions

Different vector editing instructions can be realized by the repeat statements (vector instructions) with slightly different accessing modes described earlier. It is also possible to include scalars within a vector editing instruction. Merge (combining two vectors into one) and split (the opposite of merge) can also be carried out by two repeat-statements with proper adjustments to the mask register.

The different vector editing functions available in the commercial supercomputers are nicely surveyed by Hwang and Briggs [12]. We make no attempt to specify all vector editing instructions completely in this paper.

### 3.8. Intrinsic Functions

Some vector operations, such as vector sum (the summation of all elements in a vector), are inherently difficult to vectorize. One possible way to cope with these hard-to-vectorize operations is for the compiler to generate scalar instructions [6] which implies relatively low performance. Another possible approach is to include a large set of vector macro instructions and hope that most hard-to-vectorize operations are covered by the vector macros [15]. In the QVC, most vector macros can be composed by several of its vector instructions. The realization of the following vector macros can be found in [29]: (a) vector sum; (b) inner product; (c) linear recurrence; (d) maximum/minimum; (e) search; and (f) sorting.

We believe that most, if not all, programs can take advantage of this queue-based vector supercomputer. However, advanced compiler techniques are required to fully utilize the potential parallelism provided by this proposed architecture.

## 4. Performance Evaluation

In this section, we compare the performance of a single CPU of a Cray X-MP [7] with a similar architecture with the proposed extension. In particular, there are the same number of vector registers in the Cray X-MP as the queue registers in the QVC. The identical set of function units are included in both the Cray X-MP and the QVC. For QVC, we assume two sets of execution times for the functional units. One is that the execution time (in terms of clock period) of each functional unit in the QVC is identical to that of the Cray X-MP. The other is to include an additional delay of one clock period when the data is sent through the interconnection of the QVC (*i.e.*, two additional clock periods are added to a register-register operation). One last point is that we ignore the delays because of memory bank conflict, *i.e.*, we assume the data are nicely distributed in the memory banks and all memory access requests are serviced in a predetermined amount of time (*i.e.*, 14 clock periods to do a scalar load).

Riganati and Schneck [21] summarize the supercomputer performance reported on Livermore Loops. The execution speed of loops 4, 5, 6, 11, 13, and 14 are slower than other loops. In fact, the performance of these 6 loops is below 50 mega *flops* (floating point operations per second). In other words, if the Livermore Loops are used as the workload, the execution time of the aforementioned loops dominates the total execution time. Loop 5 and loop 6 are essentially the same provided the compiler does the induction variable analysis [1]. Therefore, we use loops 4, 5, 11, 13, and 14 as the benchmark programs.

| Loop | Cray | Cray(SP) | QVC | QVC+ |
|---|---|---|---|---|
| 4 | 2778 | 1569 | 1028 | 1380 |
| 5 | 18129 | 15148 | 12963 | 16948 |
| 11 | 20261 | 12275 | 6009 | 8008 |
| 13 | 13090 | 12066 | 1386 | 1410 |
| 14 | 22736 | 20636 | 3645 | 3753 |

Table 1. The Execution Times for Different Configurations.

In Table 1, the execution times (in clock periods) of the 5 loops on 4 different configurations are shown. There are two columns associated with the Cray X-MP. The "Cray" column has the execution times of the loops while the scalar scheduling is done within the loop boundary -- we do not employ "loop carry-over" optimizations, such as loop unrolling, other than the ones specified in the source Fortran code. The "Cray(SP)" column has the execution times where the software pipelining[2] (using the algorithm described in [29]) technique is applied. The "QVC" column has the execution time of this queue-based vector computer. The "QVC+" column has the execution time of the QVC, while an additional clock period is needed for an operand to "penetrate" through the QVC's interconnection network. We apply some known optimization techniques, by hand, to the Cray code. In particular, we try to partially vectorize the loops whenever appropriate. For example, the floating point multiplication of loop 4 (X(LV)*Y(J)) is carried out using vector instructions. Also, vectors **Y** & **Z** of loop 5, vector **Y** of loop 11, vectors **P(1,IP)** & **P(2,IP)** of loop 13, and vectors **GRD**, **VX**, & **XX** of loop 14 are *block-preloaded* into the appropriate vector registers using the vector load instructions. That is, we use the vector registers as data buffers between the processor and the memory system whenever appropriate.

For loop 4, the QVC outperforms the Cray X-MP primarily because of the following:

- Mixed mode operation. In th Cray X-MP, scalar instructions must be used to get an element from a vector register. Three instructions (a vector to scalar move, an increment to the index register, and a branch instruction) are needed for each result stored in a vector register. In other words, the overhead to get an element out of a vector register is in the order of 10 clock periods. (The execution time of the vector register to scalar register data move instruction [076*ijk*] is 4 clock periods.)
- Branches. Because we cannot express a linear recurrence in a vector form on the Cray X-MP, branch instructions are used to carry out the loop. The overhead may be as many as 5 clock periods per iteration (for an in-buffer condition).
- Chaining. The first element of the vector operation results, stored in a vector register, cannot be taken from a vector register until the vector operation has completed. The chaining on the mixed mode operation is not flexible enough.

For loops 5 and 11, the difference in the execution time comes from the following:

- Mixed mode operation. Though loop 5 is hard to vectorize due to the first order linear recurrence, arrays **Y** and **Z** should be able to be "block-preloaded" into some vector registers. Because the vector mode and the scalar mode on the Cray X-MP are not readily compatible (*i.e.*, additional move instructions are needed to move each element from a vector register to a scalar register), the mixed mode operation can not be carried out efficiently.
- Branches.
- Store. In the Cray X-MP, a store instruction cannot be issued until the result is available. In the two level control scheme, however, the first level issue unit can issue a store instruction as long as the instruction queue associated with the store functional unit is not full. Therefore, instructions following the store instruction can be issued, by the first level issue unit, even before the operand for the store instruction has been computed. Thus less instruction blockage will occur in the two level control scheme than in the one level control scheme.
- Loop unrolling. The source code of loop 5 is unrolled three times. Because of the limited number of S registers in the Cray X-MP (8 of them), some instruction issuing blockages come from the static register assignment. As stated in [29] that queues provide the dynamic register assignment property. Thus, unnecessary data dependencies, owing to static register assignment, can be avoided.

Software pipelining, however, overlaps the time of the mixed mode operation (load from a vector register), the branches, and the store instruction blockages. Thus, for loops 4, 5 and 11, the performance improvement due to software pipelining is significant.

| Loops | QVC : Cray | QVC : Cray(SP) | QVC+ : Cray(SP) | QVC : QVC+ |
|---|---|---|---|---|
| 4 | 2.70 | 1.53 | 1.14 | 1.34 |
| 5 | 1.40 | 1.17 | 0.89 | 1.31 |
| 11 | 3.37 | 2.04 | 1.53 | 1.33 |
| 13 | 9.44 | 8.71 | 8.55 | 1.02 |
| 14 | 6.24 | 5.66 | 5.50 | 1.03 |

Table 2. The Relative Performance for Different Configurations.

[2] One simple kind of software pipelining is the pre-loading and post-storing of operands.

The discrepancy in the execution times of loops 13 and 14 comes from the following:

- Memory indirection load. Memory indirection load is not well supported in Cray X-MP.
- Scalar variable expansion [14]. A scalar variable can easily be expanded to a vector in the QVC by allocating such a scalar in a queue-register. By so doing, the variable from different iterations occupies different locations in a queue-register.

Two major constraints that also limit the performance of loops 13 and 14 on the QVC are:

- Number of functional units. The performance is bounded by the number of load and floating point addition pipes. This can be remedied expensively by adding functional units to the processor. The additional functional units, however, may slow the clock cycle down.
- Data conversion. It takes several instructions to do data format conversion between integer and floating point number. This, however, is not an essential limit of the architecture. If data conversion happens frequently, we can add a special data conversion pipe to the architecture.

In Table 2, we compare the relative performance for different configurations. Looking at the QVC+ : Cray(SP) column, we find that the Cray X-MP outperforms the QVC+ only for loop 5. The data dependency graph of loop 5 is very deep, *i.e.*, every instruction depends on the result from the previous instruction. In other words, the data dependency, rather than the issue unit, limits the execution speed. Thus, the QVC with additional delay runs slower than the Cray X-MP. In most cases, however, the advantage of the two level control and the queue-based vector register still outweighs the delay going through the interconnection. Looking at the QVC : QVC+ column, we find that for loops 4, 5, and 11, where the execution time is limited by the data dependencies, the performance penalty introduced by the one clock period delay going through the interconnection is about one third. (Recall that the execution times of a floating point addition and a floating point multiplication are 6 clock periods and 7 clock periods, respectively. The time to go through two interconnections [one on the input side, the other on the output side] is 2 clock periods, which is about one third of the execution times of the floating point operations mentioned above.) On the other hand, for loops 13 and 14, where the execution time is limited by the availability of the functional units, the performance penalty caused by the delay in the interconnection is between 2% and 3%.

For highly vectorizable loops (*e.g.*, loop 7), the performance is limited by the availability of the functional unit. In this case, the extra clock period delay going through the interconnection will slow down the entire system by only a small fraction.

## 5. Conclusions

This queue-based vector supercomputer supports out-of-order instruction initiation and flexible vector chaining. Its simple vector instruction format implies very short vector startup time (one clock period). This simple format, however, is very powerful to use with the queue registers and different access modes. As demonstrated earlier in this paper, many array-oriented programs are vectorizable under this proposed architecture. The destination non-blocking interconnection network avoids the unnecessary result bus conflicts in the case where a shared result bus is used for all functional units (in Cray X-MP, all results going to S registers share the same bus). Branch instructions are needed only to carry out the high level language control structures by introducing the logical/contents comparison. Access modes are provided so that it is possible to read the elements of a queue repeatedly in a variety of ways.

The preliminary study suggests that this proposed architecture is a cost-effective way to implement a processor for array-oriented programs. The major remaining problem is the design of compiler techniques to fully utilize the suggested features automatically.

### Acknowledgement

We would like to thank A.R. Pleszkun and J.E. Smith for many valuable and stimulating discussions. This work was sponsored in part by USA NSF under grants MCS-8202952 and MCS-8105904.

### Bibliography

[1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, Mass. 1986.

[2] Amdahl, Gene, "The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities", *Proceedings, AFIPS Spring Joint Computer Conference,* pp. 483-485, April, 1967.

[3] Brantley, William C., and Joseph Weiss, "FOM: A Fortran Optimized Machine--A High Performance, High Level Language Machine", *IBM Research Report RC 9640,* #40815, March, 1982.

[4] Bucher, Ingrid Y., "The Computational Speed of Supercomputers", *Proceedings, ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems,* pp. 151-165, August, 1983.

[5] Cohler, Edmund U., and James E. Storer, "Functionally Parallel Architecture for Array Processors", *IEEE Computer 14,* 9, pp. 28-36, September, 1981..

[6] Cray Research, Inc., *Cray-1 Computer Systems* S Series Mainframe Reference Manual (HR-0029), 1982.

[7] Cray Research, Inc., *Advanced Large-scale and High-speed Multiprocessor System for Scientific Applications,* Cray X-MP-4 Series, 1985.

[8] Dennis, Jack B., "Data Flow Supercomputers", *IEEE Computer 13,* 11, pp. 48-56, November, 1980..

[9] Flynn, Michael J., "Very High-Speed Computing Systems", *Proceedings of the IEEE 54,* 12, pp. 1901-1909, December, 1966.

[10] Gajski, D.D., D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages", *IEEE Computer 15,* 12, pp. 58-70, February, 1982.

[11] Goodman, James R., Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P.B. Schechter, and Honesty C. Young, "PIPE: a Decoupled Architecture for VLSI", *Proceedings, the 12th International Symposium on Computer Architecture,* pp. 20-27, June, 1985.

[12] Hwang, Kai, and Faye A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, Inc., New York, 1984.

[13] Kogge, Peter M, *The Architecture of Pipelined Computers,* McGraw-Hill, New York, 1981.

[14] Kuck, David J., Robert H. Kuhn, Bruce Leasure, and Michael Wolfe, "The Structure of an Advanced retargetable Vectorizer", *Proceedings, IEEE COMPSAC,* pp. 709-715, October, 1980..

[15] Lincoln, Neil R., "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer", *IEEE Transactions on Computers C-31,* 5, pp. 349-362, May, 1982.

[16] Matsurra, Toshihiko, Sachio Kamiya, and Masaaki Takiuchi, "Design Concept of the Facom VP Based on Extensive Analysis of Applications", *Proceedings, International Conference on Computer Design: VLSI in Computers,* pp. 232-237, October, 1984.

[17] McMahon, F.H., "Fortran CPU Performance Analysis", *Lawrence Livermore Laboratories,* 1972.

[18] Miura, Kenichi, and Keiichiro Uchida, "FACOM Vector Processor System: VP-100/VP-200", *Proceedings, NATO Advanced Research Workshop on High Speed Computing,* West Germany, June, 1983.

[19] Nagashima, Shigeo, and Yasuhiro Inagami, "Design Consideration for a High-Speed Vector Processor: The HITACHI S-810", *Proceedings, IEEE International Conference on Computer Design: VLSI in Computers,* pp. 238-243, October, 1984.

[20] Pleszkun, Andrew R., and Edward S. Davidson, "A Structured Memory Access Architecture", *Proceedings, IEEE International Conference on Parallel-Processing,* pp. 461-471, August, 1983.

[21] Riganati, John P., and Paul B. Schneck, "Supercomputing", *IEEE Computer 17,* 10, pp. 97-113, October, 1984..

[22] Rudsinski, L., and J. Worlton, "The Impact of Scalar Performance on Vector and Parallel Processors", *Proceedings, the Symposium on High Speed Computer and Algorithm Organization,* pp. 451-452, April, 1977.

[23] Smith, James E, Andrew R. Pleszkun, Randy H. Katz, and James R. Goodman, "PIPE: A High Performance VLSI Architecture", *Proceedings, IEEE International Workshop on Computer Systems Organization,* pp. 131-138, March, 1983.

[24] Smith, James E., "Decoupled Access/Execute Computer Architectures", *ACM Transactions on Computer Systems 2,* 4, pp. 289-308, November, 1984.

[25] Thornton, J.E., *Design of a Computer, The Control Data 6600,* Scott, Foresman and Co., Glenview, IL, 1970.

[26] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development 11,* 1, pp. 25-33, January, 1967. vol='1,'.

[27] Weiss, Shlomo, and James R. Smith, "Instruction Issue Logic in Pipelined Supercomputers", *IEEE Transactions on Computers C-33,* 11, pp. 1013-1022, November, 1984.

[28] Worlton, Jack, "Understanding Supercomputer Benchmarks", *Datamation,* pp. 121-130, September, 1984.

[29] Honesty C. Young, "Evaluation of a Decoupled Computer Architecture and the Design of a Vector Extension", *Ph.D. Dissertation, Computer Sciences Department,* University of Wisconsin-Madison, May, 1985; also available as *Technical Report #603, Computer Sciences Department,* University of Wisconsin-Madison, July, 1985.

# Data Synchronized Pipeline Architecture : pipelining in multiprocessor environments

Yvon JEGOU
André SEZNEC
IRISA, Campus de Beaulieu
35042 RENNES CEDEX
FRANCE

## ABSTRACT

To satisfy the growing need for computing power, a high degree of parallelism will be necessary in future supercomputers. Up to the late 70s, supercomputers were either multiprocessors (SIMD-MIMD) or pipelined monoprocessors. Future industrial realizations should combine these two levels of parallelism. In a multiprocessor, classical pipeline controls become inefficient because the interdependent behavior of the processing elements cannot be foreseen either at compile time or at decode time.

In this paper, we introduce a new model of pipeline architecture : the Data Synchronized Pipeline Architecture (DSPA). Based on an independent sequencing of the functional units, this model allows a high degree of parallelism in the pipeline, even in the case of unforeseeable behaviors of some ressource.

## I Introduction

Need for computing power seems unlimited in various scientific applications. During the last ten years, tremendous progress has been made in the domain of component integration. But todays' supercomputer clocks are of the same order of magnitude as those of supercomputers ten years ago. E.g. the clock of the Cray2 (1985) is only three times faster than the one of the Crayl (1976).

Requirements for performance have lead manufacturers to the design of parallel structures. The first industrial parallel supercomputers were pipeline processors (Crayl, CDC Cyber 205, .. ). Today, these pipeline computers can be considered as the state of the art in monoprocessor architecture. Since the late 1970's, a lot of multiprocessor projects have been initiated [5][6][14][7]. In future industrial realizations of these ambitious projects, the elementary processors will be pipeline processors. Great attention must be taken in the design of these pipeline processors.

In a pipeline computer, the execution of an instruction stream generates concurrent activities in several functional units ( FUs ). These FUs may be pipelined. Even when the FUs are not pipelined, the successive FUs crossed by the data form a macropipeline. Performance of the computer depends heavily on the overlapping of successive instructions and are bounded by the throughput of the instruction decoder. In the case of vector instructions, simple pipeline control is possible because of the regularity of the data and instruction streams. Optimization of the code can be done at compile time. For example, performance of the Crayl can be considered as near optimum on classical vector instructions. That is the reason why pipeline computers are generally refered as vector processors. Unfortunately, performance of existent pipeline computers on scalar instructions are not as good as one can expect. The overlapping of successive instructions cannot always be decided at compile time because memory access conflicts or hazards may arise at execution time. Because in a multiprocessor architecture some hardware is shared, the behavior of processing elements are interdependent. Moreover if decode time for vector instructions is small regardless of the occupation of the FUs, very fast algorithms have to be used for pipeline control in scalar mode, because the decoding delays become critical.

In section II, we recall the problems which exist on some classical pipeline computers. Then, in the following sections, we present a new model of pipeline architecture : the Data Synchronized Pipeline Architecture (DSPA). This model has been developed in order to improve the performance of pipeline computers on scalar code. We also point out that using this model, a distributed decode of instructions allows very fast sequencing and efficient overlap for both vector and scalar instructions even in the cases where memory conflicts or hazards may occur.

## II Existent pipeline models

### 1 Outlines

Today, a pipeline processor can be used as a monoprocessor ( Crayl, FPS164, Cray XMP-1, Fujitsu VP 200,.. ) or as an elementary processor in a multiprocessor system (Cray XMP-4, Cray2, BSP,..). The BSP [10] is a SIMD computer; its elementary processors (PEs) are pipelined. A set of vector instructions has been defined by the designers. These instructions are memory to memory instructions ( operands are read from memory and the result is stored in memory ). When the machine has been defined, the behavior of these instructions was studied and optimized - by introduction of delays between two functional units for example; reservation tables [1][9][8] are stored in the machine. The SIMD structure of BSP allowed global pipeline control for all PEs. In general, the distinct FUs do not have the same number of stages and many cases of memory conflicts may occur; even with a restricted number of vector instructions, the number of cases to be studied would increase exponentially with the number of operands involved by the instruction. In the BSP the number of cases to be studied remains quite limited; all the FUs have the same number of stages (the reservation table does not depend on the FUs but only on the number and the distribution of distinct FUs concerned with an instruction), and memory conflicts are almost avoided by a constant increment definition of a vector and the choice of a prime of memory banks [11][12]. On the BSP, a good overlapping of distinct iterations of the same vector instruction is performed; but there is no overlapping of the end of a vector instruction by the beginning of an other instruction : the pipeline must be filled at the beginning of an instruction, and emptied at the end. Memory to memory instruction may be quite efficient on vector operands, but scalar instructions overlapping has to be done with other techniques while the restricted choice of vector definition increases the ratio of scalar code (e.g. loop 3 proposed later must be executed in scalar mode).

In todays' multiprocessors, the behavior of a shared memory cannot be foreseen either at compile time, or at execution or decode time : it depends on the different processes being executed by the different elementary processors. Even when vector accesses are synchronized, memory conflicts may occur at anytime; memory behavior depends on the relations in the distribution of the vector elements, on the relations between two successive vector accesses and on the treatment of RAW (Read After Write) hazards [13] : memory to memory architecture are no adapted to high speed scalar execution and asynchronous PE controls.

We present the problems that remain on two existent pipelined monoprocessors.

## 2 Vector register machines

In register machines, the operands of an instruction are loaded in the functional units from registers and the results are stored in registers. In some pipeline computers, such as Cray1 [3], there are vector registers; a vector register may contain up to N words -in Cray1, N=64. A single instruction with vector register operands may generate up to N times the same operation on distinct data, results are then stored in a vector register. This approach seems very efficient on pure vector instructions.

Loop 1 :

```
        DO 1 I=1,N
      1 A(I)=B(I)+C(I)*D(I)
```

Only seven instructions are necessary to code the body of this loop for the Cray1.

```
Vector load C --> R1
Vector load D --> R2
R3 <-- R1*R2
Vector load B --> R4
R5 <-- R3 + R4
Vector store R5 --> A
Conditionnal jump
```

Time to decode such a loop is not critical : up to 256 accesses to memory are generated by these seven instructions and the Cray1 can decode one instruction by cycle.

Unfortunately accesses to memory on the Cray1 are performed in the order of the decode sequence; moreover no advance is taken on the decode sequence : the next instruction is decoded only when the present instruction is initiated. An instruction is initiated only when the presence of all its operands is guaranteed; as there is no hardware mean to verify if the $i^{th}$ element of a vector is present or not, to initiate a vector instruction, the following condition must be guaranteed :

*For every i, i cycles later the $i^{th}$ element will be present.*

These restrictions explain why 353 cycles are necessary to execute an iteration of loop 1 while the memory is only busy during 256 cycles. They also induce the same definition of a vector as in the BSP. The loop 2 is then treated in scalar mode :

Loop 2 :

```
        DO 2 I=1,N
      2 A(I)= B(H(I))+ C(I)
```

Cray1 is about ten times slower than the Fujitsu VP200 [15] for the execution of this loop; because vector registers of addresses can be computed, the Fujitsu VP200 executes this loop as a vector loop. But loop 3 is executed in scalar mode on both machines :

Loop 3 :

```
        DO 3 I=1,N
      3 A(I)= A(H(I))+ B(I)*C(I)
```

Possible hazards may occur on this loop; as H(2) may be equal to 1, A(1) has to be written before A(H(2)) is read ( at least the hardware should avoid reading A(H(2)) before writing A(1) if H(2)=1). The loop must be coded in scalar mode.

In scientific programs, some part of the code always remains scalar. If performance in scalar mode are slow compared to vector performance, the execution time of the residual sequential code may become predominant. On Cray1, for example, when loop 1 is executed in scalar mode, about 25 cycles are necessary to decode the loop body : the theoretical throughput of the memory is then more than six times higher than the real throughput!

Even in the case of a register machine where advance can be taken at the decode [16][17], it is very difficult to busy all the FUs in scalar mode : parallel decode is quite impossible for register machines. The decoder will always remain a bottleneck for performance in scalar mode for these machines : that is the reason why they are considered as vector machines.

## 3 Microcoded fully connected pipelined machines

In microcoded machines, an instruction word contains a subinstruction parcel for each of its FUs. When all operands are loaded from registers, the register throughput becomes tremendous and unrealistic. Operands for a FU may be directly taken on output buses of the FUs. On a fully connected machine, each output of a FU is connected to each input of the FUs; some connections are not so useful as others : e.g. the path between the output of the floating point adder and the input of the address unit is very rarely used; such paths may be suppressed (fig. 1).

The FPS 164 of Floating Point Systems [2] may be considered as a "nearly" fully connected pipeline processor. Each of its FUs can initiate an instruction on every cycle. As the decoder can decode an instruction word on every cycle, the FPS 164 can run at full speed on scalar code.

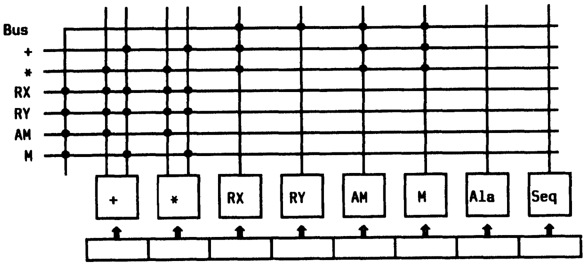
In an instruction word, the subinstruction parcel for each FU is always located at the same place. For example, the subinstruction for the adder can always be decomposed in three subparcels :

- Origin of the left operand, (i.e. the bus on which it is present)
- Origin of the right operand
- Codeoperation

The width of an instruction word for the FPS 164 is only 64 bits; it contains an order for each FU (see [2]). Controls of the crossbar network are given by the origins of the operands for the different FUs; synchronization of the FUs is explicit : the orders decoded in a cycle are initiated at the next cycle. A datum must be present on the output bus of a FU at the foreseen cycle. Delays to cross most of the FUs are constant and the date of their exit from a FU can be foreseen. Unfortunately, unforeseen memory bank conflicts may arise on an interleaved or shared memory; in this case, one solution consists of stopping the clock for the other FUs.

It is very difficult to produce efficient code for the FPS 164 because of the explicit synchronization by the microcode between the FUs. All the instructions are scalar i.e. an instruction initiates at most one order by FU. This is not critical because an instruction word can busy all the FUs; but automatic production of dense code is very difficult. Moreover efficient code cannot be produced without unrolling loops. E.g., on FPS 164, the loop body of the inner dot product may be coded on a single instruction loop preceeded by about ten instructions to fill the pipeline and about ten instructions to empty the pipeline. Unfortunately, no general compiling techniques can be used to unroll loops. To allow high performance on the FPS 164, Floating Point Systems proposes a library of frequently used functions and procedures which have been hand coded (BLAS). Performance are increased by a factor of two when coding LINPACK using BLAS instead of coding it in classical FORTRAN [4]. When possible hazards may occur (e.g. loop 3), loops cannot be unrolled : reads and writes cannot be exchanged by software.

We have pointed out the major limitations of some existent pipeline machines. Performance of existent vector register machines such as Crayl drop dramatically as soon as "good" conditions disappear, (scalar code, memory bank conflicts, scatter-gather, possible hazards ..). In scalar mode, performance are limited by the decoder : at each cycle, only one instruction can be decoded and then only one FU can be activated. Moreover the decoder cannot make any advances and no overtaking of writes by reads is allowed on memory. Fully connected microcoded pipeline processors can run at full speed in scalar mode, but they present insurmountable difficulties for efficient code production. For both models of machines, parallelism between the FUs is generated by software : possible hazards (e.g. loop 3) induce scalar execution, overlapping of the end of a loop by the beginning of the next loop is rarely performed and is always poor.



**A data interconnection scheme in the FPS 64**
**fig. 1**

In section III, we present a new model of pipeline machine. As in FPS 164, an instruction may contain a subinstruction parcel for each FU and, as the execution of these subinstructions are totally independent, a distributed decode can be done; another parallel decoder is also proposed. Vector and scalar instructions are available for this model of machine; a good overlapping of distinct instructions is possible whether they belong to same iteration of a loop or not and even when they do not belong to the same loop.

## III The Data Synchronized Pipelined Architecture ( DSPA )

### 1 goals

The basic problems which have lead us to define a new model of pipeline machines have been developed in the previous section.

As we have already pointed out, todays' pipeline processors must be designed to be included as elementary processors in a multiprocessor computer : MIMD and SIMD computers. Using fully connected microcoded pipeline processors as elementary processors in such architectures would be unrealistic : when a datum cannot be delivered by the shared memory, the clock would be stopped for the other FUs of the elementary processor; this is unacceptable and has lead us to express a first condition that we want to be satisfied by the design of a pipeline processor :

**Condition 1 :** *The sequencings of the different FUs of a pipeline processor have to be independent*

When this condition is satisfied, a delay in the delivery of a datum for a FU will not necessarily block the other FUs. Also, an instruction is not necessarily initiated at the cycle where its operands are produced and the production of the operands of an operation do not have to be synchronized.

In section I, we have seen that the decoder becomes a bottleneck for performance in scalar mode for vector register pipelined machines, so we propose a second condition :

**Condition 2 :** *Parallel decode is necessary on pipeline processors.*

We have also noticed that performance on the FPS 164 depends heavily on hand coded libraries. We wish to design machines able to reach correct performance on a very large set of scientific programs. This has led us to a third condition :

**Condition 3 :** *Natural code generation (by a compiler ) must produce efficient code.*

Two hardware tools seem to favor this goal : possible advance on decode and a RAW detection mechanism to enable reads to overtake writes on memory.

## 2 The DSPA model

We have rejected register pipeline machines because of the tremendous throughput demand on the registers and on the decoder to achieve performance in scalar mode. A Data Synchronized Pipeline processor is a "nearly" fully connected pipeline processor in which a FIFO (First In, First Out) queue is associated with each crosspoint on the interconnection scheme between the producers -outputs of the FUs- and the consumers - entries of the FUs. Except for the sequencer, each FU also has a FIFO queue of instructions (fig. 2). When a datum flows out from a producer P, it is stored in a FIFO queue associated with a pair (P,C) where C is a consumer i.e an input of a FU F : the datum is stored in this FIFO queue until F is ready to treat it (fig.3 ). An instruction for a FU may be decomposed in three parcels :

-Origins of the operands -i.e. names of producers; the FIFO queue associated to the path between the producer and the input of the FU is refered.

-Destination of the result -i.e. the name of a consumer.

-Codeoperation

The sequencing of the FUs is very simple.

We detail here the sequencing of the adder (fig. 4)

*1.If the FIFO queue of instructions is empty then GOTO 1.*

*2.Load the instruction in the adder's sequencer and decode it.*

*3.If one of the origin FIFO queues is empty then GOTO 3.*

*4.Load the operands; initiate the operation.*

*5.Store the result in the referenced FIFO queue **and** GOTO 1.*

These five steps may be pipelined :

-Step 5 is always overlapped by the other steps;

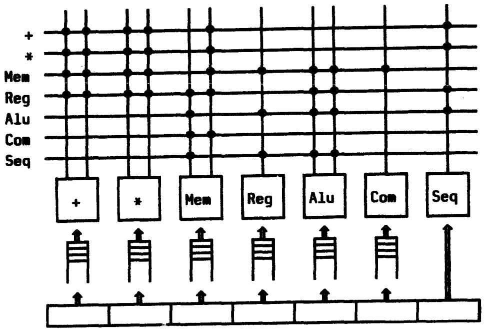
-The next instruction -when existing in the FIFO queue- may be decoded during step 3 and 4;

-Operands may always be loaded and the operation initiated : this operation can be aborted if the operands are not valid.

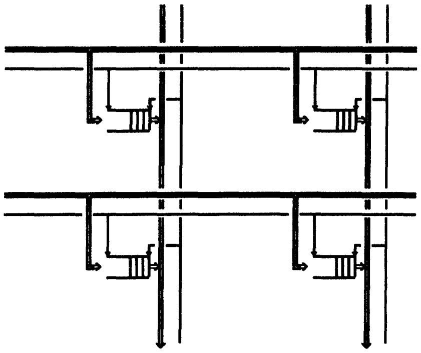
## 3 Deterministic execution

In classical machines, instructions are executed in the same order they are decoded : this guarantees the unique interpretation of a sequence of instructions. This constraint is respected on the FPS 164 : instructions are immediatly executed.

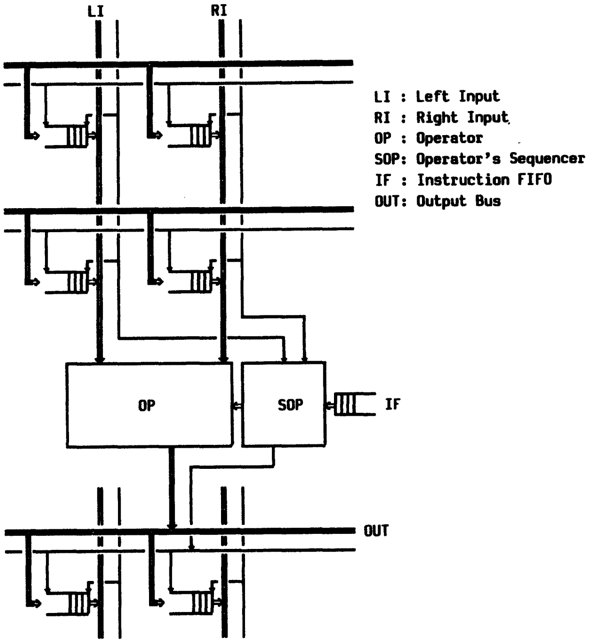
In our model, the FUs are synchronized by the data; an instruction may wait for its operands during a few cycles : there is no reason to decode the memory load of an operand for an addition before decoding the addition; if the adder is not busy, it can wait for the datum. The only significant order in this model of pipeline machines is the order of the data which enter the same FIFO queue.



An example of DSPA integration scheme
fig. 2



Crosspoint FIFOs
fig. 3



A typical DSPA functionnal unit
fig. 4

A FIFO queue is associated with only one producing functional unit. The instruction FIFO queue of a FU guarantees that this FU initiates its operations in the same order it receives its instructions : it is reasonable to impose the constraint that the results flow out from the FU in the same order that their production is initiated : in most cases, the delay to cross a FU is constant.

This constraint guarantees the unicity of the signification of a code sequence.

## 4 DSPA interleaved memory

In the DSPA model, the memory FU can be considered as a producing unit ( reads ) and also as a consuming unit ( writes ). The DSPA model imposes the constraint that the results of the reads flow out from the memory in the same order that the reads have been decoded. On an interleaved memory, this may dramatically decrease the throughput of the memory if the reads are really done in the same order they are decoded; let us suppose a four memory bank interleaved memory whose banks are busy during four cycles of a read and let us consider the following distribution of read requests :

read 1 and 2 on bank 0
read 3 and 4 on bank 1
read 5 and 6 on bank 2
read 7 and 8 on bank 3

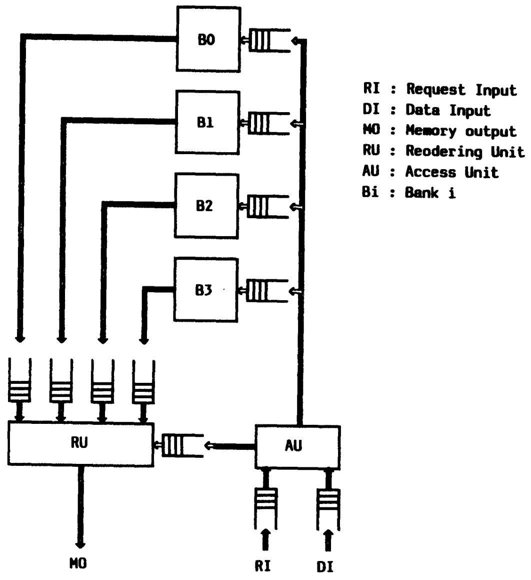
The results flow out from the banks in good order if the reads are initiated in the following order :

read 1 is initiated on cycle 1
read 2 is initiated on cycle 5
(bank 0 is busy on cycles 2, 3, 4)
read 3 is initiated on cycle 6
read 4 is initiated on cycle 10
read 5 is initiated on cycle 11
read 6 is initiated on cycle 15
read 7 is initiated on cycle 16
read 8 is initiated on cycle 20

In this extreme case, the real throughput of the memory is only two fifths of the theoretical throughput.

We propose a design of an interleaved memory which increases the real throughput of the memory and which respects the DSPA philosophy (fig.5). Addresses (and data for the write instructions) flow out from the access unit (AU) in good order and enter the FIFO queue of requests for the desired banks. Bank i loads its requests from the FIFO queue, then read data are stored in a FIFO queue associated with bank i. Then the reordering unit (RU) reads the data on the desired FIFO queues. The data flows out from the memory unit on the memory output bus (MO) in the desired order.

One can easily verify that when the sequence of reads of the previous example is repeated, the asymptotic throughput of the memory reaches the theoretical throughput. We will see later that a RAW detection hardware mechanism is necessary in the memory of a pipeline mechanism; this mechanism can be implemented in the AU (global detection) as well as in each bank (local detection).



A DSPA-compatible interleaved memory
fig. 5

## 5 Instruction decode

### 5.1 A distributed decode

We have already pointed out that the relative order of two instructions for two distinct FUs does not matter. Many solutions may be imagined for the global decoder. For example, as in the FPS 164, an instruction may contain a subinstruction parcel for the distinct FUs : these subinstructions may always be located at the same places in the instruction word and then can be directly routed to the distinct instruction FIFO queues of the FUs :

*The global decoder is only a bus.*

When, in an arbitrary sequence of instructions the most frequently used FU receives N instructions, the whole sequence can be coded on only N instruction words; the relative order of the subinstructions for the same FU is the only significant relation.

Very dense code is obtained without unrolling the loops as for the FPS 164 for example. Moreover, in the FPS 164 case, a lot of instruction words are necessary to fill and to empty the pipeline : in our model, these instructions are not necesssary; all the iterations of a loop have the same code. Let us suppose a DSPA processor which FUs have the same characteristics as in the FPS 164. Natural code generation will produce only one instruction by FU for the loop body of the inner dot product : without unrolling loops, a compiler will generate a single instruction word loop for this machine.

Nevertheless, we do not think that this solution of the decode is the only possible : 64 bits are necessary to code an instruction word for the FPS 164, more information must be given in subinstructions in our model, the width of an instruction word may result in an instruction memory or cache that is too expensive.

### 5.2 Decoder throughput

We discuss here why the throughput of the decoder may be more limited.

#### 5.2.1 Flexibility of the model

In the design of the FPS 164 and other machine as Crayl, all the FUs are assumed to have the same basic cycle : this allows initiation of a new instruction on each FU at each cycle. This simplifies the microcoding of the FPS 164 and the sequencing of the Crayl.

But this cannot be considered as a good balance between the throughput of the FUs : most of the classical algorithms require at the minimum an average of one memory access per floating point operation. On the FPS 164, this difficulty has lead to the introduction of an auxiliary memory having the same throughput as the main memory. In the design of the successor of the Crayl, the Cray-XMP, two pipelined channels are used to simultaneously access the memory : they can be considered as distinct FUs which have to respect very strong constraints.

Our model processor can support distinct basic cycles for the distinct FUs; e.g. the basic cycles of the floating point operators may be two times longer than the basic cycle of the memory. Moreover the delay in which a FU delivers a result may vary; we have only imposed the constraint that results flow out from the FU in the order their productions have been initiated.

#### 5.2.2 Vector instructions

On the FPS 164, vector instructions cannot exist because of the explicit synchronizations by the microcode at each cycle. In the DSPA model, no difficulties exist to prevent vector instructions. Executing a vector instruction of length k consists of executing the same scalar instruction k times. As in vector register machines, the physical support of the FIFO imposes a maximum vector instruction length. In some cases as in loop 4, vector instructions increase the performance by removing strict dependences between the data in the pipeline :

Loop 4 :

```
        DO 4 I=1,N
      4 A(I)= B(I) + C(I) + D(I)
```

In the body of this loop, two instructions concern the adder; the second addition cannot be initiated before the end of the first one. If the delay to cross the adder is k cycles –seven cycles in Crayl– no addition can be initiated during k-1 cycles. A vector instruction of length k works on k independent flows of data : in loop 4, the adder should then be busied when the memory throughput is sufficient. Our model allows coding many loops as vector loops; loops 1 and 2 are coded with vector instructions; the following loop 5 is also coded as a vector loop.

Loop 5 :

```
        DO 5 I=1,N
      5 A(H(I))= B(P(I))*C(Q(I))
```

The loop 3 may be coded by mixing vector instructions and scalar instructions; accesses to H, B, C, the multiplication and also the addition may be coded as vector instructions. In order to detect possible RAW hazards, the two accesses to A must be scalar coded : an internal scalar loop has to be coded in the global loop.

Let us suppose that the maximum vector length of a vector instruction is 8, the body of loop 3 can be coded by the following sequence :(1)

```
V M: ?VpRead H ↣ Ma
  M: ?VpRead A ↣ *l
  M: ?VpRead B ↣ *r
  *: ?V*(M,M)  ↣ +l
  +: ?V+(*,M)  ↣ Md
S M: ?SiRead A (M) ↣ +r
  M: ?SpWrite A (+)
 Seq: C←C-1,if C>0 then GOTO S
 Seq: N←N-1,if N>0 then (VL←min(N,8);C←VL;GOTO V)
```

The floating point additions have been extracted from the internal scalar loop : these instructions are kept in the adder's instruction FIFO queue until the operands are received.

The flexibility of the vector instruction definition associated with the possibility to access vector operands in scalar mode and to concatenate scalar operands to form vector operands increases the ratio of vector instructions in DSPA programming.

#### 5.2.3 Towards another proposition for distributed decode

In a loop body of scientific programs, some FUs are not as heavily used as others; in the previous examples, no registers were used. When an instruction word may contain a subinstruction parcel for each FU, the ratio of empty instructions parcels in a sequence of code may be tremendous. In most of the cases, there is only one instruction for the sequencer in a loop body; on the other hand, in all classical examples the memory is the critical resource : it does not seem natural to decode an instruction for the sequencer on every cycle, but it may be critical for the memory. Our experiments have shown that for a machine with well designed FUs, nearly half of the instructions are memory accesses. We recall that we consider that the memory FU computes postincremented addresses and indirect based addresses.

(1) M is the memory, Ma is the address input of the memory, +l is the right input of the adder .. . ?V (resp. ?S) refers to vector (resp. scalar) instructions. Memory instructions are p-access (post -incremented) or i-accesses (indirect). p-access refers to a descriptor containinq a base address A and an increment R. A vector p-access generates VL addresses of the form A + iR and leaves the descriptor with a new base address A + VL R; a scalar p-access sends the address A to the memory and assigns the value A + R to A. Address for a i-access is computed by adding an internal base address to a value read on Ma.

Then we have managed to decode only a few instructions at the same cycle. These instructions must be applied to distinct FUs. It seems that decoding two instructions during a memory cycle represents a good balance between the decoder throughput and the possible performance of the machine. The global decode is very simple : names of the FUs involved by the instructions are decoded and then the two instructions are routed to the correct instruction FIFO queues.

## IV RAW hazards

The DSPA model respects the conditions 1 and 2. When producing code for a processor of DSPA family, the compiler can forget that this processor may be an elementary processor of a multiprocessor computer. The code may be generated in the same manner it would have been generated for a monoprocessor : the real behavior of the shared memory can be ignored. This has allowed us to treat loop 2 and 5 as vector loops : this cannot be done on the Cray1 because the behavior of the memory cannot be foreseen at compile time.

Independency of the FUs also allows parallel decode -two solutions have been proposed, others may be imagined. Code generation may be relatively simple : as the instruction flows for two distinct FUs are completely independent, the compaction of a linear sequence of code is very easy.

Unfortunately advance on decode is not very useful when the effective accesses to memory have to be done in the order of their decode; in most of the cases, a loop body begins by a read and ends by a write to memory. This prevents the overlapping of the end of an iteration of the loop by the beginning of the next iteration unless some mechanism is used to allow passing the writes by the reads (when the read address and the write address are distinct). Software mechanisms have been proposed in the past : two distinct flows of memory accesses may be generated and reads of a flow overtake the writes of the other flow. Many examples where these solutions are not efficient can be imagined.

```
Loop 6 :
      DO 6 I=1,N
      DO 10 J=1,I
   10 A(J)=A(J)+B(I,J)
      6 CONTINUE
```

When executing the internal loop, it is very interesting to pass the write of A(J) by the reads of A(J+1), A(J+2),.. . But when I becomes small, one cannot ensure that A(1) has been written by the previous external iteration. The only software solution to prevent hazards in the execution of this loop is to empty the pipeline after each iteration of the internal loop.

We think that a hardware detection of RAW hazards on memory has to be implemented in a pipeline processor; in loop 3, there are possible hazards on memory, no software means can be imagined to treat this case. Using one of the two models of decode we have presented, the decode of this loop programmed with vector instructions and an internal scalar loop will take no more than 20 cycles : the decode will not be a bottleneck -40 accesses to the memory are done. If a hardware detection of RAW hazards is done, one can hope that when real hazards don't occur, performance will only be limited by the memory throughput.

Another advantage of a hardware detection of hazards is to enable the reaching of correct performance on unoptimized code : performance on loop 1 when scalar coded can be equal to vector performance, even where start-up delays are longer if an interleaved memory is used. One can hope for correct performance on long loop bodies. To detect RAW hazards on a DSPA computer, the memory access instruction and the corresponding address are extracted from the FIFO queues in the decode order; but write addresses are saved in some associative memory until corresponding data is present; read addresses are checked against the associative memory.

## V Some limitations of the DSPA model

### 1 Registers

In the examples we have detailed, the registers have never been used. In the case where a datum is used two times or more in an algorithm, it must be explicitly saved in a register and each operation requiring this datum as an operand will cost two instructions : the instruction to initiate the operation and the read of the register. The most important case of two uses of the same operand is the complex multiplication : a solution may consist in implementing a complex mode on the multiplier.

### 2 A blocking machine

In a DSPA processor, an instruction waits for its operands; this instruction may have been decoded before the decode of the production of its operands and if these productions are never done ( a bad generation of code may lead to this extremity ), the FU will never be activated as in example 7 :

```
 7 +:  (M,M) ↦ +1
   Seq:  GOTO 7
```

On the other hand, data may be produced and never consumed :

Example 8 :

```
 8 M:  pRead X ↦ +1
   Seq:  GOTO 8
```

Correct codes must be written : when a datum is produced by a FU, it must be consumed by an other; when a data has to be consumed, its production has to be guaranteed.

We define a correct unbreakable sequence of code (CUS) as :

*No external jump inside the sequence is possible unless to its first instruction.*

*No jump out of the sequence is possible unless from the last instruction.*

*Each datum produced in the sequence is consumed in the sequence.*

*Each datum consumed in the sequence is produced in the sequence.*

The loop body of example 3 is a CUS, but the internal scalar loop is not a CUS : data which are consumed are not produced in the loop.

The following loop body of an inner dot product is not a CUS :

```
S  M:  pRead A ↣ *l
   M:  pRead B ↣ *r
   *:  *(M,M) ↣ +r
   +:  +(+,*) ↣ +l
  Seq:  C← C-1, if C>0 then GOTO S
```

When executing this loop, the first left operand for the adder must have been previously produced in order to initiate the pipeline; this corresponds to the initialization of the sum – note that p subsums may be computed by p initializations.

The following condition must be respected to guarantee the execution of all the decoded instruction :

*Each sequence of code must be included in a CUS.*

There are no problems for a compiler to produce correct code (e.g. a solution may consist in treating DO loops as CUS, addresses of jumps delimiting distinct CUSs), but hand optimizations of the code may be dangerous : if the previous condition is not respected, the machine will be blocked. But this is not a real problem : machines on which badly coded programs produce the desired results don't really exist.

## VI Conclusion

We have presented an original model of architecture for pipeline processors. In this model, the decoder does not remain a bottleneck for performance in scalar mode as in existent pipeline processors : independency of the FUs and distributed decode of their instructions allow very fast sequencing. Synchronization of the FUs is done by the data. Such a processor can be integrated in synchronous multiprocessor architectures as well as in classical MIMD structures.

Generating code for machines based on our model will be very easy; natural sequential code can first be generated –instruction by instruction–, then independent codes for the distinct FUs can be extracted. For classical pipeline computers, complex reordering algorithms are applied to code at compile time to ensure performance at execution time; these algorithms may also be applied to the distinct codes for the FUs but this is less necessary than for classical architectures. When possible hazards may occur on memory or registers, there is no mean to optimize code for classical pipeline computers; on a DSPA architecture only real hazards may degrade performance.

At present, we are studying a real implementation of a pipeline processor of this family. Many solutions can be adopted : buses can be shared by several FUs, this sharing may be static or dynamic ... Great attention has to be taken in the design of the FUs; for example, in this paper we have already exposed some characteristics of the memory FU (computation of the addresses by the FUs, need of a RAW hazard detection mechanism). Another important point to be optimized in the design of the machine is the interconnection scheme : connections can be quite expensive in this mode because a FIFO queue is associated to each crosspoint on this interconnection scheme; infrequently used paths between the FUs can be suppressed; several FIFO queues may be implemented on the same support.

In the theoretical model, arbitrarily large FIFO queues are used. In a real machine, the sizes of the FIFO queues are limited by the physical support which limits the length of vector instructions. All these questions will be discussed in future papers.

**Bibliography**

[1] J.L.Baer, *Computer Systems Architecture,* Computer Science Press, 1980

[2] A.E.Charlesworth, "An approach to scientific array processing : the architectural design of the AP120B/FPS 164 Family" Computer, september 1981

[3] Cray-1 Computer Systems, *Hardware Reference Manual,* Cray Research Inc., Chippewa Falls, WI 1979

[4] J.J.Dongarra, "Performance of various computers using standard linear equations software in a FORTRAN environment", Computer Architecture News, pp3-11, march 1985

[5] D.Gajski, D.Kuck, D.Lawrie, A.Sameh, "Cedar : a large scale multiprocessor", International Conference on Parallel Processing 1983, pp524-529

[6] A.Gottlieb & al., "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer" IEEE Transactions on Computers, Vol. C-32, pp175-189, feb.1983

[7] R.W.Hockney, "MIMD computing in the USA - 1984", Parallel Computing, 1985, pp119-136

[8] K.Hwang, F.A.Briggs, *Computer architecture and parallel processing,* Mac Graw Hill 1984

[9] P.M.Kogge, *The architecture of pipelined processors,* Mac Graw Hill 1981

[10] D.J.Kuck, R.A.Stokes, "The Burroughs Scientific Processor (BSP)", IEEE Transactions on Computers, vol C-31, pp. 363-376, May 1982.

[11] D.H.Lawrie,"Access and alignment of data in an array computer", IEEE Transactions on Computers, vol C-24, pp.1145-1155, dec.1975.

[12] D.H.Lawrie, C.R.Vora, "The prime memory system for array access", IEEE transactions on Computers, vol C-31, pp. 435-442, May 1982.

[13] C.V.Ramamoorthy, H.F.Li, "Pipeline Architecture", Computing Surveys, Mars 1977

[14] H.J.Siegel & al., "PASM : a partitionable SIMD/MIMD system for image processing and pattern recognition", IEEE Transactions on Computers, Vol C-30, pp934-947, 1981

[15] H.Tamura, Y.Shinkai, F.Isobe, "The supercomputer FACOM VP system", Fujitsu Sc. Tech. J. March 1985

[16] R.M.Tomasulo, "An efficient algorithm for exploiting multiple arihmetic units", IBM J., Vol. 11, Jan. 1967

[17] S.Weiss, J.E.Smith, "Instructions issue logic in pipelined supercomputers". Transactions on Computers, pp1013-1022, Nov. 1984

# Multipipeline Networking for Fast Evaluation of Vector Compound Functions*

Kai Hwang and Zhiwei Xu

Computer Research Institute
University of Southern California
Los Angeles, CA 90089-0781

## Abstract

Multipipeline networking is a generalized technique to expolit maximum parallelism in vector computations. A *pipeline net* can be viewed as a two-level pipelined systolic array, which is dynamically reconfigurable for evaluating various vector compound functions. In other words, the pipeline net can best match the various data dependency relationships in program graphs which can be vectorized. Multiple vector streams flow through the net to achieve significantly higher throughput than using conventional pipeline chaining. The reconfigurability provides the flexibility in various vector processing applications. This paper discusses design issues of the pipeline net and presents techniques for converting program graphs into pipeline nets.

## 1. Introduction

This paper addresses the design of scientific supercomputers which involve heavy vector computations. In such an environment, a computational job can often be decomposed into a number of communicating tasks. Each task may comprise several vector compound function evaluations [8, 13, 21]. Henceforth, we define a *vector compound function* (VCF) as a collection of linked scalar operations, which will be repeatedly processed many times in a looping structure. So far, these VCFs have been realized in array processors (Illiac-IV, MPP), pipelined uniprocessors (Cray-1, Cyber-205, and FPS-164/max) and multiprocessors (Cray X-MP, Cyberplus, FPS Tesseract, Remps, and Cedar) [4, 6, 9, 10, 12, 14, 20]. Most commercially available supercomputers are equipped with multiple pipelines in each central processor. Pipelining has been proven effective in implementing linked vector computations. However, only linear chaining has been implemented in commercial machines. This paper generalizes the conventional linear pipelining to a multipipeline networking approach. The goal is to further speed up the evaluation of VCFs for future pipelined multiprocessors.

As illustrated in Fig.1, the pipeline networking concept originates from the internal forwarding technique used in IBM 360/91 and the dynamic linking of functional units in CDC 7600 [12]. The concept of two-level pipelining has been practised in Cray-1 and Cray X-MP in the form of pipeline chaining [6]. However, only linearly connected patterns appear in these linked vector operations. In the Cyberplus [4], fifteen FPs in each processor can be linked by a crossbar network. In FPS-164/max or the recent FPS Tesseract, dot-product operations are executed by a multiplier pipeline cascaded with an adder pipeline [10, 20]. The networking concept generalizes the chaining practice to a two-level, reconfigurable systolic approach.

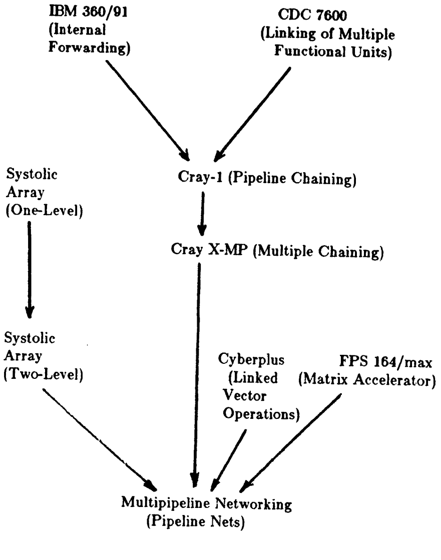


**Figure 1**. Historical evolution of the concept of pipeline networking

*This research was supported in part by an NSF grant DMC-84-21022 and in part by an AFOSR grant 86-0008

A *pipeline net* consists of multiple functional pipelines (FPs) interconnected by a buffered switching network, which is itself pipelined. Whenever a new VCF is to be evaluated, the pipeline net is reconfigured into a topology that best matches with the dataflow pattern in the program graph of the VCF. The evaluation is then performed with multiple operand strams flowing through the pipeline net in a synchronous fashion. Thus we can view a pipeline net as a two-level pipelined, dynamically reconfigurable systolic array. Using a single physical pipeline net, we can provide many *virtual* systolic arrays to support a large collection of application algorithms.

In this paper, we first characterize the functional structure of pipeline nets and describe their operational principles. We define *program graphs* and discusses their basic properties. We model VCFs with program graphs and provide a unified theory for converting program graphs into pipeline nets. Two *pipeline networking* techniques are presented, one using the concept of cut sets and the other using a retiming technique modified from Leiserson, Rose, and Saxe [18, 19].

## 2. The Concept of Pipeline Net

A *pipeline net* is made of three types of hardware resources, namely multiple *functional pipelines* (FPs), a *buffered crossbar network*, and a set of *data registers*, as illustrated in Fig.2. All FPs used in a net are identical and multifunctional. Different operations can be performed by the same FP at different times. They may, however, use different pipeline stages thus require different amount of pipeline delays. The registers are used as interface latches for holding operand and result data. The buffered crossbar network is used to provide dynamic interconnection paths among the FPs and the registers.

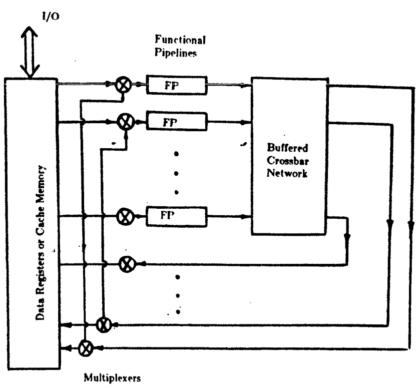


**Figure 2**. The schematic of a pipeline net

A crucial component of the pipeline net is the buffered crossbar network. We choose the crossbar over multistage packet switching networks [5] due to the demand of full connections in pipeline nets. In general purpose computations, pipeline nets with many different topologies may be needed. Thus the crossbar network should support arbitrary 1-to-1 and 1-to-many mappings. (Many-to-1 mappings are not allowed in pipeline networking operations). Apart from the full connectivity, crossbar switching has the advantages of ease to set up, which is very important since the network setup time is a major source of the startup overhead of the pipeline net. The main critique for crossbar network is the hardware complexity, which is $O(m^2)$ for an $m$ by $m$ network. However, if the number of FPs is relatively small (say 64), a crossbar switch is feasible with today's VLSI and packaging technology. In fact, several crossbar designs have been reported to have up to hundreds of inputs and outputs [2, 3, 7]. The recent progress in optical interconnection [22] promises the hope for even larger optical crossbar networks.

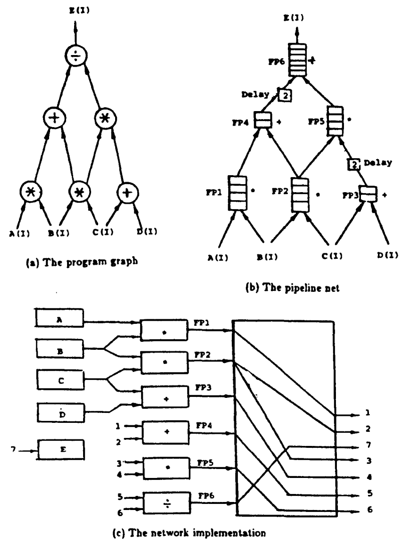
The pipeline net supports arbitrary connections among multiple pipelines. Thus local connections as necessary in a systolic array, are no longer a structural constraint in a pipeline net. However, the systolic flow of data through pipelines is still preserved. For example, when two operand streams arrive at a certain pipeline, they may have traversed through some data paths with different delays. These path delays must be equalized in order to have the correct operand pairs arriving at the right place at the right time. In the pipeline net, delay matching is handled by the crossbar network. Some programmable bufferes (latches) are provided on each output port of the crossbar network, so that proper noncompute delays can be inserted on each data path. An example design is the LINC chip [11], which is an 8-by-8 crossbar with up to 32 units of programmable delays on each data path.

The pipeline net performs computations based on the *pipeline networking* concept, which is a natural generalization of pipeline chaining in Cray-1 and Cray X-MP. The idea is that after the operand data are loaded into the register file, a pipeline net consisting of vector registers, functional pipelines, and a crossbar network is dynamically set up. A block of operands (which is called as a *wavefront* in Kung [17]) may traverse multiple pipelines via the network, before it finally returns to the register file. Intermediate results flow directly from a pipeline to another without memory accesses. The pipeline net best matches the dataflow pattern of the VCF to be evaluated, thus operand fetchings, arithmetic/logic operations, intermediate result routing, and final result storing are executed concurrently in a pipelined fashion as illustrated by the following example.

**Example 1.** Consider the following Fortran loop representing a VCF. Each iteration of the loop is characterized by the program graph shown in Fig.3a.

```
       DO 10 I=1, 400
       T(I) = B(I)*C(I)
    10 E(I) = (A(I)*B(I)+T(I))/(T(I)*(C(I)+D(I)))
```

Suppose that the add, multiply, and divide pipelines require 2, 4, and 6 pipeline stages respectively. This graph can be systematically mapped into an equivalent pipeline net shown in Fig.3b. Noncompute delays are inserted into data paths connecting FP3 and FP4 to FP5 and FP6. All inter-pipeline data paths are provided by the crossbar network as shown in Fig.3c. After the pipeline net is configured, the VCF is evaluated by passing 400 operand blocks (wavefronts) from the vector registers A, B, C, D through the net. And the final result data will be stored in register E. Note that no temporary storage is needed for the intermediate result T.



(a) The program graph

(b) The pipeline net

(c) The network implementation

Figure 3. An example of multipipeline networking

## 3. Properties of Program Graphs

A *program graph* $G=(V,E,f_1,f_2)$ is a weighted directed graph, where

1. $V$ is a set of nodes representing arithmetic/logic operations;
2. $E$ is a set of edges representing data dependency;
3. $f_1$ is a function from $E$ to $\{0,1,2,...\}$ denoting the edge delays;
4. $f_2$ is a function from $V$ to $\{0,1,2,...\}$ denoting the nodal delays; and
5. There are two specific nodes $v_{in}, v_{out} \in V$ for handling I/O operations. Note that $v_{in}$ has no inbound edges and $v_{out}$ has no outbound edges.

A *synchronous* program graph is one in which every cycle has at least one nonzero node or edge delay. Since asynchronous graph involves nondeterministic behavior, we only consider synchronous graphs in this paper. In the following, the term graph refers to synchronous graphs only. A $(k_1,k_2)$-graph, is a program graph in which $k_1 \leq f_2(v) \leq k_2$ for all $v \in V$. A $(k_1,k_2)$-graph is abbreviated as a $k$-graph when $k_1=k_2=k$. Note that 0-graphs and 1-graphs are special cases of $k$-graphs. A systolic graph is a 0-graph with positive edge delays (i.e., $f_1(e)>0$ for all $e \in E$). We shall denote such a systolic graph as $G_0^+$.

When a graph is implemented by a hardware circuit, the delay of an edge represents the number of delay registers on that edge. A computing node with delay $k$ corresponds to a $k$-stage linear pipeline which has $k$ latches. At any time, the graph has a *state* which is defined to be the contents of all edge registers and pipeline latches. All input data are issued from the input node $v_{in}$ and all results are sent to the output node $v_{out}$. A set of input data which is issued at the same time is called a wavefront (operand block). A computational task is performed by a program graph $G$ synchronously: Suppose at time $t_0$ the *initial state* of $G$ is $s_0$. A sequence of wavefronts, $I$, is input from the the input node at time $t_0+i\alpha$ for $i=1,2,\cdots,n$, which produces an ouput sequence $O$ at the output node at time $t_0+d+i\alpha$ for $i=1,2,\cdots,n$. Note that $n$ is the number of wavefronts feeding into $G$ from the input node, and $\alpha$, called the *spacing*, is defined as the number of clock periods elapsed between two successive wavefronts.

Two program graphs $G$ and $G'$ are *equivalent*, denoted as $G \equiv G'$, if the same input sequence produces the same output sequence on both graphs. In other words, equivalent graphs have the same input/output behavior. However, the nodal and edge delays and the spacings between wavefronts on each graph may be different.

The following lemmas provide basic tools for transforming a program graph into an equivalent pipeline net, as illustrated in Fig.4. Detailed proofs of these lemmas can be found in [15]. A *cut-set* in a graph $G$ is a minimal set of edges the removal of which partitions the graph into two isolated subgraphs, called the *left subgraph* (which contains $v_{in}$) and the *right subgraph* (which contains $v_{out}$). All edges from the left subgraph to the right subgraph are called ***rightbound*** edges, and those in the reverse direction are called *leftbound* edges.

**Lemma 1**. Adding $k$ delays to any node in a program garph and then subtracting $k$ delays from all inbound-edges (or all outbound-edges) of that node will produce an equivalent graph. An equivalent graph can also be obtained, if "adding" and "subtracting" are exchanged in the above operation. (Fig.4b)

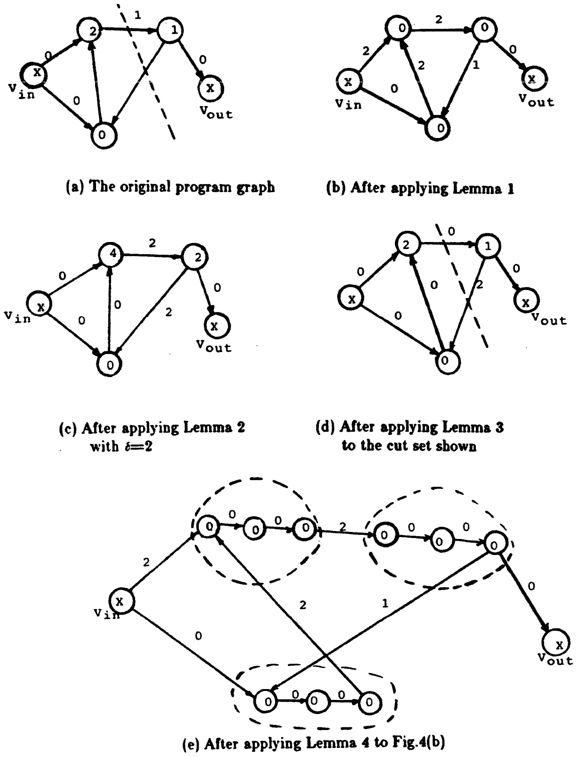
**Lemma 2**. An equivalent graph is generated, if all nodal and edge delays and the spacing are multiplied by a positive integer. (Fig.4c)

**Lemma 3**. For any cut-set of a program graph, adding $k$ delays to all leftbound (or rightbound) edges and at the same time subtracting $k$ delays from all rightbound (or leftbound) edges will result in an equivalent graph. (Fig.4d)

**Lemma 4**. For any 0-graph, an equivalent graph is obtained by splitting any node into a linear cascade of $k$ nodes with zero nodal and edge delays in the cascade. (Fig.4e)

## 4. Mapping Program Graphs to Pipeline Nets

A user task is specified with a program graph $G_0$. Usually $G_0=(V,E,f_1,f_2)$ is given as a 1-graph if we assume each arithmetic/logic operation takes one time unit to complete; or it is specified as a 0-graph if it is given as a *signal flow graph* [17]. we hope to transform it into an equivalent program graph $G_k=(V,E,f_1',f_2')$, which corresponds to the required pipeline net. Note that the transformed graph $G_k$ has the same topology as $G_0$, but the delays and the spacing may be modified.



(a) The original program graph

(b) After applying Lemma 1

(c) After applying Lemma 2 with $\delta$=2

(d) After applying Lemma 3 to the cut set shown

(e) After applying Lemma 4 to Fig.4(b)

**Figure 4. Illustrating Lemma 1-4 with an example graph**

The new nodal delay function $f_2'$ is determined as follows: for any node $v_i$ other than $v_{in}$ and $v_{out}$, if $v_i$ is implemented by a pipeline of $k_i$-stages in the required pipeline net, then $f_2'(v_i)=k_i$.

Leiserson et al [18, 19] studied the problem of minimizing the clock period of a synchronous system by a technique called "retiming". They provided algorithms to tramsform a synchronous system into a systolic array. Similar work is also reported in [17], where a systolization procedure is used to transform a signal-flow graph into a systolic array. In this paper we choose a different approach for a different purpose. All the previous researchers tramsform a program graph $G$ into a single-level systolic array, $G_0{}^+$; while our purpose is to convert the graph $G$ into a $k$-graph $G_k$, which can be implemented by a two-level pipeline net.

Two systematic methods are presented below to perform the network conversion. We first present cut-set method based on Lemmas 1, 2, and 3. This method is closer to what has been reported in [16], where a cut-set rule is used to transform feedback-free program graphs into two-level pipelined systolic arrays. We investigate more general cases which allow feedbacks in the program graphs. We then extend Leiserson's retiming method to generate two-level pipelined systolic arrays. These network conversion methods are compared with previous approaches in Fig.5. The following theorem is a direct concequence of the two network conversion algorithms to be presented shortly.

**Theorem.** Any synchronous graph $G_0$ can be transformed into a pipeline net characterized by a $k$-graph $G_k$ such that $G_0 \equiv G_k$.

In what follows, the delay of a cycle $c_j$ in a program graph $G$ is denoted as $d(c_j)$. The sum of all the nodal delays of the same cycle $(\bar{c}_j)$ in $G_k$ is denoted as $\bar{d}(\bar{c}_j)$. A *chain* is a garph whose nodes can be arranged in a linear cascade as examplified in Fig.6a. The rightbound (leftbound) edges connecting distant nodes are called *forward* (*backward*) edges as shown in Fig.6a. A *multichain* consists of several chains linked by forward or backward edges as shown in Fig.6c.

Given a graph $G$, we define a *maximal acyclic subgraph* (MAS) of $G$, as a cycle-free subgraph of $G$ which contains all the nodes of $G$ and by reattaching one more edge, we will have a cyclic graph. For example, an MAS of the cyclic graph shown in Fig.7a is depicted in Fig.7b. Before describe the main algorithm, we recall that any acyclic graph can be converted into a chain by *topologic sorting* [1]. In the following procedure, we first transform a given graph $G$ into a multichain. Then we use cut sets, which seperate successive adjacent nodes in the multichain, to convert $G$ into $G_k$.

**Cutset Conversion Algorithm:** This algorithm transforms a program graph $G$ into a $k$-graph $G_k$ such that $G \equiv G_k$.

1. Find an MAS of $G$. Note that if $G$ is a cyclic graph, then some edges are removed.

2. Perform topologic sorting to obtain a multichain $v_{in}, v_1, v_2, \cdots, v_m, v_{out}$.

3. Reattach all edges removed in Step 1.

4. Apply Lemma 1 to obtain a 0-graph $G_0$ by removing all nodal delays to the corresponding inbound edges.

5. For $i$=1,2, $\cdots$ ,$m$+1, apply Lemma 3 to the cut set $S_i$ consisting of all edges between $v_{i-1}$ and $v_i$. Note that $v_0$ and $v_{m+1}$ represent $v_{in}$ and $v_{out}$ respectively.

   a. Let $e_1, e_2, \cdots, e_r$ be all the rightbound edges in cutset $S_i$, $w$=min{delay of $e_j$ for $1 \leq j \leq r$}. If $w=k_i$, then do nothing.

   b. If $w>k_i$, then add $w-k_i$ to the delay of all leftbound edges in $S_i$ and subtract $w-k_i$ from the delay of all rightbound edges in $S_i$.

   c. If $w<k_i$, then we must consider two cases. If the cut set contains leftbound edges and there is a forward edge $e$ in the cut set with delay $w$, examine whether $e$ belongs to any previous used cut set without leftbound edges. If so, apply Lemma 3 to raise the delay of $e$ to $k_i$ and goto Step 5c. Otherwise add $k_i-w$ to the delay of all rightbound edges and subtract $k_i-w$ from the delay of all leftbound edges.

   At the end of Step 5, a new edge delay function is obtained. Denote it as $f_1'$.

6. If all inbound edges of $v_i$ have delay $\geq k_i$ for $i$=1,2, $\cdots$ ,$m$, go to step 7. Otherwise

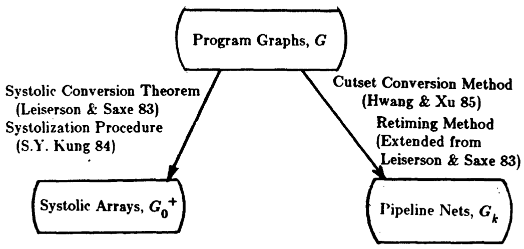
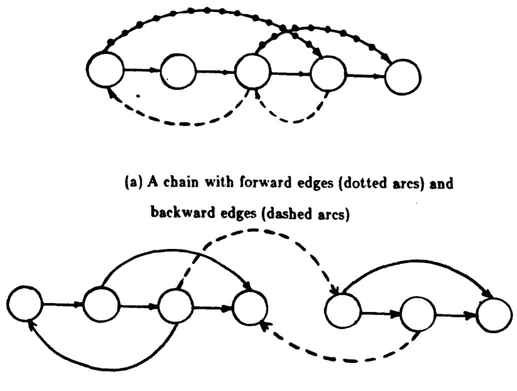


**Figure 5.** Various methods for converting program graphs to systolic arrays or to pipeline nets


(a) A chain with forward edges (dotted arcs) and backward edges (dashed arcs)

(b) A multichain consisting of two chains linked by distant edges (dashed arcs)

**Figure 6.** Chain and multichain of functional pipelines

a. Compute the scale-up factor

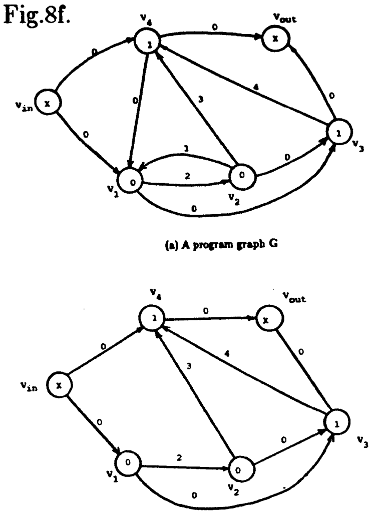
$$\delta = \operatorname*{Max}_{\substack{e_i \in c_j \\ c_j \in C}} \left\{ \frac{\bar{\bar{d}}(c_j) - f_1'(e_i)}{d(c_j)} \right\}$$

where the $e_i$ is a leftbound edge in cycle $c_j$ and $C$ is the set of all cycles.

b. Apply Lemma 2 to scale up the 0-graph $G_0$ obtained in Step 4 $\delta$ times. Goto Step 5.

7. Transfer $k_i$ delays to $v_i$ from its inbound edges for all $v_i \in V$, we obtain the required pipeline net corresponding to $G_k$.

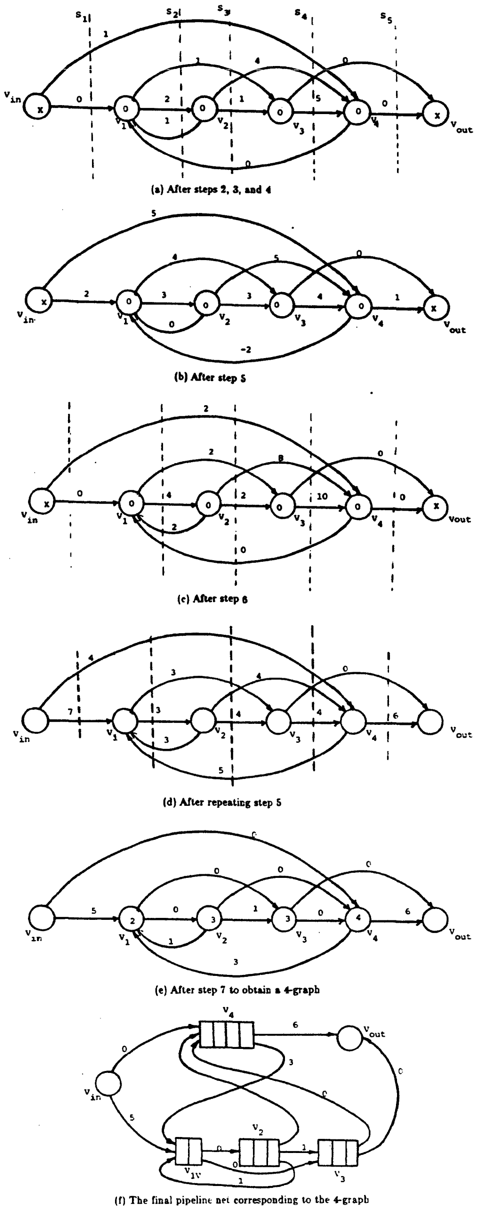
**Example 2**. Consider the program graph in Fig7a. We want to convert it into a 4-graph with $k_1$=2,$k_2$=$k_3$=3, and $k_4$=4. By removing edges ($v_4$,$v_1$), and ($v_2$,$v_1$), we obtain an MAS in Fig.7b. Applying topological sorting, reattaching the removed edges, and transfering nodal delays to corresponding inbound edges, we obtain the graph in Fig.8a. applying Lemma 3 to cut sets $S_1$, $S_2$,$S_3$, $S_4$, and $S_5$ in Fig.8a, the edge delays are modified as shown in Fig.8b. Now the two leftbound edges in Fig.8b, having delay less than $k_1$=2, creat 4 cycles in the multichain. The scale-up factor $\delta$={ 5/3, 14/8, 11/6, 11/6}=2. Scaling up the graph in Fig.8a two times, we obtain the graph in Fig.8c. We then repeat Step 5 once more to obtain the graph in Fig.8d. Now all edges have delay greater than or equal to the required value. By moving $k_i$ delays from the inbound edges of $v_i$ into $v_i$, we obtain the required 4-graph in Fig.8e, which is explicitly redrawn as a pipeline net in Fig.8f.



(a) A program graph G

(b) A maximally acyclic subgraph of G

**Figure 7.** A sample program graph and the maximally acyclic subgraph obtained after step 1 of the network conversion algorithm



(a) After steps 2, 3, and 4

(b) After step 5

(c) After step 6

(d) After repeating step 5

(e) After step 7 to obtain a 4-graph

(f) The final pipeline net corresponding to the 4-graph

**Figure 8.** Successive graphs obtained in applying the network conversion algorithm

## 5. Designing Pipeline Nets by Retiming

Leiserson and Saxe [19] studied the problem of mapping synchronous graphs into systolic graphs. Their results are modified below for generating pipeline nets. Before describing our algorithm, we restate their retiming technique using our terminology.

**Retiming Lemma.** Let $G$ be a synchronous 0-graph and *lag* be a function that maps the I/O nodes to zero and any other node to an integer. Suppose that for every edge $e=(u,v)$, the value $f_2(e)+lag(v)-lag(u)$ is nonnegative. Let $G'$ be the 0-graph obtained by replacing the delay $f_2(e)$ of every edge $e=(u,v)$ by $f_2(e)+lag(v)-lag(u)$. Then $G \equiv G'$.

**Systolic Conversion Theorem** (Leiserson and Saxe). Let $G$ be a synchronous 0-graph, and $\overline{G}$ is the graph obtained by subtracting all edge delays $G$ by 1. Suppose that $\overline{G}$ has no cycles of negative delay, then there exists a systolic graph $G_0{}^+$ which is equivalent to $G$.

A constructive proof of the above theorem can be found in [19]. Given any 0-graph $G$, they first derive $\overline{G}$. Then they specify the *lag* function as follows: for any node $v$, there exists in $\overline{G}$ a path of minimal delay from $v$ to the output node. $lag(v)$ is then determined as the delay of any such ***shortest path***. They proved that by applying the Retiming Lemma to $G$ with the defined *lag* function, $G$ can be mapped into a equivalent systolic graph if $\overline{G}$ has no cycle of negative delay. The following algorithm applies the above results to map a program graph $G_0$ into a pipeline net.

**Retiming Conversion Algorithm:** This algorithm convert a program graph $G$ into a pipeline net $G_k$.

1. Transfer all nodal delays to inbound edges to obtain a 0-graph $G_0$.

2. Compute $\delta = \max_j \{\overline{d}(c_j)/d(c_j)\}$ for all cycles $\{c_j\}$ in $G_0$. Scale up the graph $G_0$ $\delta$ times using Lemma 2.

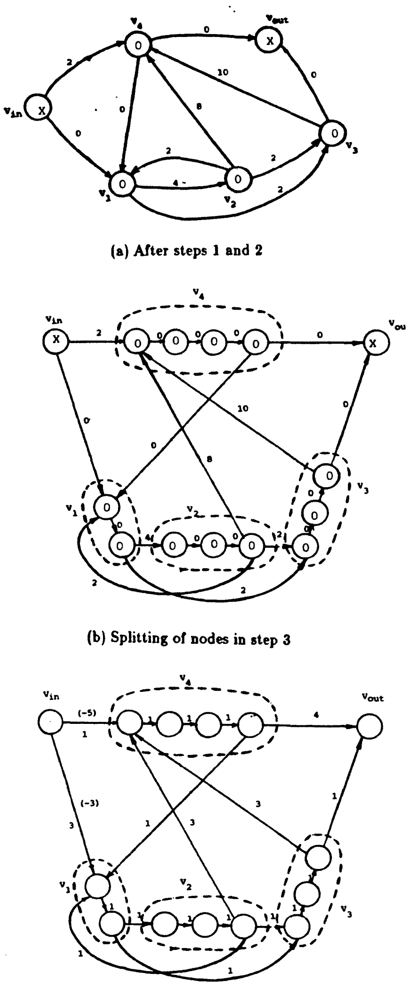
3. For every node $v_i$, if it corresponds to a $k_i$-stage pipeline in the required pipeline net $G_k$, then apply Lemma 4 to split it into a linear cascade of $k_i$ nodes with zero nodal and edge delays. Denote the obtained graph as $G'$.

4. Apply the Systolic Conversion Theorem to $G'$ to obtain a systolic graph $G_0{}^+$. Apply Lemma 3 to eliminate negative or zero delays on all outbound edges of $v_{in}$, if there is any.

5. Merge the splitted nodes in each cascade into a single node by assigning $k_i-1$ delays to $v_i$ for all $v_i \in V$.

6. Transfer one delay from all inbound edges $v_i$ for all $v_i \in V$, we obtain the required pipeline net corresponding to $G_k$.

**Example 3.** Consider the conversion of the program graph Fig.7a to an equivalent 4-graph. After Steps 1 and 2, we obtain the 0-graph shown in Fig.9a. Note that the multiplying factor $\delta$ is found to be 2. The 0-graph is then expanded into a 14-node graph $G'$ as shown in Fig.9b. The Systolic Conversion Theorem is then applied to obtain an equivalent systolic graph $G_0{}^+$ as shown in Fig.9c. Note that Lemma 3 has been applied to the outbound edges of the input node to eliminate negative delays. Grouping every cascade of nodes into a single node, we obtain the graph $G_3$ shown in Fig.9d. The final pipeline net is shown in Fig.9e corresponding to the graph $G_4$.

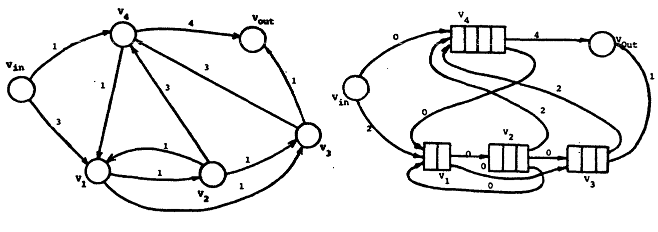


(a) After steps 1 and 2

(b) Splitting of nodes in step 3

(c) Applying the retiming lemma in step 4

**Figure 9.** Successive graphs obtained in applying the retiming conversion algorithm



(d) Grouping of nodes in step 5 (e) The final pipeline net after step 6

**Figure 9**(Continued). Successive graphs obtained in applying the retiming conversion algorithm

## 6. Conclusions

This paper generalizes the conventional linear pipelining principle. Pipeline networking offers a new design methodology for vector multiprocessor supercomputers. Basic techniques are provided for mapping program graphs into pipeline nets. These techniques are useful for designing two-level pipelined systolic arrays. By matching the multiprocessor configuration with the dataflow patterns of user programs, and by combining pipelining with parallelization, the pipeline net architecture provides higher flexibility and higher throughput, especially for compound vector/matrix operations.

## References

**1.** Aho, A., Hopcroft, J., and Ullman, J.. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1976.

**2.** Burggraff, T., Love, A., Malm, R., and Rudy, A. •The IBM Los Gatos Logic Simulation Machine Hardware•. *Proc. ICCD* (1983), 584-587.

**3.** Broomell, G. and Heath, J.R. •An Integrated-Circuit Crossbar Switching System•. *Proc. 4th Int'l. Conf. on Distributed Computing Systems* (May 1984), 278-287.

**4.** Control Data Corporation. *An Introduction to the Cyberplus Parallel Processing System.* 1985.

**5.** Chin, C.Y. and Hwang, K. •Packet Switching Networks for Multiprocessors and Dataflow Computers•. *IEEE Trans. on Computers C-33,* 11 (Nov. 1984), 991-1003.

**6.** Cray Research, Inc. *The Cray X-MP Series of Computer Systems.* Minneapolis, Minnesota, 1984.

**7.** Denneau, M.M. •The Yorktown Simulation Engine•. *Proc. 19th Design Automation Conference* (1982), 55-59.

**8.** Gajski, D.D., Kuck, D.J., and Padua, D.A. •Dependence-Driven Computation•. *Proc. Compcon Spring* (Feb. 1981), 168-172.

**9.** Gajski, D., Lawrie, D., Kuck, D., and Sameh, A. •Cedar•. *COMPCON* (Spring 1984), 36-309.

**10.** Gustafson, J.L., Hawkinson, S., and Scott, K. •The Architecture of a Homogeneous Vector Supercomputer•. *Proc. 1986 Int'l. Conf. on Parallel Processing* (Aug. 1986)

**11.** Hsu, F.H., Kung, H.T., Nishizawa, T., and Sussman, A. *LINC: The Link and Interconnection Chip.* Dept. of Computer Science, Carnegie-mellon Univ., 1984.

**12.** Hwang, K. and Briggs, F.A.. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

**13.** Hwang, K. and Cheng, Y.H. •Partitioned Matrix Algorithms for VLSI Arithmetic Systems•. *IEEE Trans. on Computers C-31,* 12 (Dec. 1982), 1215-1224.

**14.** Hwang, K. and Xu, Z. •Remps: A Reconfigurable Multiprocessor for Scientific Supercomputing•. *Proc. 1985 Int'l. Conf. on Parallel Processing* (Aug. 1985), 102-111.

**15.** Hwang, K. and Xu, Z. “Multipipeline Networking for Fast Evaluation of Vector Compound Functions.” CRI-85-008, Computer Research Institute, University of Southern California, Los Angenes, CA 90089, July, 1985.

**16.** Kung, H.T. and Lam, M. •Wafer-Scale Integration and Two-Level Pipelined Implementation of Systolic Arrays•. *J. of Parallel and Distributed Computing 1,* 1 (Sept. 1984), 32-63.

**17.** Kung, S.Y. •On Supercomputing with Systolic/Wavefront Array Processors•. *Proceeding of IEEE 72,* 7 (July 1984).

**18.** Leiserson, C.E., Rose, F.M, and Saxe, J.B. •Optimizing Synchronous Circuitry by Retiming•. *Proc. of Third CalTech Conf. on VLSI* (1983), 87-116.

**19.** Leiserson, C.E. and Saxe, J.B. •Optimizing Synchronous Systems•. *J. VLSI and Comput. Syst. 1,* 1 (1983), 41-68.

**20.** Madura, D., Broussard, R., and Strickland, D. *FPS-164/MAX: Parallel Multiprocessing for Linear Algebra Operations.* Floating Point Systems, inc., 1984.

**21.** Ni, L.M. and Hwang, K. •Vector Reduction Techniques for Arithmetic Pipelines•. *IEEE Trans. on Computers C-34,* 5 (May 1985), 404-411.

**22.** Sawchuk, A.A., Jenkins, B.K., Raghavendra, C.S., and Varma, A. •Optical Interconnection Networks•. *Proc. 1985 Int'l. Conf. on Parallel Processing* (Aug. 1985), 388-392.

# A Parallel Vector Reduction Architecture

H.X. Lin and H.J. Sips
Delft University of Technology
The Netherlands

## Abstract

The ability to perform fast vector reduction evaluations, such as inner product and vector summation, is very important in many scientific and engineering computer applications. Most vector reductions are performed by using a single arithmetic unit in a repetitive way. In this paper the architecture and performance of a pipelined, parallel vector reduction processor is discussed.

The performance of the parallel reduction processor is given as a function of the size of the vector to be reduced and the number of pipeline segments of the reduction processors involved. In addition to that, the number of required reduction processors can be determined as a function of the length of the vector to be reduced and a prespecified space-time criterion based on a doubling improvement ratio.

## I. Introduction

Vector reduction techniques for arithmetic pipelines have been proposed by Kuck [1], Kogge [2], and Ni and Hwang [3]. In essence, the vector reduction operation is the application of the dyadic operator on the elements of a vector, such that the order of evaluation is immaterial, i.e. the operations are associative and commutative. Let "$o$" be the dyadic operator, $X[i]$, $i=1,\dots,n$ the vector, and $Z$ the scalar output, the vector reduction operation can then be formulated as

$$Z = X[1]oX[2]o\dots oX[n] \qquad (1)$$

The vector reduction operation can be performed by a pipelined reduction processor (Fig. 1), where $q$ is the number of sections in the pipeline. To establish the performance of a vector reduction method M, it is convenient to partition the process into three phases:

$$T_M^{re} = T_M^f + T_M^m + T_M^d \qquad (2)$$

where $T_M^f$ is the number of cycles needed to enter all elements of the vector $X$ into the reduction processor (feed phase), $T_M^m$ is the number of cycles needed to merge the partial results in the pipeline to a single result (merge phase), and $T_M^d$ is the number of cycles to drain the pipeline (drain phase).

It can easily be observed that $T_M^f + T_M^d = n + q - 1$ for all methods $M$, leading to

$$T_M^{re} = n + q - 1 + T_M^m \qquad (3)$$

Therefore, the main difference between the methods in [1-3] is the merging time $T_M^m$. The smallest values of $T_M^m$ are obtained by applying the methods of Ni and Hwang [3], which are further improvements of Kuck's and Kogge's method [1,2]. They developed two different methods: a *symmetric* and an *asymmetric* method.

In Fig. 1 the hardware configuration for both methods is shown. The first $min\{n,q\}$ cycles the elements of $X$ are merged with a constant $C$ into the reduction processor ($C=0$ for vector summation and inner product). If $n \leq q$, then the merging phase is started. If $n>q$ the next elements of $X$ are paired with the results leaving the pipeline. This procedure continues until no element of $X$ is left.

In the symmetric method (SM-method) two consecutive productive segments are merged, when leaving the pipeline. A segment is said to be productive, if it contains data which is part of the overall reduction operation. The register is used to hold a partial result until the next partial result leaves the pipeline. These two partial results are then entered again into the reduction processor. If the number of productive segments is odd, the last partial result is merged with a dummy result, keeping the distance between the productive segments always $2^i$ after the $i$-th iteration. The merging time $T_M^m$ then equals [3]

$$T_{SM}^m = \begin{cases} q\lceil log q\rceil + 2^{\lceil log q\rceil} - q & if \quad n>q \\ q\lceil log n\rceil + 2^{\lceil log n\rceil} - n & if \quad n\leq q \end{cases} \qquad (4)$$

The asymmetric method (AM-method) differs from the symmetric method in the treatment of the last partial result in the case of an odd number of productive segments. In this situation the last partial result is not merged with a dummy partial result, but is left in the register waiting for the next partial result leaving the pipeline. The merging time of the asymmetric method equals [3]

$$T_{AM}^m = \begin{cases} q\lceil log q\rceil - 2^{\lceil log q\rceil} + q & if \quad n>q \\ q\lceil log n\rceil - 2^{\lceil log n\rceil} + n & if \quad n\leq q \end{cases} \qquad (5)$$

Clearly, the asymmetric method is faster than the symmetric method, with the exception of $q=2^r$, $n \geq q$, when both methods have equal performance. However, the control sequence of the asymmetric method is more difficult to implement, because it cannot be expressed in a simple formula [3].

In this paper a parallel processing architecture for performing the vector reduction operation, consisting of a linear array of pipelined

0190-3918/86/0000/0503 $01.00 © 1986 IEEE

processors, is presented. The performance of the proposed architecture is determined. A formula is derived, by which the number of required reduction processors can be determined as a function of the size of the vector to be reduced and a prespecified space-time criterion.

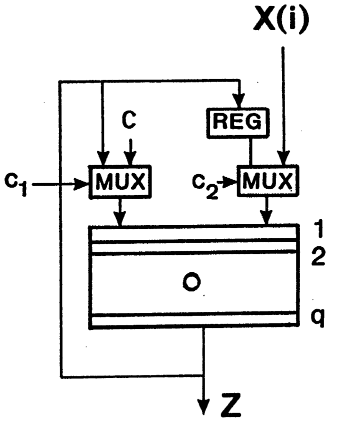


*Fig. 1. Pipelined reduction processor*

## II. Architecture of the parallel reduction processor

The parallel reduction processing structure considered in this paper is the linear array. The linear array consists of $N$ reduction pipelines (Fig. 2); every reduction pipeline having its own local memory. In order to be able to produce a scalar output value, the $N$ pipelines are connected by means of an interconnection network. A single reduction pipeline consists of $q$ segments. Some values of $q$ in actual systems are $q=7$ (o=*) and $q=6$ (o=+) for the CRAY-1 [5], and $p=7$ (o=*), $q=8$ (o=+) for the Cyber 205 [6].

In the following, we will consider the reduction of a vector of dimension $n$, by using $r$ of the $N$ reduction pipelines.

The $n$ vector is divided into $r$ subvectors, each with $\lceil n/r \rceil$ elements. Each of the $r$ subvectors is fed to one of the processors $P_1, P_2, \dots, P_r$. The merging phase can be done in two different ways:

method-1: All subvectors are merged into a single result in each of the $r$ processors. These partial results are then transferred, one by one, from $P_2$ until $P_r$ to $P_1$. In fact this can be viewed as the reduction of an $r$ element vector into a single result. The process is depicted in Fig. 3a. The total number of processing cycles is equal to

$$T_M^{re} = \left\lceil \frac{n}{r} \right\rceil + r + 2(q-1) + T_M^m\left(min\left(\left\lceil \frac{n}{r} \right\rceil, q\right)\right) + T_M^m(min(r,q)) \qquad (6)$$

method-2: After the individual feed-phase of the $i$-th subvector on the $P_i$-th processor, the $min\{\lceil n/r \rceil, q\}$ results in each pipeline are pairwise merged; $P_1$ and $P_2$ in $P_1$, $P_3$ and $P_4$ in $P_3$, etc. Then $P_1$ and $P_3$ are merged in $P_1$, $P_5$ and $P_7$ in $P_5$, etc. This process continues until all results are transferred to $P_1$, where they are finally merged to a single result (Fig. 3b). The transfer and merge process from $P_1, \dots, P_r$ to $P_1$ takes $q \lceil log r \rceil$ cycles. The total number of processing cycles equals

$$T_M^{re} = \left\lceil \frac{n}{r} \right\rceil + q.\lceil log r \rceil + q - 1 + T_M^m\left(min\left\{\left\lceil \frac{n}{r} \right\rceil, q\right\}\right) \qquad (7)$$

An alternative way of achieving the same result as in (7) is shown in Fig. 3c. The $min\{\lceil n/r \rceil, q\}$ results are first merged to a single result. Then the results are pairwise merged, taking $q.\lceil log r \rceil$ cycles. Therefore, with the inclusion of the feed phase and the final drain phase, the same number of processing cycles, as denoted in Eq. (7), is derived. However, this version of method-2 has the important advantage that only $\lceil log r \rceil$ instead of $(min\{\lceil n/r \rceil, q\})$. $\lceil log r \rceil$ results have to be transferred between the reduction processors.

The difference between method-1 and method-2 equals

$$\begin{aligned} DIFF(M) &= T_M^{re}(method1) - T_M^{re}(method2) \\ &= r + T_M^m(min(r,q)) + q - 1 - q.\lceil log r \rceil \qquad (8) \end{aligned}$$

For $r \leqslant q$, we substitute the AM-method, yielding $DIFF(AM) = 2r - 2^{\lceil log r \rceil} + q - 1$. The worst case is obtained for $r = 2^k + 1$, resulting in $DIFF(AM) = q + 1$. Therefore, if $r \leqslant q$, method-2 is always faster than method-1. For $r > q$, we get from eq. (8) $DIFF(AM) = r + q.\lceil log q \rceil - 2^{\lceil log q \rceil} + 2q - 1 - q.\lceil log r \rceil$. Substitution of $r = k.q$, $k > 1$ leads in the worst case to $k.q + q\lceil log q \rceil - 2^{\lceil log q \rceil} + 2q - 1 - q.\lceil log(k.q) \rceil \geqslant k.q - 2^{\lceil log q \rceil} + 2q - 1 - q.\lceil log k \rceil$. Since $k > \lceil log k \rceil$ and $-2^{\lceil log q \rceil} + q \geqslant -q + 2$, it follows that $k.q - 2^{\lceil log q \rceil} + 2q - 1 - q.\lceil log k \rceil > 0$. Therefore, it is concluded that method-2 is faster than method-1.

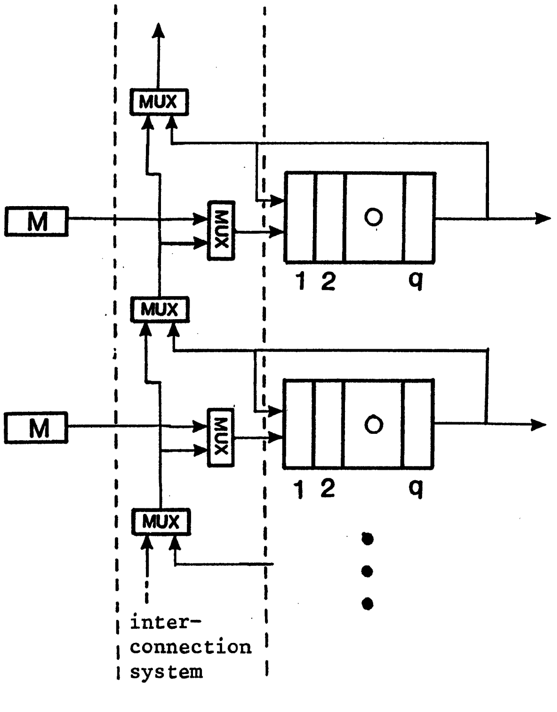


*Fig. 2. The partitioned linear array of reduction processors*

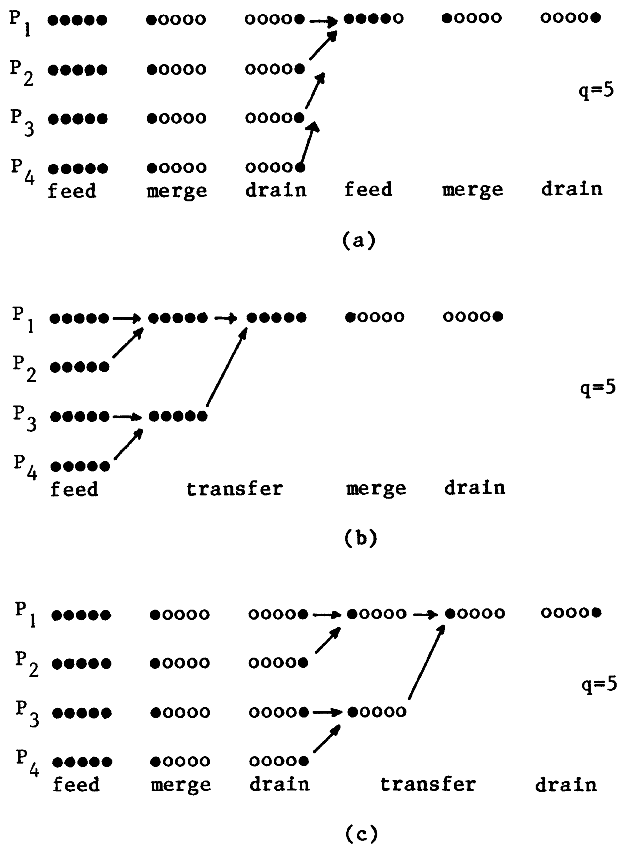


*Fig. 3. a) Processing according to method 1,*
*b) Processing according to method 2,*
*c) alternative for method 2.*

A possible realization of the interconnection system between the $r$ pipelines is shown in Fig. 2. By means of a linear chain of multiplexors the interconnection system can easily be partitioned in $1, 2, \ldots, \lceil r/2 \rceil$ separate buses. Since the inter-pipeline transfers do not overlap in time, this simple system is adequate. The interconnection network in Fig. 2 has an advantage over non-partitionable linear array approaches, since the latter needs $O(N)$ cycles to merge the $N$ partial results from the individual reduction pipelines, whereas by using the approach of Fig. 2, this requires only $O(logN)$ cycles.

### Performance of the binary tree

In the following, we consider the performance of a pipelined binary tree processor for vector reduction. The structure of such a processor is illustrated in Fig. 4.

Suppose a subtree with r pipelined units of the (maximal) $N$ pipelined units is used to reduce an $n$-element vector, $r=1,2,4, \ldots, N$. To evaluate Eq. (1), $n$ elements of the vector are divided into $\lceil n/r \rceil$ groups, $G(j)$, for $j=1,2, \ldots, \lceil n/r \rceil$ . When $n$ is not an integral multiple of $r$, $\lceil n/r \rceil . r-n$ zero-elements are added to the end of the vector, such that each group consists of $r$ elements. Group $G(j)$ is fed into the uppermost $r/2$ pipelined units at the $j$-th clock cycle, for $j=1,2, \ldots, \lceil n/r \rceil$ . After $\lceil n/r \rceil +q. \lceil logr \rceil$ cycles, $min\{q, \lceil n/r \rceil \}$ partial results are obtained in the segments of the cumulative pipeline unit. Next, these $min\{q, \lceil n/r \rceil \}$ partial results in the cumulative unit must be merged into the final result, the reduction time of this group-merging phase is given in Eq. (4) and Eq. (5). Finally, $q-1$ cycles are required to drain the last pipeline. Thus the total reduction time is

$$Trt^{re}_{M}(n,r)=\left\lceil\frac{n}{r}\right\rceil+q.\lceil logr\rceil+ T^{m}_{M}\left(\left\lceil\frac{n}{r}\right\rceil\right)+q-1 \qquad (9)$$

($rt$=reduction-tree)

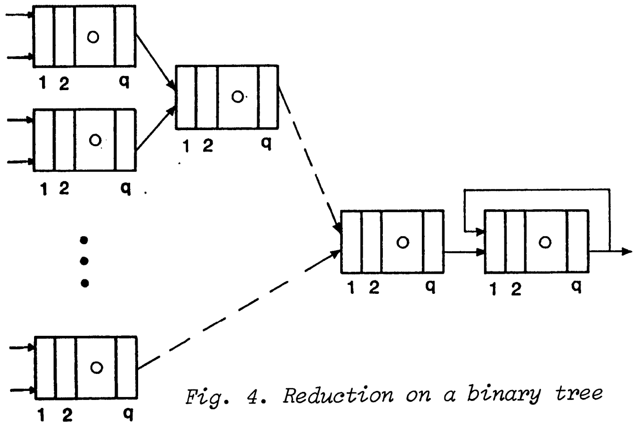


*Fig. 4. Reduction on a binary tree*

From Eq.(7) and Eq.(9), we can see that the reduction time is the same for the partitioned linear array and the binary reduction-tree. However, the proposed approach is much simpler to implement. It is also very regular, so this approach is suited to VLSI-implementation. For large chains, the multiplexors will give some extra delay. However, if the method of Fig. 3c (method-2) is followed, only one result per bus topology connection has to be transferred. Another solution is to construct a system with a large number of processors in a modular and hierarchical way. For example, every $p=2^{8}$ processors form a cluster. Within the cluster the interconnection is as described above; between every $p$ clusters a similar type of (multiplexor) interconnection network is established, etc. In vector reduction, the data only has to be transferred between the "bottom" pipelines of each cluster ($P_1$ in Fig. 3). Consequently, the the maximal interconnection distance is $2+log(p)$ multiplexors. The partitioned linear array is also fault tolerant. If one of the processor faults, it can easily be isolated. When the global controller of the system has detected a faulty processor, it just "shunts" the corresponding multiplexor to allow the incoming data to be passed on to the next processor. Of course, the performance will degrade, since the data for the faulty processor must be fed to the correctly operating processors within the cluster. However, the decrease in performance is gracefully. In this way, any malfunctioning processors can be localized. Compared with a fault-tolerant approach for a binary tree structure [9], the partitioned linear array is much simpler and more superior; there is no need of extra processors and connections.

## III. Minimal computation time

For the determination of the minimal computation time, we will only consider method-2. Suppose we have a maximum of $N$ pipelines in a reduction system. Then the minimal evaluation time of an $n$ term vector can be expressed as follows:

$$T_{min} = min\left\{T_M^{re}(n,r)\,\middle|\,r=1,2,\ldots,N\right\} \tag{10}$$

We are interested in the value of $r$ for which Eq. (10) is reached. This is not so simple, since the function in Eq. (7) has many discontinuities and is difficult to handle analytically. Instead of that we will try to approach Eq. (10) by the following method. First Eq. (10) is replaced by a continuous function by means of canceling the ceiling signs. Substitution of the SM-method or AM-method then gives

$$Tc_M^{re} = \frac{n}{r}+q\,log\,r+q-1+\begin{cases} q\,log\,q, & \frac{n}{r}\geqslant q \\ q\,log(\frac{n}{r}), & \frac{n}{r}<q \end{cases} \tag{11}$$

Differentiating (11) with respect to $r$ leads to

$$\frac{d(Tc_M^{re})}{dr} = \begin{cases} \frac{-n}{r^2} + \frac{q}{r.ln2} & if\ \frac{n}{r}\geqslant q \\ \frac{-n}{r^2}, & if\ \frac{n}{r}<q \end{cases} \tag{12}$$

The minimum of (11) is reached at either $r_1 = min\{(n.ln2)/q,N\}$ or $r_2=min\{n,N\}$ (remember that $N$ is the maximum number of available pipelines). Whether $r_1$ or $r_2$ is chosen depends on the minimum of $T(n,r_1)$ and $T(n,r_2)$. The approximation of $r$ is denoted as

$$r_{appr}(n)=opt(r_1,r_2)=opt\left(min\left\{\frac{n.ln2}{q},N\right\},min\{n,N\}\right) \tag{13}$$

where $opt(r_1,r_2)=\{r\,|\,T(r)=min(T(r_1),T(r_2))\}$. It is shown in Fig. 5a and 5b that equation (13) is a reasonable approximation. In these curves the optimal and approximated values of $r$ are shown as a function of $n$, with as parameter values $N=16$ and $q=7$. Fig. 5a also shows a peculiar behavior of the reduction system. The optimal and approximated optimal value of $r$ behave quite discontinuously. The curve jumps up and down and gives large differences in $r$ for some nearby values of $n$. In Fig. 5b it is shown that despite the wild behavior of the curve in Fig. 5a, the number of processing cycles is a monotone increasing function of $n$. If we fix the choice of $r_{appr}$ to $r_{appr}=r_1$, the curves of Fig. 6a and 6b result. The latter choice $r_{appr}$ is a monotone function of $n$, but the deviation from the minimal computation time is slightly larger. Simulations with other values of the parameters $N$ and $q$ globally give the same results [7].

## IV. Computation time and efficiency, the DI-ratio

We will now face the question whether the choices of $r$ as derived in the previous section are efficient choices of $r$. The only thing that the formula (13) guarantees is that the choice of $r$ is such that the computation time is within a certain bound of the minimal computation time. This is not necessary an efficient choice of $r$. Fig. 7 shows the behavior of the computation time of a vector with length $n=320$ on $r$ pipelines, with $q$ segments each. It shows that the computation time decreases fastly with an increase of $r$ in the first part of the curve, but soon the curve flattens and the improvement becomes insignificant. This implies that in order to obtain a minimal computation time, many more pipelines are required beyond the point where the computation time decreases significantly as a result of an increase in the number of pipelines. For example, if $q=8$ and $n=320$, the optimal choice of $r$ is $r=32$ (with $N=32$), however, above the values of $r$ in the region $(4,8)$, no significant improvements of the computation time are obtained. At $r=8$, the computation time is $T(320,8)=95$, and at $r=32$, the computation time is $T(320,32)=81$, yielding a $10\%$ improvement in speed at the cost of $4$ times the number of pipelines. Therefore, the minimum computation time criterion leads to a very low processor-efficiency.

For practical purposes a different criterion on the choice of $r$ is needed. We will try to establish a criterion for determining $r=r_0$, such that beyond the point $r_0$ no acceptable improvement in speed can be made.

Normally, we could use an efficiency measure, such as $\eta(r)=T(1)/(r.T(r))$. However, $\eta(r)$ is strongly nonlinear and the above definition leads to cumbersome formulas. Moreover, an excessive number of processors may be needed to meet an efficiency criterion (see also the discussion at the end of point 2 of this section). Instead of that, we will introduce the Doubling Improvement ratio (DI-ratio). We are interested in the relative speed improvement (DI-percentage) when the number of pipelines is doubled. The DI-improvement can be expressed as follows:

$$k_{DI} = \frac{T(n,r)-T(n,2r)}{T(n,r)} \tag{14}$$

Thus, $k_{DI}.100\%$ improvement in speed is obtained by doubling the number of pipelines. In general, $k_{DI}$ will be in the range $(0,\ 0.5)$.

The use of the DI-ratio as an efficiency measure has the advantage that it is a better match to the behavior of the computation time curve of a vector reduction, it smoothens discontinuities, and leads to formulas, which can be analyzed.

In the following the behavior of $k_{DI}$ for a system using the SM-method of reduction will be analyzed first. Three cases are considered:
1. $1\leqslant\lceil n/r\rceil\leqslant q$, 2. $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$, and 3. $\lceil n/2r\rceil\geqslant q$.

1. $1\leqslant\lceil n/r\rceil\leqslant q$. For $\lceil n/r\rceil=1$, substitution of Eq. (4) and Eq. (7) into (14) leads to a negative improvement, so we are only interested in the range of $2\leqslant\lceil n/r\rceil\leqslant q$. It can be proved that $\lceil log\,r\rceil - \lceil log\,2r\rceil=-1$ and for $\lceil n/r\rceil>1$ , $\lceil log\lceil n/r\rceil\rceil - \lceil log\lceil n/2r\rceil\rceil=1$ [7]. By using these properties, substitution of Eq. (4) and (7) into Eq. (14) leads to

$$k_{DI} = \frac{\frac{1}{2}.2^{\lceil log\lceil n/r\rceil\rceil}}{2^{\lceil log\lceil n/r\rceil\rceil}+q.\left\lceil log\left\lceil\frac{n}{r}\right\rceil\right\rceil+q.\lceil log\,r\rceil+q-1} \tag{15}$$

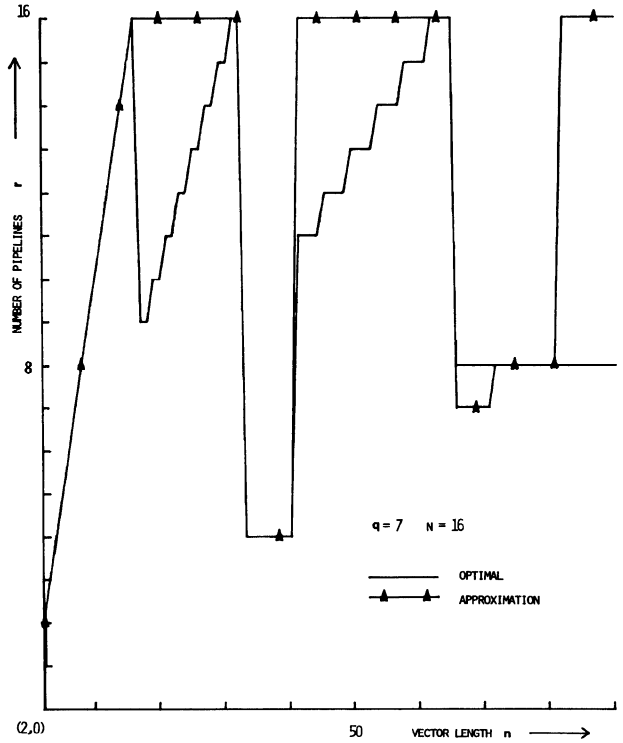


*Fig. 5a. The number of reduction processors as a function of the vector length.*

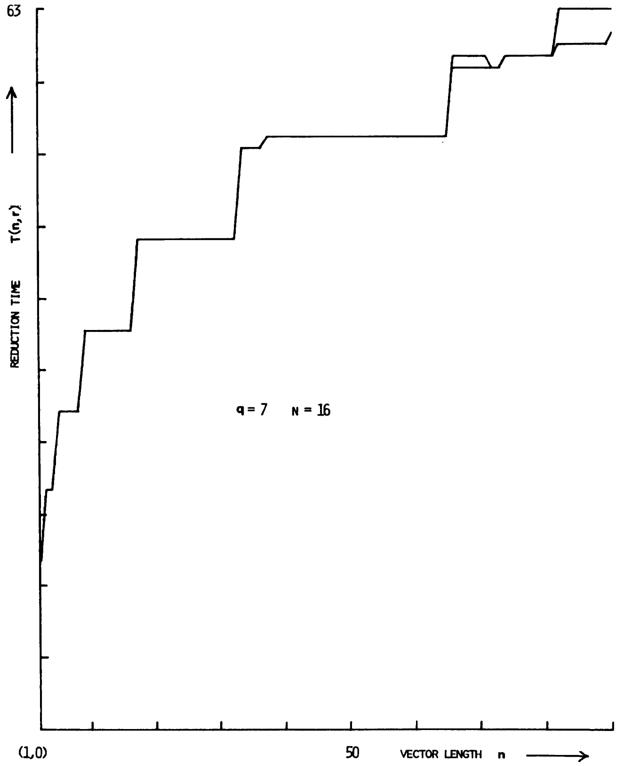


*Fig. 5b. The reduction time as a function of the vector length.*

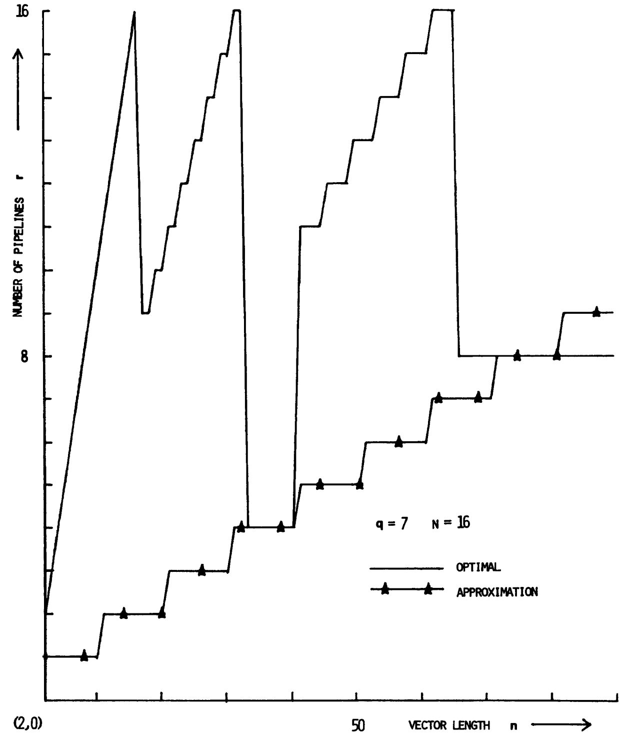


*Fig. 6a. The number of reduction processors as a function of the vector length, if $r_{appr}=r_1$.*

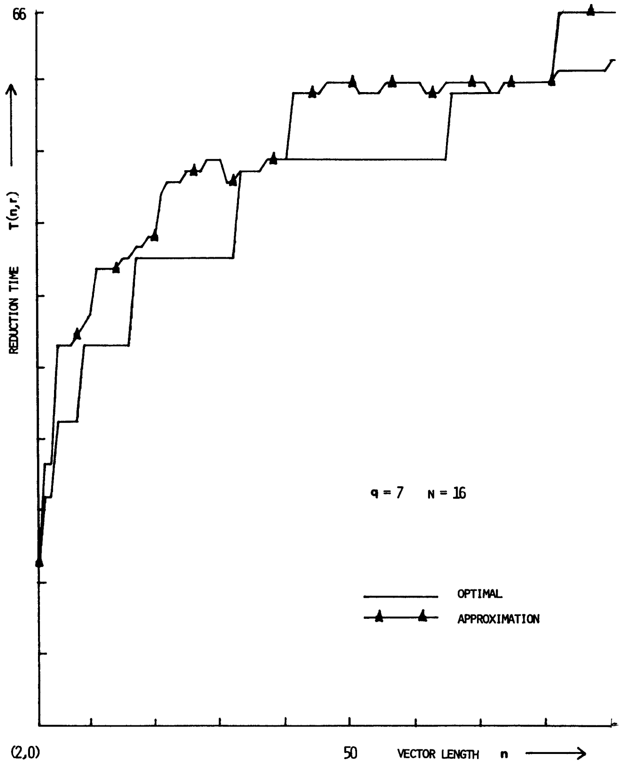


*Fig. 6b. The reduction time as a function of the vector length, if $r_{appr}=r_1$.*

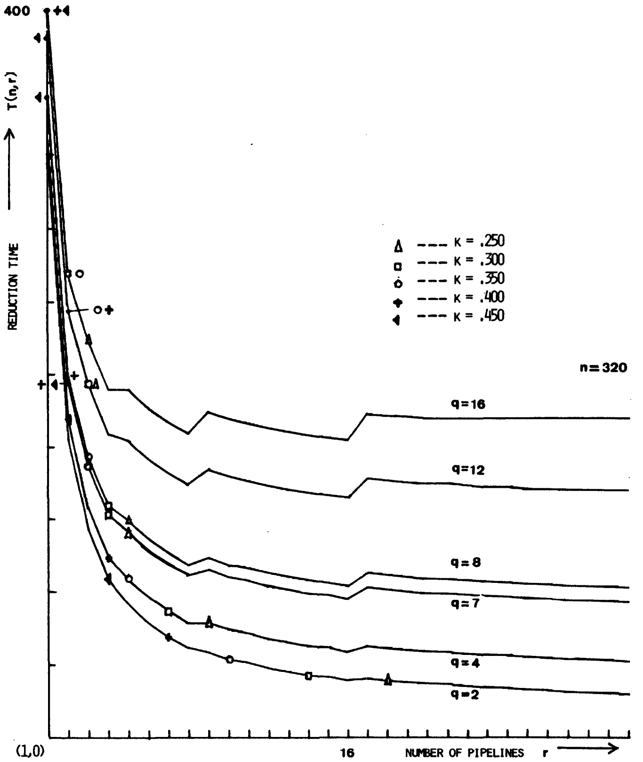


*Fig. 7. The reduction time as a function of the number of reduction processors for several values of q.*

Let $i=\lceil \log\lceil n/r\rceil\rceil$, and then differentiating $k_{DI}$ in Eq. (15) with respect to $i$ yields $\partial k_{DI}/\partial i>0$ for $i\geqslant 0$. Thus, $k_{DI}$ is maximal at $\lceil n/r\rceil=q$ for $1\leqslant\lceil n/r\rceil\leqslant q$, i.e.,

$$k_{DI}\leqslant\frac{\frac{1}{2}\cdot 2^{\lceil \log q\rceil}}{2^{\lceil \log q\rceil}+q.\lceil \log q\rceil+q.\lceil \log r\rceil+q-1} \qquad (16)$$

It also holds that $2^{\lceil \log q\rceil}\leqslant 2q-2$ [7], therefore, from Eq. (16) it follows

$$k_{DI}\leqslant\frac{q-1}{3q-3+q.\lceil \mathit{loq}r\rceil+q.\lceil \log q\rceil} \qquad (17)$$

For $r=1$ and $q\geqslant 2$, $k_{DI}\leqslant 0.2$, and for $q\geqslant 4$, $k_{DI}\leqslant 0.18$. For $r\geqslant 2$, this figure is $k_{DI}\leqslant 0.14$ for both $q\geqslant 2$ and $q\geqslant 4$. Therefore, no impressive improvements can be achieved.

Furthermore, since the inequality (17) gives only an upper bound, the actual values of $k_{DI}$ may be much smaller. Thus, we may summarize the above consideration as follows: for $1\leqslant\lceil n/r\rceil\leqslant q$, doubling the number $r$ to $2r$ leads to a speed up of less than $20\%$.

<u>2. $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$</u>. For $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$, after some substitution, equation (14) can be transformed into

$$k_{DI}=\frac{\left\lceil\frac{n}{r}\right\rceil-2q+q(\lceil \log q\rceil-\left\lceil \log\left\lceil\frac{n}{2r}\right\rceil\right\rceil)+2^{\lceil \log q\rceil}-2^{\lceil \log\lceil n/2r\rceil\rceil}}{\left\lceil\frac{n}{r}\right\rceil+q\lceil \log r\rceil+2^{\lceil \log q\rceil}+q\lceil \log q\rceil-1} \qquad (18)$$

From $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$, it follows $q/2\leqslant\lceil n/2r\rceil\leqslant q$, this means that either a. $\lceil \log\lceil n/2r\rceil\rceil=\lceil \log q\rceil-1$ or b. $\lceil \log\lceil n/2r\rceil\rceil=\lceil \log q\rceil$.

a. Substitution of $\lceil \log\lceil n/2r\rceil\rceil=\lceil \log q\rceil-1$ in Eq. (18) leads to

$$k_{DI}=\frac{\left\lceil\frac{n}{r}\right\rceil-q+\frac{1}{2}\cdot 2^{\lceil \log q\rceil}}{\left\lceil\frac{n}{r}\right\rceil+q.\lceil \log r\rceil+2^{\lceil \log q\rceil}+q.\lceil \log q\rceil-1} \qquad (19)$$

Since $r\geqslant 1$, $\lceil n/r\rceil\leqslant 2.\lceil n/2r\rceil\leqslant 2.2^{\lceil \log\lceil n/2r\rceil\rceil}=$ $=2^{\lceil \log q\rceil}$, and (19) is an increasing function of $\lceil n/r\rceil$, it follows that

$$k_{DI}\leqslant\frac{2^{\lceil \log q\rceil}-q+\frac{1}{2}\cdot 2^{\lceil \log q\rceil}}{2^{\lceil \log q\rceil}+2^{\lceil \log q\rceil}+q.\lceil \log q\rceil-1} \qquad (20)$$

For $q\geqslant 2$, $q$ can be written as $q=2^s+t$ with $s\geqslant 0$ and $1\leqslant t\leqslant 2^s$; substitution of $q=2^s+t$ in (20), leads to

$$k_{DI}\leqslant\frac{2^{s+1}-t}{2.2^{s+1}+(2^s+t)(s+1)-1}\leqslant\frac{2^{s+1}-1}{2^{s+2}+(2^s+1)(s+1)-1} \qquad (21)$$

After some calculation, it follows that for $s\geqslant 0$ the maximum of $(2^{s+1}-1)/\{2^{s+2}+(2^s+1)(s+1)-1\}$ is $0.233$ at $s=2$. Therefore, it holds $k_{DI}\leqslant 0.233$.

b. Substitution of $\lceil \log\lceil n/2r\rceil\rceil=\lceil \log q\rceil$ in (18) leads to

$$k_{DI}=\frac{\left\lceil\frac{n}{r}\right\rceil-2q}{\left\lceil\frac{n}{r}\right\rceil+q\lceil \log q\rceil+2^{\lceil \log q\rceil}+q\lceil \log q\rceil-1} \qquad (22)$$

from $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$ and $\lceil n/r\rceil\leqslant 2.\lceil n/2r\rceil$, it follows that $k_{DI}\leqslant 0$.

From a and b, it has been shown that $k_{DI}\leqslant 0.233$ for $q\geqslant 2$ and $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$. Therefore, we may now conclude that for $1\leqslant\lceil n/r\rceil\leqslant q$ and $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$, the DI-percentage is less than $23.3\%$ for $r\geqslant 1$ and $q\geqslant 2$. For $r\geqslant 2$ and $q\geqslant 2$, this DI-percentage is even smaller, having a value of less than $20\%$.

From the above consideration, we conclude that in the cases of $1\leqslant\lceil n/r\rceil\leqslant q$ and $\lceil n/2r\rceil\leqslant q\leqslant\lceil n/r\rceil$, the DI-percentages for the SM-method are below $23.3\%$ for $r\geqslant 1$ and $q\geqslant 2$, and below $20\%$ for $r\geqslant 2$ and $q\geqslant 2$. For the AM-method, a similar analysis can be made. It can be shown that in this case the DI-percentage is below $20\%$ for all $q\geqslant 2$ [7].

In order to get an interpretation about the consequences of such a low DI-percentage, we want to discuss shortly the relationship between the DI-improvement and the processor-efficiency. It is a well-known fact that the processor-efficiency in a multi-processor system is lower than a single processor system as a result of the intercommunication overhead (which is the price paid for greater speed). In the following, we will use the reduction time $T(r=1)$ as the reference to define the processor-efficiency as

$$\eta(r)=\frac{T_M(1)}{r.T_M(r)}\cdot 100\% \qquad (23)$$

where $M$ denotes the reduction-method, and $T_M(r)$ is the reduction time under method $M$ when $r$ pipelines are involved in the computation. The relationship of the DI-percentage and processor-efficiency can be expressed as

$$\eta(2r) = \frac{\eta(r)}{2.(1-k)} \tag{24}$$

Further, it is shown in [7] that $\eta(r) \leqslant 65.2\%$ for $r=2$, $1 \leqslant \lceil n/r \rceil \leqslant 2q$ and $q \geqslant 2$ for both SM- and AM-method. Therefore, from the fact that $k \leqslant 23.3\%$, it follows $\eta(2r) < 0.652/(2^* 0.767) = 42.5\%$ for $r \geqslant 2$. Thus, the processor-efficiency is less than $42.5\%$ for $1 \leqslant \lceil n/r \rceil \leqslant q$ and $\lceil n/2r \rceil \leqslant q \leqslant \lceil n/r \rceil$ with $q \geqslant 2$ and $r > 2$. To reduce a vector of length $n$, the number of (dyadic) operations is equal to $(n-1)$. If we use the definition $\eta(r) = (n-1)/r.T_M(r)$, then we will have $\eta(r) < 50\%$ for $1 \leqslant \lceil n/r \rceil \leqslant 2q$ and $q \geqslant 2$ for both SM- and AM-method, leading to $\eta(2r) < 30.6\%$ for $r \geqslant 1$. This processor-efficiency is far too low, so it should be avoided. Thus, for a given problem with a vector length $n$, we should choose $r$ such that $\lceil n/2r \rceil \geqslant q$, in order to obtain a reasonable processor-efficiency (in the case of $n \leqslant 2q$, $r=1$ is an obvious choice).

3. $\lceil n/2r \rceil \geqslant q$. Substitution of Eq. (7) with the SM-method into Eq. (14), it yields

$$k_{DI} = \frac{\lceil n/r \rceil - \lceil n/2r \rceil - q}{\lceil n/r \rceil + q.\lceil log r \rceil + q.\lceil log q \rceil + 2^{\lceil log q \rceil} - 1} \tag{25}$$

The asymptotic value of $k_{DI}$ is $50\%$ as $n/r \rightarrow \infty$. However, the requirement of $k_{DI}=50\%$ is not realistic, even undesirable for the sake of computing speed. If we require that $k_{DI} \geqslant 0.45$, then the usage of more than one reduction pipeline is only possible when $n$ is very large. So, our aim is to determine a reasonable compromise between speed and processor-efficiency.

We want to determine a choice $r=r_0$ as a function of $n$ under a given condition $k_{DI}=k$ $(k \in [0.25, 0.5))$. It is not trivial to solve Eq. (25) and to express the DI-choice $r_0$ as an explicit function of $k_{DI}$. Furthermore, it is desirable to have a simple formula for the determination of $r_0$.

By canceling all the ceiling functions in Eq. (25), and substituting $r=r_0$ and $k_{DI}=k$, the following approximation is obtained:

$$r_0 \simeq \frac{(0.5 - k).n}{q(1+k+k log r_0 + k log q)} \tag{26}$$

Eq. (26) is an implicit function of $r_0$. We want to find a simple formula from which an approximation to $r_0$ can be easily calculated.

Substitution of $r_0=\alpha.n$ into Eq. (26), gives

$$\alpha \simeq \frac{0.5 - k}{q(1+k+k.log\alpha + k.log n + k.log q)} \tag{27}$$

where $\alpha$ is an unknown parameter which is a function of $k$, $q$ and $n$.

For a given vector of length $n$, $\alpha$ in Eq. (27) can be computed by means of one of the known numerical root finding methods (e.g., Newton-method). However, it is impractical to iterate $\alpha$ before every vector reduction operation (it may even take more time than the reduction itself!). From Eq. (27), we have learned that $log\alpha$ varies very smoothly as $n$ increases; it is approximately a function of $log(log n)$. One solution is to approximate $log\alpha$ by $a+b.log(log(n))$, where the parameters $a$ and $b$ can be determined by means of the least square error method [7].

Thus, by using the approximation function of $log\alpha$, Eq. (26) is transformed to

$$r_0 = \frac{c_1.n}{c_2+c_3 log(log(n)) + c_4 log(n)} \tag{28}$$

where $c_1=(0.5-k)$, $c_2=q^*(1+k+k.a+k log q)$, $c_3=q.k.b$ and $c_4=q.k$ with $k=k_{DI}$. All $c_i$'s are constants when a choice of $k$ is made. Simulations show that the approximation formula (28), together with the calculated values of $a$ and $b$, gives satisfactory results (error between Eq. (28) and Eq. (26) is less than 5%). Furthermore, Eq. (28) is valid for both the SM-method and the AM-method.

The computation of $log(n)$ and $log(log(n))$ can be done in a simple way by means of the following approximation scheme: determine the position $P_{MSB}$ of the most significant bit (MSB) of $n$; then $P_{MSB}$ is an approximation of $log(n)$ with an error of less than $1$. Another approximation with an error of less than $0.585$ can be obtained by looking at the bit next to the MSB, if this bit is $1$ then $P_{MSB}+1$ is used as the approximation for $log(n)$, otherwise, $P_{MSB}$ is used. Applying now the same process to the obtained result of $log(n)$, an approximation value of $log(log(n))$ is determined.

Fig. 7 shows a plot of the reduction time $(T(n,r))$ as a function of the number of pipelines $(r)$ involved in the processing of a given vector of length $n$. The points related to several different $k_{DI}$ values are shown by means of the markers. The numerical results in Fig. 7 also show, that a choice of $k_{DI} < 25\%$ will only greatly increase the use of the number of pipelines and result in very low processor-efficiency as has already been pointed out earlier.

## V. Performance

Generally, in vector processing a parallel pipelined system performs better than a parallel non-pipelined system. Suppose a non-pipelined processor has a cycle time of $t_{nonp}=s$ $ns$, then with the same technology a pipeline cycle time of $t_{pipe}=(s/q + c)$ $ns$ can be realized, where $q$ is the number of segments of the pipelined processor, and $c$ represents the delay of a latch which is introduced to hold the partial result between each segment. For example, consider a 6 segment adder (like the one in the Cray X-MP) having a cycle time of $t_{pipe}=9.5$ $ns$, then in the ideal case a non-pipelined adder has a cycle time of $t_{nonp}=45$ $ns$ (i.e. assuming that the delay of a latch is about $2.0$ $ns$ for the pipelined case). However, in reality, the ratio $R_\infty$ of the maximum vector processing rate and the maximum scalar processing rate is usually in the order of $10$ (e.g., $R_\infty=13$ for Cray-1 [8]), where $R_\infty$ is defined as $R_\infty = r_{\infty v}/r_{\infty s}$ [8], and $r_{\infty v}$ and $r_{\infty s}$ are the maximal vector and scalar processing rates respectively. Thus, in reality the cycle time of a non-pipelined (general purposed) processor will have a cycle time much greater than the ideal cycle time of *45 ns* in our example. This is due to the fact that instructions must be fetched and decoded for each operation before performing the operation itself, whereas this is not the case for a pipelined processor. The performances of the non-pipelined approach and the pipelined approach are shown in Table I for a system of *8* processors. The computation speeds are listed for different values of $k$. In Table I, the performance of both the ideal parallel non-pipelined system and the practical non-pipelined system with $R_\infty=10$ are shown. $S_{pipe}$ is the speed of the pipelined approach, $S_{nonp}$ and $S_R$ are the speed of the ideal nonpipelined approach and the practical nonpipelined approach respectively. From the table, it is clear that the pipelined system outperforms the ideal non-pipelined system, except for very small vector lengths, and the parallel pipelined system outperforms the practical non-pipelined system in all the cases shown in the table.

## VI. Conclusion

In this paper, we have studied vector-reduction techniques in a parallel arithmetic pipeline processing environment. It has been shown that with a simple partitionable linear array structure, reduction algorithms can be implemented efficiently. The proposed approach performs the same as a binary-tree network in the reduction of an $n$-element vector.

Regularity and fault tolerance of a system are important criteria to VLSI implementations, the partitioned linear array approach satisfies these two criteria.

The minimal reduction time in a system consisting of $N$ pipelines and an approximation method to achieve the minimal reduction time also have been analyzed. It is known that parallelism and processor-efficiency are tradeoffs. In a multi-pipeline system, the issue of processor-efficiency is a more important problem than in the non pipelined cases, since the time overhead in merging the partial results of each pipeline ($=q\cdot\lceil log r\rceil$, see Fig. 3) is larger. The DI-ratio has been introduced as a means to be able to make a trade-off between computing speed (parallelism) and processor-efficiency.

The DI-ratio can be used to calculate a proper choice of the number of pipelined processors, given the typical maximal vector length of the operations the system will be dealing with, and a required processor efficiency.

## VII. References

1. D.J. Kuck, *"The structure of computers and computation"*, Wiley, N.Y., 1978.

2. P.M. Kogge, *The architecture of pipelined computers,* McGraw-Hill, N.Y., 1981.

3. L.M. Ni, K. Hwang, "Vector-reduction techniques for arithmetic pipelines", *IEEE Transactions on Computers*, vol. C-34, no.5, May 1985.

4. L.M. Ni, K. Hwang, "Vector reduction methods for arithmetic pipelines", *Proceedings 6th Symposium on Computer Arithmetic*, June 1983.

5. ---, *Cray-1 computer systems*, reference manual, Cray Research Inc, 1977.

6. ---, *CDC Cyber 200 model 205*, technical description, Control Data Corporation, November 1980.

7. H.X. Lin, "Multi-operand processing: architectures and performances", *M.S. Thesis*, Department of Mathematics and Computer Science, Delft University of Technology, 1986.

8. R.W. Hockney, and C. R. Jesshope, *Parallel computers*, Adam Hilger, Bristol, 1981.

9. C.S. Raghavendra, A. Aviziens, and M.D. Ercegovac, "Fault tolerance in binary architectures", *IEEE Transactions on Computers*, vol. C-33, no.6, June 1984.

*Table I. The performance of a parallel arithmetic pipeline system and a non-pipelined parallel system is compared for different loads (the unit is MFLOPS); the number of processors N=8, and q=6.*

| | n = 5 | | | n = 15 | | | n = 100 | | | n = 1000 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $S_{pipe}$ | $S_{nonp}$ | $S_R$ | $S_{pipe}$ | $S_{nonp}$ | $S_R$ | $S_{pipe}$ | $S_{nonp}$ | $S_R$ | $S_{pipe}$ | $S_{nonp}$ | $S_R$ |
| k= .25 | 13.6 | 16.7 | 8.4 | 35.9 | 19.4 | 9.8 | 127.1 | 40.4 | 20.4 | 622.2 | 162.6 | 82.2 |
| k= .30 | 13.6 | 16.7 | 8.4 | 35.9 | 19.4 | 9.8 | 127.1 | 40.4 | 20.4 | 622.2 | 162.6 | 82.2 |
| k= .35 | 13.6 | 16.7 | 8.4 | 35.9 | 19.4 | 9.8 | 127.1 | 40.4 | 20.4 | 622.2 | 162.6 | 82.2 |
| k= .40 | 13.6 | 16.7 | 8.4 | 35.9 | 19.4 | 9.8 | 82.7 | 20.6 | 10.4 | 622.2 | 162.6 | 82.2 |
| k= .45 | 13.6 | 16.7 | 8.4 | 35.9 | 19.4 | 9.8 | 82.7 | 20.6 | 10.4 | 365.1 | 82.6 | 41.7 |

# SHARED PIPELINES: EFFECTIVE PIPELINING IN MULTIPROCESSOR SYSTEMS

Amos R. Omondi
J. Dean Brock

Department of Computer Science
University of North Carolina at Chapel Hill

## Abstract

This paper discusses pipelining in the context of multiprocessor systems. It is proposed that for such an architecture, it is possible to design near-optimal pipeline machines by coupling pipelines, using buffers to balance instruction/data flows, concurrently processing several instruction streams, and allowing out-of-order execution of instructions within a stream. We have investigated the issues involved in the design of such a machine by using a uniprocessor pipeline machine as a basis for the type of architecture proposed.

## Introduction

Pipelining has long been a method of obtaining high throughput in uniprocessor systems by overlapping the execution of several instructions; an example of such an arrangement being that shown in the block diagram of Figure 1 in which each stage of the pipeline performs one of the phases in the sequence involved in processing an instruction. Ideally the throughput of such a system is one result per pipeline beat. In practice, however, it is rarely the case that this ideal throughput is achieved. Instructions requiring the transfer of control usually cause severe disruption in the flow of instructions; similar effects occur, although to a lesser extent, in the operand-fetch stage, as a result of the methods employed to resolve conflicts, and a multitude of other (relatively minor) reasons.

In addition to pipelining, performance may be gained through the use of multiple processors; examples of machines which combine extensive pipelining and multiple processors being the S-1 [14] and the HEP[ 11]. This paper deals with the subject of pipelining in the context of multiprocessor systems. In such a case, we believe that it is possible to overcome many of the problems inherent in pipelining by "sharing" pipelines between different instruction streams and using buffers to balance flows through the pipelines. Such an architecture is shown, at a fairly high level, in the diagram in Figure 2. The main idea here is to take several pipelines of the type shown in Figure 1 and to couple them through inter-stage buffers (capable of holding several instructions) in such a manner that during peak processing rates, the flow into the buffer exceeds the flow from the buffer. Thus when processing instructions that would normally cause a disruption in the rate of inter-stage flow, the overall effect would be that while the flow out of some stage would fall to below maximum, the succeeding stage would continue to accept instructions at the maximum rate (by processing instructions buffered during peak flow); strictly, therefore, buffers are not necessary at the output of every stage as implied by Figure 2. The ability to process several instruction/data streams also has many benefits which are discussed below. The same techniques can be applied to data streams in order to ensure that pipelined vector processing units do not experience any performance degradations as a result of disruptions in the delivery of data.

The type of architecture most closely related, in as far as the use of buffers goes, to what we propose is the HEP although essentially the same reasoning also lies behind the use of queues in PIPE [2] and a related scheme, in which several independent processors (cyclically) share a buffer, has also been described by Goto and Shimizu [3]. We have attempted to identify the issues that are involved in the design of a more practical machine than that depicted by the block diagram of Figure 2 and have been concerned with the complexity of such a task relative to that of designing a multiprocessor system consisting of independent pipelines. For this purpose we chose a practical pipelined uniprocessor, the MU5 [8], as a starting point and attempted to develop a shared-pipeline multiprocessor from this.

### The MU5 Processor

In this section we briefly describe the uniprocessor that has formed the basis of our studies [9]. A high-level organization of the MU5 processor is shown in Figure 3. It consists of two major pipelines, the Primary Operand Pipeline (PROP) which is largely concerned with the accessing of primary (named) operands and the Secondary Operand Pipeline (SEOP) which is largely concerned with the accessing of secondary operands such as data structure elements. Instructions fetches are initiated by the Instruction Buffer Unit (IBU) to which the store returns upto eight instructions at a time; IBU then supplies the Primary Operand Unit at the rate of about one instruction per beat (peak). IBU also contains a branch prediction mechanism, the Jump Trace. The B-Unit executes fixed point arithmetic instructions, such those required for computing array indices, and some organisational instructions. The A-Unit executes floating point and fixed point arithmetic instructions.

PROP is a five-stage pipeline: *Initial Decode* in which sufficient decoding is done to isolate the name and select a base register (Stack Pointer, Name Base, etc.), *Add Name to Base Register* in which a name is added to the contents of a base register; *Associate Address* which is the first stage of accessing the Name Store (a "cache" for named operands); *Read Value*, the second part of Name Store access; and *Assemble Operand* in which the operand is assembled and the Program Counter is incremented.

SEOP consists of three main stages: *Descriptor Addressing Unit* (Dr) generates secondary addresses using named descriptors from PROP and the output of the B-Unit as modifier, *Operand Buffer Store* (OBS) makes store requests and buffers operands returned, and *Descriptor Operand Accessing Unit* (Dop) performs masking and shifting to select the required element.

## A Shared Pipeline Multiprocessor

In this section we present a shared-pipeline multiprocessor. The philosophy underlying the architecture is very similar to that of the HEP computer [11]; it centers around the idea of processing several instruction/data streams in a single pipeline and can be viewed as one that develops the multiprocessing-pipelines concept to its ultimate conclusion in that several instructions from the same stream may be processed concurrently and possibly out of sequence.

As previously stated, the main driving problem here is

0190-3918/86/0000/0511 $01.00 © 1986 IEEE

that of pipeline flow disruptions. Conventional pipelined machines have dealt with the problem of flow disruptions by designing hardware to limit the effects of these within individual instruction streams; an example in point being branch prediction strategies [7,10]. The success of these has, however, been limited by the inherent nature of pipelining and of the instructions being processed. Considering conditional control transfers, for example, it can be observed that when both the instruction setting condition codes and the control transfer order fall within the so-called Gulf of Ignorance [5], i.e. the separation between the pipeline entry and the execution units, the latter instruction is unresolvable and such mechanisms as double-fetching (employed in the IBM 360/91 [1]) or the prediction Jump Trace of MU5 must necessarily be of limited success. Pipeline sharing is based on the observation that disruptions within an individual stream are acceptable as long as some other stream can be processed to fill the gap created. The main design issue is, therefore, that of being able to switch streams rapidly; in fact, ideally, the delay incurred should be nil. This is precisely what the interleaving of instructions from different streams is intended to achieve. There are also gains to be made if the instructions in a particular stream can be executed out of their initiation order if the data dependencies so permit; this, for example, reduces the number of active processes (streams) that are needed in order to sustain maximum throughput.

From the above description and Figure 3, it will be observed that in an MU5-like pipeline, the points at which the averaged input rate into a stage may exceed the averaged output rate out of the stage are:

- At the output of IBU as a result of processing branch instructions and other organisational instructions. This is the case even with branch prediction and prefetching.
- At the Name Store, since some instructions will give non-equivalence on association.
- As with the Name Store, an instruction may fail to find its operand in the Operand Buffer Store hence the rate of flow into the A-Unit may be disrupted on certain occasions.
- At the Control-Point/Assemble-Operand stage. Since some instructions are executed at this stage, the flow into the stage will, on the average, exceed the outward flow into the arithmetic units.

Thus recalling the diagrams of Figures 1 and 2, buffers can be inserted at these points to balance the flow into and out of the various stages. This would be the obvious organization and represents the architecture that was considered initially. A much better organization turned out to be that shown in Figure 4. In this all the buffers, that one would expect, in the primary stages of the pipelines still exist although in a different form and they, in effect, been moved into the IBUs. This excludes buffering at the end of the primary pipelines; which we have not, for various reasons, investigated at the present. To keep the diagram simple, not all the relevant features of the architecture are shown as will be apparent from the following discussions. Such omissions include connections from the Instruction Issue to the IBUs (via the IBU-PSB network). Although only two inter-stage buffers per stage are shown, a practical system would include as many buffers as necessary to balance the average flows without any undue performance requirements on the hardware. It is also to be expected that the number of Scalar Execution Units will be much smaller than the number of Vector Execution Units; reasonable numbers being, say, 4 and 32 respectively. In general, the number of units in the primary pipelines can be altered independently of the number of units in the secondary pipelines. No claim is made that Figure 4 of itself depicts a practical machine; at this point, we have emphasized concepts over practical matters whenever conflicts existed.

There are fundamental differences between the organization shown and one that would be obtained from a straightforward extension of an MU5-like machine to multiprocessor system. By far, the most apparent is the decision to separate the single primary store into a *Program Store* and a *Data Store*; in fact the latter is further split into a *Scalar Store* and a *Vector Store*. The main reason for this is to reduce the bandwidth requirements of the various interconnections. With a single (interleaved) store an interconnection system with enough bandwidth to support the accessing rates of all the IBUs, the Scalar Units, and the Vector Units may be an impractical proposition. The separation of a (possibly) single data store into two separate stores arose from the decision to have highly pipelined vector processing units; the use of a separate Vector Store, apart form reducing bandwidth requirements, also allows vector processing orders to be implemented in a more natural way.

## Multiple Instruction Buffer Units

Essentially, the system runs several processes concurrently with instructions from each stream being initiated in a different IBU. A Decode module cycles through the IBUs connected to it and interleaves instructions from several streams into the next stage, after the necessary decoding. Whenever an instruction that would normally cause a disruption, e.g. a branch instruction, is decoded, the corresponding IBU is (temporarily) abandoned until it is known that valid instructions may be taken from the IBU, say when a transfer of control actually takes place.

The important point about the use of several IBUs per Decode is that since instructions from different streams are continually being interleaved, the end effect is that streams (as seen by the pipeline stages) as also being changed at the same rate which is also the maximum rate possible. Thus when a disruption occurs within some stream, the necessary change of stream is obtained, in effect, "for free".

No attempt is made to predict control transfers or to explicitly trap loop instructions. Regarding the former, the implications of trying to do otherwise are quite clear: identifying the instructions belonging to some stream for which an incorrect prediction was made and flushing these out is likely to be very messy at best. Hence all instructions entering the pipelines must be guaranteed to complete. Loop-trapping on the other hand would be quite easy to include since in the case of loops an IBU tends to contain the same set of instructions most of the time.

## Name-Associate Units and Name Caches

At the Name-Associate Stage those instructions causing a non-equivalence are held-up until it can be guaranteed that their operands are available; other instructions may also be held-up at this stage in order to resolve conflicts. We have extensively modified [9] the original MU5 Name Store (Cache) to allow a very large number of instructions to execute out of sequence. In additions to modifications to the Name Store itself, an associatively addressed buffer has been added to each cache unit to store instructions that are held-up in order to resolve conflicts and a queue has also been added to hold instructions with pending store requests. The end result, admittedly, is a rather complex organisation that in its present form needs to be simplified.

Instructions for which the association is successful proceed to the next stage, tagged with a specification of a Name Cache unit holding the operand, and have their operands read out. At the Control Points, branch instructions and some organisational instructions are executed and other instructions are issued to a Scalar Execution Unit or a Vector Associate Unit (VAU) and to a Vector Execution Unit, as appropriate.

### Vector Processing Modules

For vector instructions, the VAU units compute addresses and performs the necessary associate operations, operands are obtained from the Vector Cache, (possibly after a store fetch) and the instruction and operands are forwarded to a Vector Execution Unit for processing. The results of a VEU operation are stored in one of the two data caches, depending on its type, while those of a scalar operation are stored in the Name Cache. The modules involved with the processing of vector orders are also another area where we have had to make significant changes to the original organisation. Modifications in the former have been made to allow out-of -sequence execution, and are similar to the changes made to the Name Store, and to allow data-driven communications such as are to be found in the MU6-V prototype [6].

### Miscellaneous

The Instruction Issue Units (IIUs), which are also the location of Control Points, have also turned out to be more complex than their counterpart in the MU5 PROP; a major addition has been to include a unit resembling the CDC 6600 Scoreboard [13]. Some of the added complexity arose from the attempt to maintain compatibility with the MU5 instruction set, primitive architecture, and general philosophy. An example of this is that while compiler-generated code assumes (for simplicity), for example, that only one Scalar Execution Unit exists, IIU on the other hand assigns schedules instructions according to the availability of execution units. This creates many difficulties, particularly when code assumes that the stack will be used to hold temporary values. The same applies to many orders that explicitly use the base registers; since these registers are located in the early stages of the primary pipelines while part of the execution takes place in the IIUs or the Scalar Execution Units, there is some difficulty in providing the necessary communication. Our solution at the moment is to connect the IIU-SEU/VAU network and the PSB-IBU network and to communicate through these, but this is clearly a solution that leaves much to be desired. It, therefore, appears that any solution short of changing the instruction set and/or changing the primitive architecture is likely to end up being rather complex.

Other significant architectural changes that have been necessitated by the sharing of pipelines are: Certain machine registers, such as the Processor Status Register, Program Counter, and Base Registers have to be replicated so that there is one register of each type for each IBU. As a consequence of this, and of the need to provide communication (which takes place via the PSB-IBU network) between Control Points and IBUs, it is necessary for instructions to be tagged both with a process identifier and the identifier of the IBU in which they are initiated.

## CONCLUSIONS

The original inspiration for the architecture described came from [4] although it may now be hard to discern many similarities with what is described therein. The abstract architecture is one that that combines pipelining and multiprocessing and whose main features are: *shared-buffered pipelines with almost no overhead in stream switching* and *no disruptions in the inter-stage flow of instruction/data* and, hence, no performance degradation on all types of orders, *ability to execute instructions out-of-sequence across a wide window in the stream*, and *high-performance vector processing* It is based on a practical uniprocessor machine and the relative complexity/major issues of realizing such an abstract machine have been partially evaluated (detailed in [9]). The main lessons that have been learned from the exercise so far are that: (1) such a machine probably needs to have a both "tailor-made" instruction set and primitive architecture as its basis and (2) there are inherent difficulties in some of the underlying ideas and these needs to be re-examined carefully; the splitting of the data store, for example, has resulted in far more difficulties than were anticipated. These represent the major avenues for any further work. have

## ACKNOWLEDGEMENTS

We have been influenced by the past and present work of Roland Ibbett and Nigel Topham. This includes, in particular, our vector instruction set, which was wholly designed by the latter, and also the same individual's own work on dealing with pipeline flow disruptions [12]. Topham also read early drafts of [9] and trying to answer some of his harder questions has resulted in many improvements. The first author acknowledges the encouragement given to his research by David Plaisted

This work was partially supported by the National Science Foundation under grant number DCR-8406850.

## REFERENCES

[1] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM 360 Model 91: Machine Philosophy and Instruction Handling", *IBM Journal of Research and Development* **11** (1) (1967) 8-24.

[2] J. R. Goodman et al, "PIPE: A VLSI Decoupled Architecture", *Proceedings, 12th Annual International Symposium on Computer Architecture* (1985) 20-27.

[3] E. Goto and K. Shimizu, "Architecture of a Josephson Computer (FLATS-3)", in: N. Inada and T. Soma, Eds., *Symbolic and Algebraic Computation by Computers* (World Scientific, Singapore, 1985) 205-214.

[4] R. N. Ibbett, "Vector Processing", *Proceedings, International Computing Symposium* (1981) 337-347.

[5] R. N. Ibbett, *The Architecture of High Performance Computers* (Springer-Verlag, New York, 1982)

[6] R. N. Ibbett, P. C. Capon, and N. P. Topham, "MU6-V: A Parallel Vector Processing System", *Proceedings, 12th Annual International Symposium on Computer Architecture* (1985) 136-144.

[7] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer* **17** (1) (1984) 6-22.

[8] D. Morris and R. N. Ibbett, *The MU5 Computer System* (Springer-Verlag, New York, 1979)

[9] A. R. Omondi, Sharing Pipelines in Multiprocessor Systems: A Case Study, Internal Report, Department of Computer Science, University of North Carolina at Chapel Hill, 1986.

[10] J. E. Smith, "A Study of Branch Prediction Strategies", *Proceedings, 8th Annual Symposium on Computer Architecture* (1981) 135-148.

[11] B. Smith, "The Architecture of HEP", in: J. S. Kowalik, Ed., *Parallel MIMD Computation: HEP Supercomputer and Its Applications* (MIT-Press, Cambridge, 1985) 41-55.

[12] N. P. Topham, Private Communication, 1986.

[13] J. E. Thornton, *Design of a Computer: The Control Data 6600* (Scott, Foresman and Co., Glenview, 1970)

[14] L. C. Widdoes, Jr. and S. Correll, "The S-1 Project: Developing High-Performance Computers", *Digest of Papers, IEEE Spring COMPCON* (1980) 282-291.

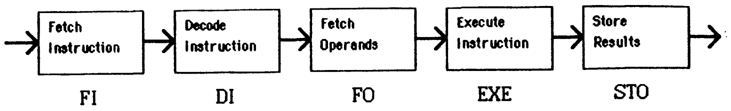


Figure 1.

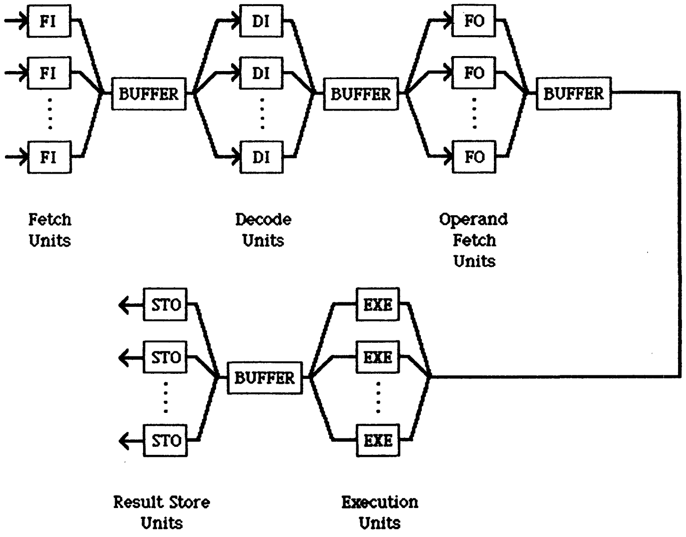


Figure 2.

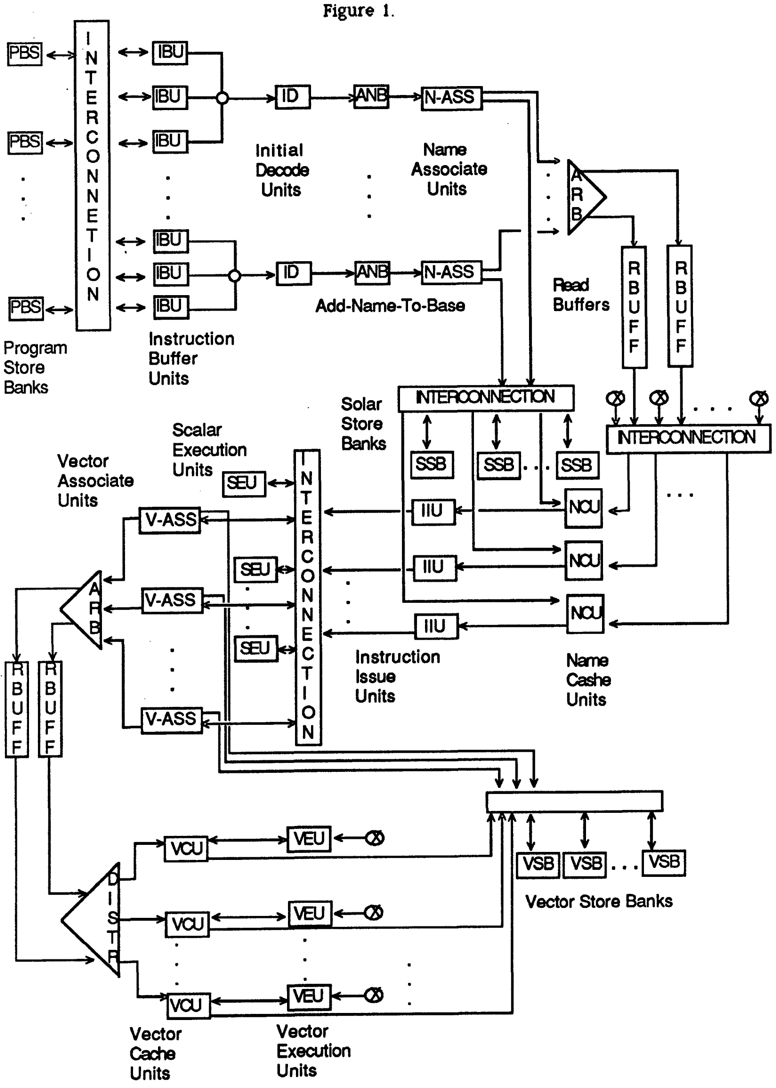


FIGURE 4: A SHARED PIPELINE MULTIPROCESSOR

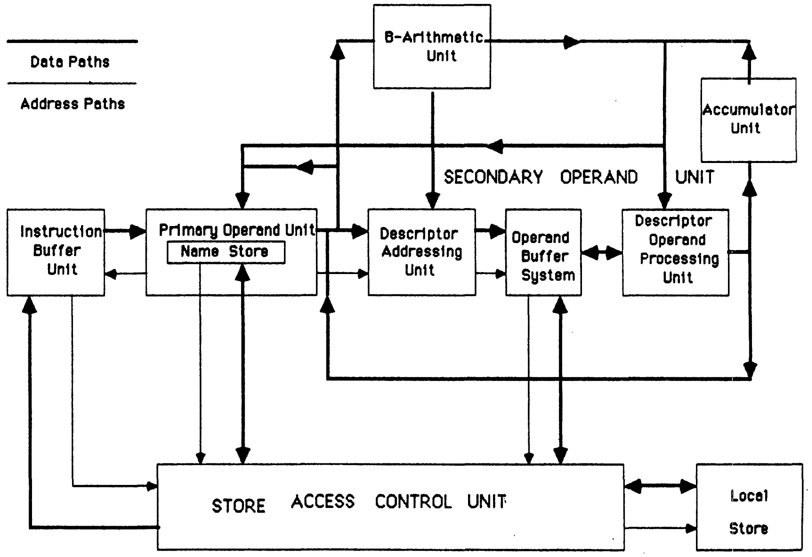


Figure 3: MU5 Central Processor

# PIPELINED REGISTER-STORAGE ARCHITECTURES [1]

Steven R. Kunkel and James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, Wisconsin 53706

## Abstract

Instruction sets of Register-Storage (RS) architectures allow functional operations to take one operand from memory and one from a register. In contrast, Register-Register (RR) architectures always take both operands from registers. A simple, generic instruction set is defined for an RS architecture. A basic RS pipeline structure is developed for studying design and performance characteristics. The resolution of register hazards, placement of the memory access, branch instructions, and handling of stores to memory are examined in detail. For purposes of comparison, a basic RR architecture and pipelined implementation are also developed. The RR and RS models are used to calculate the performance of several of the Lawrence Livermore Loops to gain additional insight into the effect that the architectures have on performance.

## 1. Introduction

Register-Storage (RS) architectures are based on instruction sets that allow functional operations (e.g. floating point addition) to take one operand from memory and one from a register. The result is placed in a register. In contrast, Register-Register (RR) architectures always take both function operands from registers; explicit loads and stores to or from registers must always be used to communicate with central memory. A third class of architectures, those that permit Storage-Storage (SS) operations, can also be defined, although we do not study them here.

The suitability of RR architectures for high performance computation has been clearly demonstrated. The CDC6600 [THO70], CDC7600 [BON69], and the more recent CDC CYBER 180 [CDC84] architectures are based on an RR architecture. In addition the scalar section of the CYBER 205 [CDC81] is an RR architecture, as are both the scalar and vector sections of the CRAY-1S [CRA80], CRAY X-MP [CRA82], and CRAY-2 [CRA85]. Furthermore, the suitability of RR architectures for high performance implementations has not escaped the attention of researchers studying high-speed microprocessor architectures [PAT85], [RAD82].

RS architectures are widely used in practice; the IBM 370 architecture can generally be considered an RS architecture, although it does support some SS operations. Unfortunately, the IBM 370 architecture also contains some features that make high performance implementations difficult (e.g. implicitly-set condition codes, self-modifying code, and precise interrupts) [AND67]. Consequently, based on real implementations, it is difficult to clearly visualize the fundamental structures of pipelined RS architectures and to see the relative advantages and disadvantages of RS architectures for high performance implementations.

In this paper, we propose a simple RS architecture, suitable for pipelining, in order to clearly study the advantages and disadvantages of RS architectures for high performance implementations. To permit comparison, we also describe and discuss RR pipelines. The discussion in this paper is primarily in the context of numeric (floating point) applications, although many of the results and observations apply to other applications.

## 2. Model Architectures

A fundamental RS instruction couples a memory access with a data operation. An RS instruction is of the form $Ri \leftarrow Rj \; op \; (Rk + disp)$. The content of register $Rk$ is added to the displacement to form the memory address from which one of the operands is fetched. There are several other ways that a memory address could be formed, but we use only this mode. An operation, signified by $op$, is performed between the memory operand and a second operand taken from register $Rj$. The result is placed in register $Ri$.

Also included as part of a basic RS architecture are simple register loads, register-register operations, register stores, and branches. The register-register instructions are of the form $Ri \leftarrow Rj \; op \; Rk$ and $Ri \leftarrow Rj \; op \; data$. Here the second operand comes from a register or directly from the instruction. Loads and stores are of form $Ri \leftarrow (Rk + disp)$ and $(Rk + disp) \leftarrow Ri$ respectively. For single operand instructions, the operand can come from either a register, memory, or the instruction. Conditional branches are of the form $BrCond \; Rk + disp$ where a branch condition is specified and the target address is computed by adding the contents of register $Rk$ to the displacement. In all cases the register R0 is tested and the branch decision is based on this test. Condition codes are not used.

As a model RR architecture, we use the RR subset of the above. Hence, it is like the scalar CRAY-1 architecture, except one uniform set of registers is used, not two types.

## 3. Pipeline Structures

### 3.1. The Basic RS Pipeline

Fig. 1 shows a typical implementation. An instruction fetch sequence consists of address generation, memory reference, and instruction decode. The instruction fetch step may contain an instruction cache.

The operand fetch phase consists of reading address registers, operand address calculation, and memory access. Reading registers requires checking for hazards which will be detailed later. In the memory access step, an operand cache may be used. In fact, an operand cache is an important aspect of RS designs and will be discussed further later.

Before an instruction can be issued to the execution unit, register operands must also be obtained. This can be performed either in parallel with or after the memory operand fetch. In our basic pipeline, instructions are placed in a queue, shown in Fig. 1, after completing the memory operand fetch. Note that the queue may be of length 1. The instruction at the front of the queue checks for register hazards, reads register operands, and then issues to the execution unit.

The execution unit may take many forms. It may be a single non-pipelined unit, several parallel non-pipelined units, or several parallel pipelined units. Note however that the type of execution unit used is

[1] This is material based upon work supported by the National Science Foundation under Grant ECS-8207277.

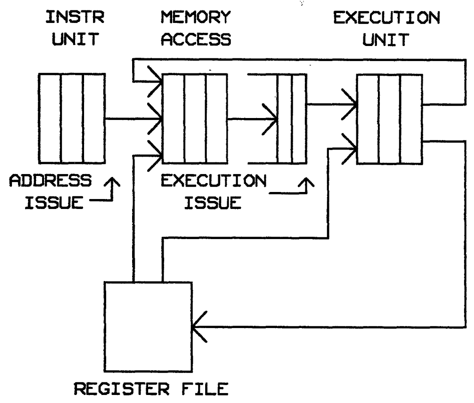


Fig. 1 A typical Register-Storage implementation model

independent of the type of architecture being implemented. The type of execution unit used is based on other issues such as performance and cost. In our study, a parallel pipelined execution unit is assumed to minimize execution time and expose other delays that are inherent in the implementation of the architecture.

Once execution has completed the result needs to be stored. Most instructions return their result to a register, so storage is a simple register file write. However, stores to memory can take place at this point also. In a later section, we consider the tradeoffs involved with performing memory stores at different points in the pipeline.

### 3.2. A Basic RR Pipeline Structure

The basic pipeline phases are much the same as in the RS model. Here, the instruction fetch step is followed by either memory operand access or execution, but not both. Storage to memory is done at the same point as operand access. Fig. 2 shows a typical RR model. As in the RS pipeline, instructions flow through the instruction fetch unit where they are decoded. Following instruction fetch, all register interlocks are cleared and then an instruction is issued to either the execution unit or the memory unit. All registers are read at the same point in the pipeline whether they are for addressing or contain operands. Each of these units returns its result to the register file.

## 4. Design Considerations

### 4.1. Register Hazards

The way register hazards are handled is an important difference between RS and RR implementations. In the RS implementation, there are two points at which registers are read. The first follows the instruction fetch phase; at this point registers for addressing are read. The second follows the memory access phase; here register operands are read. We refer to these as *issue* points. To be more specific, the point where address registers are read will be referred to as *address issue* and the point where operand registers are read will be referred to as *execution issue.*

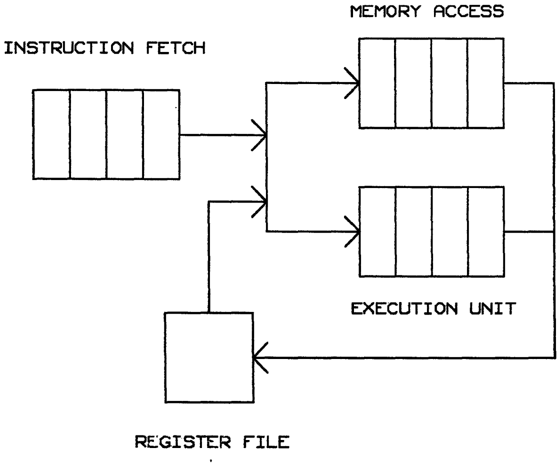


Fig. 2 A typical Register-Register implementation model

In an RR pipeline, where all registers are read at one point, one bit is typically used for each register to indicate that the register is *reserved.* When an instruction is ready to be issued, it checks the reserved bits for all the registers it uses. It is blocked from issuing until they are all free. A result register is reserved when an instruction that writes the register is issued. The reserved bit is cleared when the result register is written.

We first consider a simple-minded extension of the above method for an RS pipeline. Place a separate set of reservation bits at both issue points. The reservation bits at each issue point are handled as with the RR architecture. The first set is held at the *address issue* point, and are checked and set for all result registers as instructions pass. They are also checked for address registers as part of *address issue.* The second set is held at the *execution issue* point and are also reserved by all instructions that write a register. They are checked for both operand and result registers for functional operations before instructions issue to the execution phase. Both reservation bits are simultaneously cleared when the appropriate result register is written.

However, this method limits performance. This can be seen in the following sequence of instructions

(1) R1 <- R2 op R3
(2) R4 <- R5 op R1
(3) R1 <- R6 op R7.

Register R1 is reserved by instruction (1) at *address issue.* Instruction (2) does not read R1 until *execution issue* and is therefore not held up at *address issue.* The problem occurs with instruction (3). Because R1 is already reserved, this instruction must wait at *address issue* until the register is no longer reserved. Otherwise, if it is not held, the reserved bit for R1 will be cleared when instruction (1) finishes. When in fact, R1 should still be reserved for instruction (3). Because only one bit is used to reserve a register at *address issue,* only one instruction that writes R1 can be beyond the *address issue* point at any given time.

This problem can be corrected by using a small reservation counter for each register at *address issue* instead of a single reserved bit. The counter keeps tracks of the number of instructions that have passed *address issue.* The counter is incremented at *address issue* by an instruction that writes the register, and is decremented when the register is actually written. When an instruction needs to read an address register, it must wait until the counter becomes zero. When an instruction needs to write a register, it does not have to wait at *address issue;* it simply increments the counter and proceeds. This method enables the two issue points to make decisions independently, and performance is not restricted. In our later performance study, we use the method just proposed. The counter method for handling register reservations at the *address issue* point was proposed in [AND67]. There are of course, other methods for handling register conflicts but they will not be discussed.

To summarize, register reservation logic is more complex in an RS pipeline than in an RR pipeline. This is because registers must be read in two different places rather than one. In our RS implementation, there is a set of reservation counters, as well as a set of reservation bits; there is only a single set of reservation bits in an RR pipeline. This additional RS complexity is in terms of the amount of logic required, not in terms of the time required to make control decisions. That is, the length of time needed to make an issue decision regarding the availability of registers should be no greater with an RS pipeline.

### 4.2. Memory Access

Because memory accesses are often tied to data operations in an RS architecture, it would seem to be more difficult to schedule code to hide memory access delays. In fact, the organization of the RS pipeline with the memory access phase preceding the execution phase can actually improve performance. In the RS pipeline, instructions may go through the memory access phase and obtain memory operands while previous instructions are waiting for execution. In effect, a RS instruction is given a head start by beginning the memory access ahead of execution operations occurring earlier in the program. In a typical RR implementation this does not occur.

Because load operations sometimes get a head start, cache misses also get a head start, reducing their degradation effect on performance. Fig. 3 shows an example of code in which this occurs. In this example, assume the second load instruction has a cache miss. The first load is issued to memory at the same time in both implementations. In the RR implementation the second instruction waits for the load to complete while in the RS implementation the second instruction follows the load through memory. This causes the second load to wait in the instruction fetch unit in the RR implementation. Whereas, in the RS implementation, the second load is issued to the memory unit without any wait. This gives the cache miss a head start over the same cache miss in the RR implementation.

To counter the above advantage of an RS implementation, there is a corresponding disadvantage that sometimes occurs at the beginning of a segment of code. Fig. 4 shows an example in which this occurs. In the RR implementation, the second instruction in this example starts execution immediately following the load. But in the RS implementation, the second instruction follows the load through the memory unit before starting, causing a delay. Because the three floating point operations are all dependent, the initial delay in the RS implementation is retained, and the entire sequence is slower. As we shall see later, there is a similar effect involving conditional branches. This effect can be reduced by shortening the length of the memory access phase; this is most easily done by using an operand cache.

```
              RS architecture                     RR architecture

start:   R3   <-   (R1 + disp1)          start:   R3   <-   (R1 + disp1)
         R2   <-   R3 +f R4                       R2   <-   R3 +f R4
         R5   <-   R2 *f (R1 + disp2)             R6   <-   (R1 + disp2)
                                                  R5   <-   R2 *f R6
```

Fig. 3 Code in which RS memory access/cache miss gets a head start.

```
            RS architecture                      RR architecture

start:   R5   <-   (R1 + z(12))    start:   R5   <-   (R1 + z(12))
         R6   <-   R2 *f R3                 R6   <-   R2 *f R3
         R7   <-   R6 *f R5                 R7   <-   R6 *f R5
         R8   <-   R7 +f R5                 R8   <-   R7 +f R5
```

Fig. 4 A code segment with lower RS performance.

It should be noted that in the examples given, we have taken a small piece of code out of its original context. If R2 or R3 are dependent on an instruction occurring just before the sequence given in Fig. 4 begins, then the second instruction may be delayed by the same amount in both sequences.

It is our observation from the code we have examined, that overall performance speedups for RS architectures as typified in Fig. 3 are more common that performance losses as in Fig. 4. To summarize, placing the memory access phase ahead of the execution phase leads to performance advantages because memory accesses may be started earlier than they would with an RR pipeline. A disadvantage comes occasionally when execution must sometimes wait longer because of an unnecessary transit through the memory access phase.

### 4.3. Stores to Memory

In an RS pipeline, there are several points at which stores can be issued to memory. First, they can be issued at the same point as loads. This has the advantage of keeping all memory accesses in program order, preventing many hazards involving memory. It also eliminates conflicts for a single memory path; the path is required at only one point in the pipeline (disregarding the instruction fetch path). A major disadvantage is that a store, waiting for the data to be stored, blocks at an early point in the pipeline and holds all the following instructions until the data becomes available. A second disadvantage is that it makes implementation of precise interrupts more difficult because stores may complete before previous instructions in the instruction stream are known to be error-free.

Second, stores can be issued at the same point at which function execution begins. With this method, the store instruction can read its address registers and "reserve" its result address in the memory access phase, then wait for its data at the *execution issue* point. When the store data are available, the instruction must then "steal" a memory access cycle to perform the store. The act of "reserving" its result address amounts to saving the address in a table so that hazards involving later loads from the same address can be resolved. With this second method, the primary advantage is that instructions following the store are not blocked as early in the pipeline. The disadvantage is that control complexity is increased. Memory hazards must be resolved with additional logic, and the memory access path may be used from two different pipeline phases so conflicts may occur.

Third, stores may be sent to memory following the execution phase. Here, registers are read at the same point as in the second method. This method further increases the control complexity of resolving memory hazards by delaying the store to memory. But all instructions would complete in order making the implemenation of precise interrupt easier.

### 4.4. Branches

Up to this point we have said very little about branch instructions, which are a very important aspect of pipeline design. Both RR and RS architectures can be made to handle branches in a similar way. Because of the variety of ways of performing branches, we have chosen a specific one for illustrative purposes. Conditional branches all test register R0. In an RR pipeline, this test occurs at the single issue point, and in an RS pipeline, at the *address issue* point. This assumes that there is hardware in the instruction fetch phase to perform the test of R0 for branches. This eliminates the delay of issuing the instruction to a functional unit to perform the test.

The primary performance advantages and disadvantages are very similar to those already noted for the placement of the memory access phase. The advantage for an RS pipeline occurs when R0 is set early and an independent instruction, preceding the branch, is blocked due to data dependencies. In this case, an RR pipeline holds the branch instruction behind the blocked instruction. In the RS pipeline, the blockage of the independent instruction may occur at the *execution issue* point, so that the branch is not blocked and may be performed earlier.

A potential performance loss occurs when an instruction modifying R0 is followed closely by a conditional branch testing R0. In this case, the branch instruction may block waiting for R0 to be written. The wait can take longer in an RS architecture if the instruction modifying R0 must pass through the memory access phase before being executed as in our implementation model. This can be viewed as part of the penalty one pays in return for earlier execution of conditional branches as described above and for early accessing of memory data as was pointed out in section 4.2.

For the FORTRAN code we have examined, R0 is often set well in advance of a conditional branch that tests it. This is because the branch is typically based on a loop counter which can be incremented early in a loop. Hence, the performance penalty for these kinds of branches is low. Nevertheless, for other types of code where a test is made closely before a branch, an RS pipeline could suffer significant delays for conditional branches.

## 5. Experimental Results

Following are the results of a preliminary study we have performed to test some of the conclusions given above. Because of the limited set of benchmarks and the methods used, no general conclusions should be based solely on these results.

Using the above-defined architectures and the implementation models, the execution time for eight of the Lawrence Livermore Loops [MCM72] were calculated. These are a standard set of small FORTRAN kernels. These eight loops were chosen because a hand compilation method was used, and quite simply, these were the easiest to hand compile. Compiler output from the Cray FORTRAN Compiler was used as a guide, and our hand compilation is at the same level of optimization.

We assumed execution times similar to those in the CRAY-1. For models with no operand cache we assume an 11 clock period memory access time. For models with operand cache, we assume a 3 clock period memory access. We assume the instruction fetch unit contains an instruction cache with a 100% hit ratio. The code is scheduled separately for each architecture and each memory speed to give optimum performance for each case. Given these assumptions, we calculated the time required for one iteration of each loop. The results of the timings are given in Table 1.

These results indicate that six of the eight loops executed in the same time for both architectures regardless of the memory speed used. It would seem that there is little difference between the performance of the two architectures based on these timings. However two of the loops did have different execution times. Loop 2 is a case where the RS architecture is able to take advantage of issuing fewer instructions to perform better than the RR architecture. The RS architecture had five fewer instructions to issue in this loop. Loop 3 also had a different execution time. In this loop, the RS architecture is faster because it reads operand registers at a different place than it executes branch instructions. Since there is no store near the end of this loop, the branch at the end of the loop is not held up waiting for the final result to be stored. In the RR implementation the branch must wait for the last data operation to obtain its operand data before it can be issued. In the RS implementation the branch is able to execute sooner because the last data operation reads operand registers at *execution issue* and does not hold up the branch instruction.

| loop | RR | | RS | |
|---|---|---|---|---|
| | M= 11 | M = 3 | M= 11 | M = 3 |
| 1 | 38 | 30 | 38 | 30 |
| 2 | 45 | 39 | 45 | 37 |
| 3 | 25 | 17 | 19 | 11 |
| 5 | 57 | 49 | 57 | 49 |
| 6 | 57 | 49 | 57 | 49 |
| 7 | 69 | 61 | 69 | 61 |
| 11 | 23 | 15 | 23 | 15 |
| 12 | 23 | 15 | 23 | 15 |

Table 1 Results of Timing of Models. Execution time is in clock periods. M is the memory access time in clock periods.

## 6. Conclusions

RS architectures have three primary performance advantages.

(1) There are fewer instructions; in some code sequences, the limitation of issue rate to one per clock period is important, and in these cases, RS architectures can give better performance.

(2) Placing the execution segment of the RS pipeline after the memory access segment permits memory load instructions to proceed even though an earlier instruction may be blocked at the execution units due to a data dependency.

(3) Placing the execution segment of the RS pipeline after the conditional branch logic permits some conditional branches to proceed even though an earlier instruction may be blocked at the execution units due to a data dependency.

Occasionally, the last two advantages are offset by a performance loss when an execution operation must pass through the memory access segment part of the pipeline before it can begin. Nevertheless, it is our observation that, at least for scientific code, the performance advantages outweigh the disadvantages. Our experimental results show a relatively small difference in performance between the two architectures. Whenever there is a difference, however, the RS pipeline is faster.

The primary disadvantage of RS pipelines is the additional implementation complexity. This comes about because registers are read from at least two different points in the pipeline. This at least doubles the amount of register interlock logic that is needed. On the other hand, the control complexity at the individual stages is not significantly increased, so the clock frequency does not necessarily need to be reduced.

## 7. References

[AND67] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "The IBM/360 Model 91: Machine Philosophy and Instruction-Handling", *IBM Journal of Research and Development*, January 1967, pp. 8-24.

[BON69] P. Bonseigneur, "Description of the 7600 Computer System", *Computer Group News*, May 1969, pp. 11-15.

[CDC81] Control Data Corporation, "CDC CYBER 200 Model 205 Computer System Hardware Reference Manual," Arden Hills, MN, 1981.

[CDC84] Control Data Corporation, *CDC Cyber 180 Computer System Model 990 Hardware Reference Manual*, pub. No. 60462090, 1984.

[CRA80] *CRAY-1S Series Hardware Reference Manual*. HR-0808, June 1980.

[CRA82] Cray Research Inc., *CRAY X-MP Series Mainframe Reference Manual*, HR-0032, Nov. 1982.

[CRA85] Cray Research Inc., *CRAY-2 Computer System Functional Description*, HR-2000, Nov. 1985.

[MCM72] F. H. McMahon, "FORTRAN CPU Performance Analysis", Lawrence Livermore Laboratories, 1972.

[PAT85] David A. Patterson, "Reduced Instruction Set Computers", *Communications of the ACM*, vol. 28, No. 1, January 1985, pp. 8-21.

[RAD82] George Radin, "The 801 Minicomputer", *IBM Journal of Rsch. and Dev.*, vol. 27, no. 3, May 1982, pp. 39-47.

[THO70] J. E. Thornton, *Design of a Computer - The Control Data 6600*, Scott, Foresman and Co., Glenview, IL., 1970.

# EXECUTION OF PARALLEL LOOPS ON PARALLEL PROCESSOR SYSTEMS

*Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua*

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

**ABSTRACT** -- *Ways of coordinating large numbers of processors to execute a parallel program as fast as possible are of key importance to the design and efficient use of parallel processor systems. In this paper we discuss issues on program parallelism and scheduling for parallel processor systems. We break the scheduling process into three basic activities, and focus on processor assignment to parallel loops. Optimal processor assignment algorithms are presented for simple and complex nested parallel loops. These algorithms can be applied at compile-time, or can be implemented as hardware modules to solve the processor assignment problem optimally at run-time. Speedup measurements for EISPACK and IEEE DSP subroutines that result from the optimal assignment of processors to parallel loops are also presented. These measurements indicate that optimal assignments result in almost linear speedups on parallel processor machines with a few tens of processors, and significantly high speedups for machines with hundreds or thousands of processors.*

## 1. INTRODUCTION

It is becoming increasingly clear that modern and future supercomputers will be based on the shared memory parallel processor architecture. The flexibility, scalability and high potential performance offered by parallel processor machines are simply necessary "ingredients" for any high performance system. The performance potential for these machines is indeed greater than that of single array processor computers [7], and they can run more efficiently a larger family of programs. However we have little experience in using efficiently a large number of processors. This is especially true when maximum (single) program speedup (as opposed to high throughput) is our objective. This inexperience in turn is reflected in the small number of processors used in modern commercially available multiprocessors such as the CRAY-XMP and Alliant FX/8 systems.

Truly parallel languages, parallel algorithms, and ways of defining and exploiting program parallelism are still in their infancy. Only recently these problems have attracted the appropriate attention. Several factors should be considered when designing high performance supercomputers [7]. Parallel algorithms, carefully designed parallel architectures and powerful programming environments including sophisticated restructuring compilers, all play equally important roles on program performance. This unified approach has been adopted in the design of the Cedar multiprocessor at the Center for Supercomputing Research and Development of the University of Illinois. In addition several crucial problems such as scheduling and synchronization and communication overheads, must be adequately solved in order to take full advantage of the inherent flexibility of parallel processor systems.

Loops are the largest source of parallelism but the problem of using several processors for the fast execution of complex parallel loops had not been given enough attention until recently [10], [1]. A key problem in designing and using large parallel processor systems is finding out how to schedule independent processors to execute a parallel program as fast as possible. There is more to be said on ways of exploiting loop parallelism, both from the system and the compiler point of view. From the system's point of view we know little about coordinating large numbers of processors to execute multiply nested parallel loops; and from the the compiler's point of view no significant work has been done so far to adequately solve this problem. This paper addresses the second part of this problem namely, compile-time scheduling of arbitrarily complex parallel loops on parallel processors. We discuss the different activities of the problem traditionally referred to as scheduling of a parallel program on a parallel processor system, and present algorithms that generate optimal loop schedules at compile-time. Some experiments we conducted to illustrate the speedups resulting from such schedules are also presented.

The rest of this paper is organized as follows. In Section 2 we present some background for the following discussion, including basic definitions and the structure of the Parafrase compiler used for our experiments. In Section 3 we briefly consider the general problem of scheduling parallel programs on parallel processor systems. In Section 4 we discuss processor assignment issues for parallel loops, and propose an optimal processor assignment algorithm based on dynamic programming. In Section 5 we present some experimental results for EISPACK [13], and IEEE DSP subroutines [2], and finally Section 6 gives the conclusion of this paper.

## 2. BACKGROUND AND BASIC CONCEPTS

In this paper we consider parallel Fortran programs. By parallel, we mean programs that have been written using language extensions or programs that have been restructured by an optimizing compiler. For our purposes we use output generated by the Parafrase restructurer [6], [15]. Parafrase is

This work was supported in part by the **National Science Foundation** under Grant No. US NSF DCR84-10110, the **U. S. Department of Energy** under Grant No. US DOE DE FG02-85ER25001, and the **IBM** Donation.

0190-3918/86/0000/0519 $01.00 © 1986 IEEE

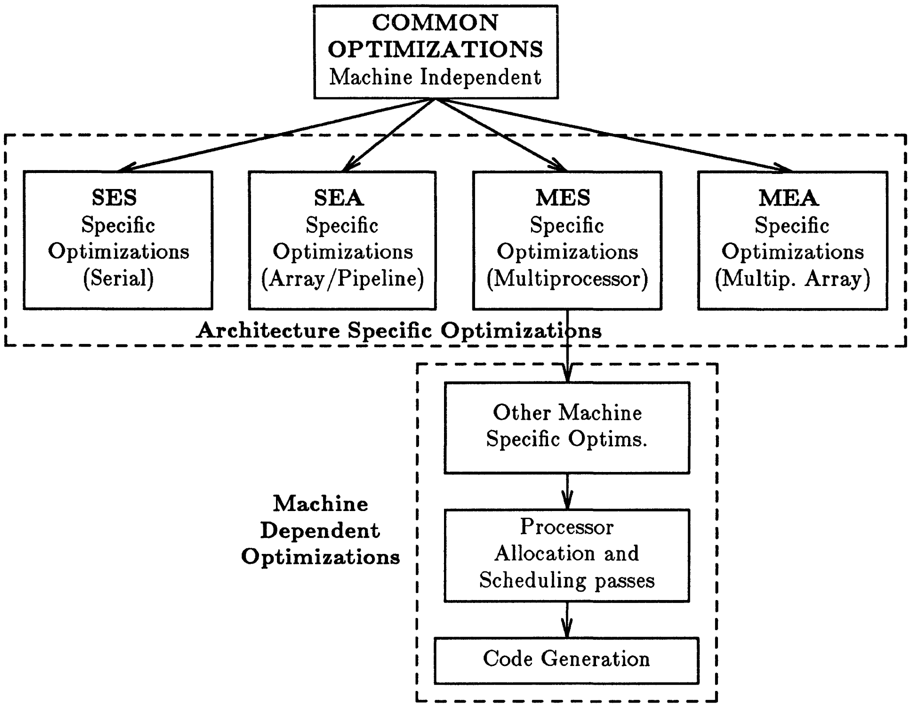

COMMON OPTIMIZATIONS
Machine Independent
SES
Specific Optimizations (Serial)
SEA
Specific Optimizations (Array/Pipeline)
MES
Specific Optimizations (Multiprocessor)
MEA
Specific Optimizations (Multip. Array)
Architecture Specific Optimizations
Other Machine Specific Optims.
Machine Dependent Optimizations
Processor Allocation and Scheduling passes
Code Generation

**Figure 1:** The structure of PARAFRASE

a restructuring compiler which receives as input Fortran programs and applies to them a series of machine independent and machine dependent transformations. The structure of Parafrase appears in Figure 1. The first part of the compiler consists of a set of machine independent transformations (passes). The second part consists of a series of machine dependent optimizations that can be applied on a given program. Depending on the architecture of the machine we intend to use, we choose the appropriate set of passes to perform transformations targeted to the underlying architecture. Currently Parafrase can be used to transform programs for execution on four types of machines: *Single Execution Scalar* or SES (uniprocessor), *Single Execution Array* or SEA (array/pipeline), *Multiple Execution Scalar* or MES (multiprocessor), and *Multiple Execution Array* or MEA (multiprocessor with vector processors) architectures [7].

In a restructured Fortran program we observe several types of parallelism and all of them can be potentially utilized by an MES machine. We can roughly classify the different types of parallelism into two categories: *Fine grain parallelism* and *Coarse grain parallelism.* Fine grain parallelism includes the parallel execution of different statements of the program on different processors, or even different operations of the same statement on different processors or functional units. Coarse grain parallelism arises from the parallel execution of independent disjoint modules of the program, or from parallel loops. For the purposes of this paper we ignore branching statements (IFs and GOTOs) without loss of generality. In Parafrase the branches of each such statement are assigned *branching probabilities* either by the user or by the compiler automatically. We can therefore view a program as consisting of a sequence of block of assignment statements (BAS), with each such BAS having a weight associated with it.

To expose program parallelism, Parafrase builds for each program the data dependence graph [5] which indicates data and control dependences between statements of the program [4]. A series of machine in/dependent optimizations are then applied on the program using the data dependence graph as a guide. During parallel execution of a program, data and control dependences must be observed in order to preserve the semantics of the program. One of the most vital Passes in Parafrase involves the recognition of several types of parallel loops.

In the transformed programs, loops can be of one of the following four types: *Serial, reductions* (e.g., vector sum/product), *do all* or DOALL (all iterations of the loop can execute in parallel) [9], and *do across* or DOACR (successive iterations can be partially overlapped) [9], [1]. The restructured programs are to be executed on a multiprocessor such that DOALLs and DOACRs can be distributed across many processors. Furthermore, execution of many loops of different kinds may also be overlapped. Given a Fortran program that executes on a $P$ processor multiprocessor system, we define program speedup $S_P$, as follows: $S_P = T_1 / T_P$, where $T_1$ is the serial execution time of the program and $T_P$ the execution time when it runs on $P$ processors. The efficiency of the execution of a parallel program that achieves a speedup of $S_P$ on $P$ processors, is then defined as $E_P = S_P / P$, and obviously $0 \leq E_P \leq 1$.

## 3. PROCESSOR ALLOCATION, ASSIGNMENT, AND SCHEDULING

Our approach to program scheduling is a combination of compile and run-time schemes. At compile-time we decide how to partition a given program into independent tasks. A task graph called a *Task Flow Graph* (TFG) is then constructed by the compiler. Each node of the TFG corresponds to a program segment, and arcs represent data and control dependences. A task node for instance, may correspond to a block of assignment statements, or a nested parallel do loop. Nodes of the TFG are executed in order so that data and control dependences are satisfied. Each node of the task graph may execute serially or it may be distributed across different processors. At compile-time it is decided whether the iterations of a given loop will execute in a *pre-scheduled* or a *self-scheduled* fashion as explained below.

The scheduling of a restructured program (TFG) on a parallel processor system is performed in three phases. As mentioned above the first phase decides whether a loop will run in a pre- or self-scheduled fashion. During the second phase which we call *processor allocation,* we decide how to distribute the available processors to the nodes of the graph. Each node receives a number of processors by the second scheduling phase, and it is up to the *processor assignment* (third) phase to decide how to use the allocated processors for each particular node. Recall that a single node of the task flow graph can be an arbitrarily complex nested parallel loop. Processor assignment decides how to assign the allocated processors to the parallel loops of each node (if any).

Allocation is usually done by the run-time system by taking a subset of the available processors (real or virtual), and devoting them to a running program or to a part of it. Allocation requires decision on the number of processors. The decision on the number of processors could be totally made by the run-time system. However, this is inefficient. To circumvent this problem, the user program could advise the system on the number of processors (or range of them) that it can efficiently use. Processor allocation may be performed at load-time, and remain constant until execution completes. On the other hand, processors may be allocated at different times during execution. The decision on how often allocation is to be performed depends on the amount of overhead involved in processor allocation, since frequent allocations is clearly better if overhead is ignored.

After allocation, a decision has to be made on the number of processors to assign to the different program components. For the purposes of this paper we are interested in processor assignment to do loops. The assignment is trivial if the process consists of a sequence of singly nested parallel loops and sequential code. The assignment may be more complex if multiply nested parallel loops are present. The algorithms of this paper deal precisely with this problem. If the loop bounds and the number of processors to be allocated at run-time, are not known at compile-time, then the algorithms described below will have to be executed at run-time once these values are known. A hardware device will accelerate the computation required for assignment.

Loops may be executed in a pre-scheduled or self-scheduled fashion. This decision can be made when the program is written or at compile-time. Pre-scheduled means that the order and the iterations to be executed on each assigned processor is decided at compile-time. The output from the compiler can be parametrized on the number of processors assigned to the loop by producing a blocked loop with a variable blocking factor. Loops can also be executed in a self-scheduled way. This means that each processor upon completion of an iteration will continue by executing the next iteration not yet executed or being executed. If $p$ processors are assigned, the first $p$ iterations are executed first, one by each processor. Pre-scheduling has the advantage that the local memory of a processor (including its registers) can be used to transfer information from one iteration to subsequent ones executed in the same processor, so that in certain cases interprocessor communication cost may be greatly reduced. Consider for example two disjoint adjacent loops at the same nest level. Both loops will be allocated the same number of processors and corresponding iterations will execute on the same physical processors. When data are communicated from one loop to the other they can be temporarily stored into a local memory, thus eliminating the need for interprocessor communication. On the other hand, self-scheduling is useful if the execution time of the different iterations varies widely. These factors should be taken into consideration by the compiler to decide how to compile a given loop. Also, the presence or not of hardware for self-scheduling should be taken into account. It should be noted that pre-scheduling and self-scheduling can be combined in a program, and even in a multiply nested loop. Thus, the outer loop could be scheduled in one way, and the inner loop in another.

## 4. OPTIMAL PROCESSOR ASSIGNMENTS

It has been shown that in most programs parallel loops are the source of the greatest percentage of parallelism [7]. In this section we will investigate the problem of processor assignment to parallel loops. This problem becomes especially important when we deal with nested parallel loops where inefficient assignment algorithms may result in an execution time far worse than the optimal. In programs with several nested parallel loops the efficiency may then drop down to unacceptable levels. We can informally define the limited processor assignment problem as follows: Given an arbitrary multiply nested loop which contains serial and parallel (DOACR, DOALL) loops and a number of $P$ processors, find the (optimal) way of assigning the $P$ processors to the loops so that the parallel execution time of the entire module is minimized. For loops with very few nest levels and systems with a small number of processors, exhaustive search might be affordable at compile-time. But as the number of processors increases, the number of processor-loop combinations grows exponentially. Moreover, loops with large nest levels are not very uncommon in scientific computations. As an example, 10 to 17 nested parallel loops were observed in several subroutines of the restructured (by Parafrase) IEEE Digital Signal Processing Package.

### 4.1. Simple Assignments to DOALLs

In the following sections we discuss processor assignments schemes. A metric called the *efficiency index* is defined in this section. The usefulness of this metric is twofold. First it makes it easier to formulate the processor assignment problem, and secondly it allows us to observe several interesting properties of the problem that are otherwise hidden in modular arithmetic.

A processor assignment algorithm (OPTAL) is proposed that solves the general problem optimally. The optimal processor assignment is guided by the use of a function called the *assignment function.* The assignment function can be easily defined to measure parallel execution time.

Before we discuss processor assignment issues we need to introduce some notation and definitions. To simplify the notation each loop is assumed to be normalized and denoted by the upper bound of its iteration space. Thus, $N_i$ denotes a DOALL whose loop body is executed $N_i$ times, and $L = (N_1, N_2, \ldots, N_m)$ denotes an $m$ level nested DOALL where loop $N_i$ is surrounded by $N_{i-1}$ and surrounds $N_{i+1}$, $i = 2, \ldots, m-1$ ($N_1$ and $N_m$ are the outermost and innermost loops respectively).

In what follows the number of available processors $P$ is always assumed to be "useful", that is, less or equal to the maximum number of processors that a loop $L$ can fully utilize. As mentioned in Section 3, processor allocation (phase two) allocated a number of processors to each outermost loop (task node). Theorems and lemmas given in this paper are stated without proof [11].

**Definition 4.1.** For a DOALL with $N_i$ iterations that has been assigned $p_i$ processors we define $\epsilon_i$, the *efficiency index* or *EI* of $N_i$ as follows:

$$\epsilon_i^{p_i} = \frac{N_i / p_i}{\lceil N_i / p_i \rceil} \tag{1}$$

The efficiency index is an indicator of how efficiently a loop runs on a given number of processors. The higher the EI the higher the efficiency (as defined in Section 2). Some other properties of the efficiency index that will be used directly or indirectly in the following sections are:

**P1:** For any DOALL $N_i$ and any number of processors $p$ we have: $0 < \epsilon_i^p \leq 1$.

**P2:** For any $N_i$, $\epsilon_i^1 = 1$.

**P3:** For $N_i \geq p$, $\epsilon_i^p > 1/2$.

It should also be noted that $p \neq q$ does not necessarily imply $\epsilon_i^p \neq \epsilon_i^q$.

**Definition 4.2:** For a nested DOALL $L = (N_1, N_2, \ldots, N_m)$, a number of $P = p_1 p_2 \ldots p_m$ processors and a particular assignment of $P$ to $L$ we define the *efficiency index vector* $\omega = (\epsilon_1^{p_1}, \epsilon_2^{p_2}, \ldots, \epsilon_m^{p_m})$, of $L$, where $\epsilon_i^{p_i}$ is the EI for loop $N_i$ and for $p_i$ processors.

In what follows the terms "assignment of $P$" and "decomposition of $P$" are used interchangeably. Any assignment of $P$ to $L$ defines implicitly a decomposition of $P$ into factors $P = p_1 p_2 \ldots p_m$ where each of the $m$ different loops receives $p_i$, $i = 1, 2, \ldots, m$ processors. A processor assignment profile $(p_1, p_2, \ldots, p_m)$ can also be described by its efficiency index vector as defined above.

**Definition 4.3:** For an assignment $\omega = (\epsilon_1^{p_1}, \ldots, \epsilon_m^{p_m})$, of $P = p_1 p_2 \ldots p_m$ processors to $L$, we define $\mathrm{E}_L$, the *compound efficiency index* ( *CEI* ) of $L$ as

$$\mathrm{E}_L = \prod_{i=1}^{m} \epsilon_i^{p_i} \tag{2}$$

For any $L$, $P$, we also have $0 < \mathrm{E}_L \leq 1$. Let $T_1$ be the serial execution time of a perfectly nested DOALL $L$. Next suppose that $L$ is executed on $P$ processors and let $T_P$ and $T_P'$ denote the parallel execution times for two different assignments $\omega = (\epsilon_1, \epsilon_2, \ldots, \epsilon_m)$ and $\omega' = (\epsilon_1', \epsilon_2', \ldots, \epsilon_m')$ of $P$ to $L$, where $P = p_1 \ldots p_m = p_1' \ldots p_m'$. We can express the parallel execution time $T_P$ of $L$ in terms of its CEI as follows:

$$T_P = \prod_{i=1}^{m} \lceil N_i / p_i \rceil B \rightarrow$$

$$\frac{1}{T_P} = (1/B) \frac{\prod_{i=1}^{m} N_i / p_i}{\prod_{i=1}^{m} \lceil N_i / p_i \rceil} \prod_{i=1}^{m} p_i / N_i =$$

$$(1/B) \left( \prod_{i=1}^{m} \frac{N_i / p_i}{\lceil N_i / p_i \rceil} \right) \frac{\prod_{i=1}^{m} p_i}{\prod_{i=1}^{m} N_i} = \prod_{i=1}^{m} \epsilon_i^{p_i} \frac{P}{NB} = \frac{\mathrm{E}_L P}{NB} \quad \text{or}$$

$$T_P = \frac{NB}{\mathrm{E}_L P} \tag{3}$$

where $B$ is the execution time of one iteration of the loop, and $N = \prod_{i=1}^{m} N_i$. The following is a direct application of (3).

**Lemma 4.1:** $T_P < T_P'$ iff $\mathrm{E}_L > \mathrm{E}_L'$.

In the next few sections we show how the efficiency index can be used to direct the efficient assignment of processors to perfectly nested parallel loops.

Given a nested loop $L$ and a number of processors $P$ we call a *simple* processor assignment one that assigns all $P$ processors to a single loop $N_i$ of $L$. A *complex* processor assignment on the other hand is one that assigns two or more factors of $P$ to two or more loops of $L$.

**Theorem 4.2.** The optimal simple processor assignment over all simple assignments of $P$ processors to loop $L$ is achieved by assigning $P$ to the loop with $\epsilon = \max_{i=1,m} \{\epsilon_i^P\}$.

For the next lemma and most of what follows we assume that processors are assigned in units that are equal to products of the prime factors of $P$ unless stated otherwise. Therefore each loop is assigned a divisor of $P$ including one.

**Lemma 4.3.** If $N$ is a (single) DOALL loop, $P = p_1 p_2 \ldots p_m$ is the number of available processors and $\epsilon_i$, $i = 1, 2, \ldots, m$ are the efficiency indexes for assigning $p_1, p_1 p_2, p_1 p_2 p_3, \ldots, p_1 p_2 \ldots p_m$ respectively, then

$$\epsilon_1 \geq \epsilon_2 \geq \epsilon_3 \geq \ldots \geq \epsilon_m.$$

From Lemma 4.1 we conclude that the optimal processor assignment of $P$ to $L$ is the one that maximizes $\mathrm{E}_L$. Each assignment defines indirectly a decomposition of $P$ into a number of factors less than or equal to the number of loops in $L$. As $P$ grows the number of different decompositions of $P$ into factors grows very rapidly. From number theory we know that each integer is uniquely represented as a product of prime factors. Theorem 4.4 below can be used to prune (eliminate from consideration) several decompositions of $P$, or equivalently, several assignment profiles of $P$ to $L$ that are not close to optimal. From several hand generated tests we observed that the use of Theorem 4.4 in a branch and bound algorithm for determining the optimal assignment of processors eliminated more than ninenty percent of all possible assignments. In some instances all but the optimal assignments were pruned by the test of Theorem 4.4.

Again let $L = (N_1, \ldots, N_m)$ be a perfectly nested DOALL that executes on $P$ processors and $P = p_1 p_2 \cdots p_k$ be any decomposition of $P$ where $k \leq m$. Now let $\epsilon = \max_{1 \leq i \leq m} \{\epsilon_i^P\}$ be the maximum efficiency index over all simple assignments of $P$ to $L$, and $\epsilon_i = \max_{1 \leq j \leq m} \{\epsilon_j^{p_i}\}$, $i = 1,2,\ldots,k$ be the maximum efficiency indexes (over all loops of $L$) for the factors $p_1, p_2, \ldots, p_k$ of $P$ respectively (where $\epsilon_j^{p_i} = (N_j/p_i) / (\lceil N_j / p_i \rceil))$. Note that here we do not perform any actual assignment of processors to loops but simply compute the maximum efficiency index for each factor $p_i$ of $P$ over all loops of $L$ excluding the loop that corresponds to $\epsilon$. If $T_s$ and $T_c$ are the parallel execution times for $L$ corresponding to the optimal simple assignment of $P$ and the optimal complex assignment of the specific factors of $P$ respectively, and $S_s$ and $S_c$ their respective speedups, we have the following theorem.

**Theorem 4.4.** If there exists $i \in \{1,2,\ldots,k\}$ for which $\epsilon \geq \epsilon_i$ then $T_s \leq T_c$ and thus $S_s \geq S_c$. In other words if one of the factors of $P$ has a maximum efficiency index equal or less than the maximum efficiency index of $P$, then we gain more speedup by assigning the entire $P$ to a single loop than from any complex assignment of the factors of $P$ (including the optimal).

Thus, given any decomposition of $P$ into factors $P = p_1 \ldots p_k$, a necessary (but not sufficient) condition for a complex assignment to be better than the best simple assignment is $\epsilon < \epsilon_i$, for all $i = 1,2,\ldots,k$ (where $\epsilon_i$ is the maximum efficiency index for factor $p_i$ over all loops of $L$). Obviously if $\epsilon = 1$ then the optimal simple assignment is the overall optimal as well. An example of the application of Theorem 4.4 is shown in Figure 2(a), where the simple assignment of 32 processors to the outermost loop is also the optimal one.

**Corollary 4.5.** If $N = \prod_{i=1}^{m} N_i$, $\epsilon$ is the efficiency index of the optimal simple assignment, $\epsilon_i$, $i = 1,2,\ldots,m$ is the efficiency index for the $i$-th loop in a complex optimal assignment and $\mathrm{E}_L$ the corresponding compound efficiency index, then any optimal complex assignment should satisfy,

$$\epsilon < \epsilon_i \leq 1, \quad i = 1, 2, \ldots, m \quad \text{and} \quad \epsilon < \mathrm{E}_L \leq 1.$$

$$\text{Let} \quad \mathrm{E}_0 = \frac{N / P}{\lceil N/P \rceil} \quad \text{where} \quad N = \prod_{i=1}^{m} N_i.$$

Then any optimal assignment of $P$ to $L$ satisfies

$$\mathrm{E}_L \leq \mathrm{E}_0 \tag{4}$$

where $\mathrm{E}_L$ is the CEI of an optimal assignment. Only in special cases there would be an optimal assignment of $P$ to $L$ for which the equality in (4) holds. A compiler transformation called "loop-coalescing" can be applied to certain types of loops and always achieves $\mathrm{E}_L = \mathrm{E}_0$ [11].

Corollary 4.5 can be used to check whether a given complex assignment is better than an optimal simple assignment. It would be useful however to be able to answer the question about the existence of such assignment. That is, given a loop $L$ and a number of $P$ processors, is there an optimal complex assignment better than the optimal simple assignment? If for a particular loop the answer is negative, the optimal simple assignment is chosen and therefore the problem for that loop is solved in constant time, assuming the efficiency indexes have been computed. Proposition 4.6 below provides the test for the existence of an optimal complex assignment.

For each loop $N_i \in L$ we define the *critical capacity* $g_i$, of $N_i$ as the maximum number of processors that can be assigned to $N_i$ with its efficiency index remaining strictly greater than $\epsilon$ (the maximum efficiency index of $P$). In other words, for each $N_i$ $g_i$ is chosen to satisfy,

$$\epsilon_i^{g_i} > \epsilon \quad \text{and} \quad \epsilon_i^{g_i + r} \leq \epsilon$$

for any $r \geq 1$. Then we have the following proposition.

**Proposition 4.6.** A necessary condition for the existence of a complex assignment of $P$ to $L$ which is better than the corresponding optimal simple assignment is

$$\prod_{i=1}^{m} g_i \geq P.$$

The obvious approach for optimally solving the processor assignment problem is exhaustive search. For small nested loops and a very small number of processors exhaustive search would probably be tolerable at compile-time. For medium size loops and a few tens of processors however, the cost of exhaustive search becomes intolerable even at compile-time. For example, the number of different assignments of 50 processors to 15 nested loops is $4.8X10^{13}$. If it takes 1000ns (on a fast machine) to process each different assignment it would take more than 555 days CPU time to find the optimal assignment of 50 processors to 15 loops.

Using the results of this section however, we can design a branch and bound algorithm that greatly reduces the number of candidate optimal assignments. In several cases the tests of this section can prune all possible assignments but the optimal and in practice such a branch and bound algorithm would have polynomial complexity for most cases. The problem remains unsolved though since we can never guarantee polynomial complexity and we can always come up with an example loop which can make even the branch and bound algorithm run in exponential time.

In the next section we present an optimal processor assignment algorithm that has a low polynomial complexity and finds the optimal assignment for all types of loops and any number of processors.

### 4.2. Optimal Complex Assignments

In order to better illustrate the ideas of this section we start by considering perfectly nested DOALLs and $P = 2^k$ processors. As we proceed the concepts are generalized to include more complex loop structures such as nonperfectly nested combinations of serial, DOALL, and DOACR loops.

Let us consider an $m$-level nested DOALL $L = (N_1, N_2, \ldots, N_m)$ and a number of $P = 2^k$ processors.

```
DOALL 1 I1=1,63                       DOALL 1 I1=1,15
  DOALL 2 I2=1,7                        DOALL 2 I2=1,17
    DOALL 3 I3=1,31                       DOALL 3 I3=1,17
      DOALL 4 I4=1,20                       DOALL 4 I4=1,25
          . . .                                 . . .
(a)       . . .                     (b)         . . .
          . . .                                 . . .
4       CONTINUE                      4       CONTINUE
3     CONTINUE                        3     CONTINUE
2   CONTINUE                          2   CONTINUE
1 CONTINUE                            1 CONTINUE
```

Figure 2. (a) An application of Theorem 2.2.
(b) The nested loop of example 4.1.1.

Algorithm OPTAL which is analytically described below will give us the optimal assignment profile of the $P$ processors to the $m$ loops of $L$. For each loop $L$ we compute the *efficiency table* $M$. Each column $j$ of $M$ corresponds to a loop $N_j$ of $L$ and each row $i$ corresponds to a number of $2^i$, $i=0,1,...,k$ processors. An entry $(i,j)$ of table $M$ contains the efficiency index for assigning $2^i$ processors to loop $N_j$. This $(m \times k)$ efficiency table will be used repeatedly by OPTAL to obtain the optimal assignment of $P$ processors to loop $L$.

From Lemma 4.3 we observe that each column of $M$ is ordered in nonincreasing order. If the loops are ordered by size then each row of $M$ is also ordered in nonincreasing order. Therefore if $\epsilon_{ij}$ is the element of $M$ in the $i$-th row and $j$-th column, then

$$\epsilon_{ij} \geq \epsilon_{iw} \quad \text{for} \quad \text{w} \geq j.$$

It is clear that in any assignment of $P$ to $L$ there can be at most one entry of the lower half of $M$ involved in that assignment. Let us give an outline of the basic steps of the algorithm. We start by assigning the $P$ processors to the innermost or outermost loop, and let us always start from the innermost in our case. The second step finds the optimal assignment of $P$ to the two innermost loops. In the process we also need to compute the optimal assignment of $1, 2, 2^2, \ldots, 2^k$ processors respectively to the two innermost loops. These assignments however are computed only once for each loop and stored for later use by successive steps.

In general, after the $(m - i)$-th step OPTAL has found the optimal assignment of $1, 2, 2^2, ..., P$ processors to loops $L_i=(N_i, N_{i+1}, ..., N_m)$. The next $(m - i + 1)$-th step considers loop $N_{i-1}$ and finds the optimal assignment of $1, 2, 2^2, \ldots, P$ processors to loop $(N_{i-1}, L_i)$ possibly by reassigning processors from $L_i$ to $N_{i-1}$. All possible assignments for $N_{i-1}$ are considered. Note that all possible assignments for $L_i$ have already been computed. At the end of the $m$-th step OPTAL outputs the profile of the optimal assignment of $P = 2^k$ to loop $L=(N_1, N_2, ..., N_m)$. Based on Lemma 4.1 the optimal assignment of $P$ to $L$ would be the one that maximizes $\text{E}_L$. This is precisely what OPTAL does.

### 4.2.1. The Perfectly-Nested Loop Case

In this section we describe processor assignment for perfectly nested DOALLs and $P = 2^k$. We use this case as an example of the application of the general algorithm described in the next section. The simplicity of this case helps illustrate the algorithm clearly. It is followed by a simple example that describes the details of computing the optimal assignment. The heart of OPTAL is a recursive function $G$, that is defined as follows: Given $P=2^k$ and $L$ an $m$-way nested loop as previously, we define $G_i(q)$ as the product of efficiency indexes of the optimal assignment of $q$ processors to loops $(N_i, N_{i+1}, ..., N_m)$. More specifically a closed form expression of function $G_i(q)$ is given by,

$$G_i(q) = \max_{1 < p_j < q} \left\{ \prod_{j=i}^{m} \epsilon_j^{p_j} \right\}$$

and such that $q = \prod_{j=i}^{m} p_j \leq P$. The recursive definition of $G_i(q)$ that we will be using from now on is given by (5)

$$G_i(P) = \max_{0 \leq r \leq k} \left\{ \epsilon_i^{2^r} G_{i+1}(P/2^r) \right\} \quad \text{or} \qquad (5)$$

$$G_i(P) = \max \left\{ G_{i+1}(P), \epsilon_i^2 G_{i+1}(P/2), \epsilon_i^4 G_{i+1}(P/4), \right.$$

$$\left. \epsilon_i^8 G_{i+1}(P/8), \ldots, \epsilon_i^P G_{i+1}(1) \right\}$$

where $\epsilon_i^q$ is the efficiency index for assigning $q$ processors to loop $N_i$ (available from table $M$). The optimal assignment of $P$ processors to loops $(N_i, N_{i+1}, ..., N_m)$ can be found by selecting from all assignments of $2^r$ processors to loop $N_i$ and $2^{k-r}$ processors to $(N_{i+1}, ..., N_m)$, $r = 0, 1,..., k$, the one that maximizes $\prod_{j=i}^{m} \epsilon_j$ (from (5)).

The function in (5) is computed for $i=m, m\text{-}1, ..., 1$ and for each $i$ we also compute $G_i(1)$, $G_i(2)$, $G_i(2^2)$,..., $G_i(P=2^k)$. The optimal assignment of the $P$ processors to loop $L$ will be given at the end of the $m$-th step by $G_1(2^k)$. Initially (first step) for $i=m$ we have $G_m(q) = \epsilon_m^q$. For each $G_i(q)$ the corresponding processor assignment profile is stored and when $G_1(2^k)$ is computed the profile for the optimal assignment is available.

The algorithm completes in $m$ steps. In each of the $m$ steps, $k=logP$ function evaluations are performed and each of the $r=1,2,...,k$ function evaluations involves the computation of the maximum of $r$ values. The overall complexity of the algorithm is therefore $O(mlog^2P)$. Using the results of the previous sections we can easily avoid unnecessary computations and further reduce the complexity of OPTAL.

The explicit processor assignment vector (with the exact number of processors assigned to each loop) is computed as a side effect of the computation of $G_i$. When a particular $G_i$ is chosen as optimal the corresponding assignment vector can be trivially reconstructed. In order to illustrate the computational details of the algorithm we give below a simple example involving four DOALLs and $2^5$ processors. It should be noted that this approach not only finds the optimal assignment of the given $P$ processors to a particular loop nest, but it also finds the optimal assignments of $P/2$, $P/4$, $P/8$, $P/16$,..., 1 processors to the same loop. We can therefore determine the minimum number of useful processors with little extra cost.

**Example 4.1.1:** Consider the loop $L=(N_1 = 15,\ N_2 = 17,\ N_3 = 17,\ N_4=25)$ of Figure 2(b) and let $P=2^5$ be the available processors. The optimal assignment of $P$ to $L$ is computed as follows: First the 5X4 efficiency matrix $M$ is computed. At the first step for $i=4$ we have $G_4(2^r) = \epsilon_4^{2^r}$ for $r = 0, 1, \ldots, 5$. The computations for the remaining three steps are shown analytically below.

**Step 2**

$$G_3(2)=\max\left\{G_4(2),\epsilon_3^2G_4(1)\right\}$$

$$G_3(4)=\max\left\{G_4(4),\epsilon_3^2G_4(2),\ \epsilon_3^4G_4(1)\right\}$$

$$G_3(8)=\max\left\{G_4(8),\epsilon_3^2G_4(4),\ \epsilon_3^4G_4(2),\epsilon_3^8G_4(1)\right\}$$

$$G_3(16)=\max\left\{G_4(16),\epsilon_3^2G_4(8),\epsilon_3^4G_4(4),\epsilon_3^8G_4(2),\epsilon_3^{16}G_4(1)\right\}$$

$$G_3(32)=\max\left\{G_4(32),\ \epsilon_3^2G_4(16),\epsilon_3^4G_4(8),\epsilon_3^8G_4(4),\epsilon_3^{16}G_4(2),\epsilon_3^{32}G_4(1)\right\}$$

**Step 3**

$$G_2(2)=\max\left\{G_3(2),\ \epsilon_2^2G_3(1)\right\}$$

$$G_2(4)=\max\left\{G_3(4),\ \epsilon_2^2G_3(2),\epsilon_2^4G_3(1)\right\}$$

$$G_2(8)=\max\left\{G_3(8),\epsilon_2^2G_3(4),\ \epsilon_2^4G_3(2),\epsilon_2^8G_3(1)\right\}$$

$$G_2(16)=\max\left\{G_3(16),\epsilon_2^2G_3(8),\ \epsilon_2^4G_3(4),\epsilon_2^8G_3(2),\epsilon_2^{16}G_3(1)\right\}$$

$$G_2(32)=\max\left\{G_3(32),\ \epsilon_2^2G_3(16),\epsilon_2^4G_3(8),\epsilon_2^8G_3(4),\epsilon_2^{16}G_3(2),\epsilon_2^{32}G_3(1)\right\}$$

**Step 4**

$$G_1(2)=\max\left\{G_2(2),\ \epsilon_1^2G_2(1)\right\}$$

$$G_1(4)=\max\left\{G_2(4),\epsilon_1^2G_2(2),\epsilon_1^4G_2(1)\right\}$$

$$G_1(8)=\max\left\{G_2(8),\epsilon_1^2G_2(4),\epsilon_1^4G_2(2),\epsilon_1^8G_2(1)\right\}$$

$$G_1(16)=\max\left\{G_2(16),\epsilon_1^2G_2(8),\epsilon_1^4G_2(4),\epsilon_1^8G_2(2),\epsilon_1^{16}G_2(1)\right\}$$

$$G_1(32)=\max\left\{G_2(32),\epsilon_1^2G_2(16),\epsilon_1^4G_2(8),\epsilon_1^8G_2(4),\epsilon_1^{16}G_2(2),\ \epsilon_1^{32}G_2(1)\right\}$$

In each case the maximum element appears in bold letters. The optimal assignment in this example is therefore the one that assigns 16 processors to loop $N_1 = 15$ and 2 processors to loop $N_4 = 25$. The processor assignment profile is reconstructed as follows. The number of processors assigned to a loop $l$, is equal to the superscript of the $\epsilon$ factor of the maximum term in $G_l(P)$. If an $\epsilon$ factor does not exist, the corresponding loop receives 1 processor. First we look at the maximum element of $G_1(32)$. This element is $\epsilon^{16}G_2(2)$ which tells us that loop $N_1$ receives 16 processors, and the remaining processors are allocated to $G_2(2)$. The maximum element of entry $G_2(2)$ is $G_3(2)$ which indicates that loop $N_2$ receives 1 processor. Continuing in the same way, the maximum element of entry $G_3(2)$ is $G_4(2)$ which again tells us that loop $N_3$ is assigned 1 processor, and therefore loop $N_4$ is assigned the remaining 2 processors.

### 4.2.2. The General Algorithm

Although most real multiprocessor systems have $P = 2^k$ for some integer $k$, OPTAL can be used to generate optimal processor assignments for any integer $P$. It also handles arbitrarily complex nested parallel loops. Before we describe the details of the general algorithm however, we need to define the concepts of DOACR and loop nesting more precisely. The details about the DOACR [9], [1] construct are beyond the scope of this paper and we only define terms used in later discussion.

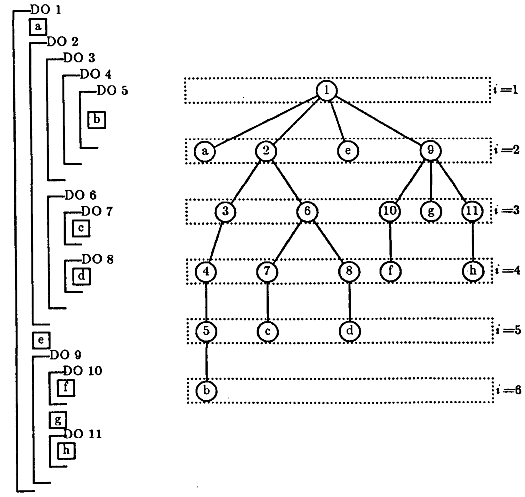


Figure 3. A nested loop and its tree representation. Squares and leaves denote BASs.

As mentioned in Section 2, a DOACR loop can be informally defined as a parallel loop in which data dependences allow for partial overlap of successive iterations during execution on an MES system. In other words, if iteration $i$ starts at time $t$ on a given processor, iteration $(i+1)$ can start at time $t + d$, where $d$ is a constant. Constant $d$ is called delay and represents the execution time of a subset of loop statements whose data dependence graph forms a cycle. If $B$ is the size (serial execution time) of the loop body, then $d/B$ is defined as the percentage of overlap, (or doacross percentage). When $d=B$ the loop is serial while if $d=0$ the loop is said to be DOALL. In DOALL loops all iterations may start execution simultaneously. DOALL and serial loops are therefore special cases of DOACR loops. The parallel execution time of a DOACR loop with $N_i$ iterations, $d_i$ delay and a body size of $B_i$ that executes on $P$ processors is given by the following [12].

$$T_P^i(B_i) = \left(\left\lceil\frac{N_i}{P}\right\rceil - 1\right) * \max\{B_i,\ Pd_i\} + d_i * \Big((N - 1) \bmod P\Big) + B_i \qquad (6)$$

To simplify the notation in the following discussion, we assume that a block of assignment statements (BAS) can be considered as a DOACR loop with $N_i = 1$, and $d_i = 0$.

An arbitrarily complex nested loop can be uniquely represented as a $k$-level tree where $k$ is the maximum nest depth. The leaves of the tree correspond to BASs and intermediate nodes correspond to (DOACR) loops. Obviously the total number of nodes in a loop tree is $\lambda + \mu$ where $\lambda$ is the number of individual loops in the structure and $\mu$ the number of BASs. An example of a nested loop and its tree representation are shown in Figure 3. Intermediate tree nodes at level $m$ correspond to loops at nest depth $m$. We assume that individual loops in an arbitrarily nested loop are numbered increasingly, in lexicographic order.

In the general case loops are not perfectly nested and therefore the efficiency index as defined in Section 4 is not useful. We can redefine the efficiency index for the general case but it is more convenient to define the assignment function so that it measures directly parallel execution time. Therefore the max term of the assignment function becomes min in this case since our objective here is to minimize execution time and thus maximize speedup.

The steps of the general algorithm are almost identical to the case of perfectly nested loops. The example of Figure 3 is used whenever it helps illustrate the computations involved. A $\lambda \times P$ table can be used to store intermediate values. ($\lambda$, $P$ are the numbers of loops and processors respectively). During the first step we compute the parallel execution time of the DOACR loops at level $k$ on the tree, where $k$ again is the maximum nest level. This is done as follows:

$$G_k^i(q) \;=\; T_q^i(B_i)\,, \qquad q \;=\; 1, 2, ..., P \qquad (7)$$

and for all leaves i.

where $T_q^i$ is given by (6). The general step is defined recursively as in the perfectly nested loop case. The optimal assignment of $P$ processors to loops in levels $i$ through $k$ ($i < k$), (assuming the optimal assignment of $P$ to loops at level $i$+1 is known), is then generated by:

$$G_i^j(q) \;=\; \min_{1 \leq r \leq q} \left\{ T_r^j \left( \sum_{n\ child\ of\ j} G_{i+1}^n(\lfloor q/r \rfloor) \right) \right\} \qquad (8)$$

and for $q$=1, 2, 3, ..., $P$

where (8) is computed for all nodes (loops) $j$ at level $i$, and $T(^*)$ is given by (6). The summation in (8) accounts for all nodes at level $i + 1$ that are descendants of node $j$, that is, all loops nested inside loop $N_j$. The optimal assignment of $P$ processors to a given loop is given by $G_1^1(P)$. Recall from the example of the previous section that the detailed processor assignment vector is automatically constructed during the evaluation of 8. For each loop the number of processors assigned to that loop correspond to the minimum term in 8.

It should be noted that all optimal assignments of 1, 2,..., $P$-1 processors to a loop $L$ are computed as intermediate results of the computation of $G_1^L(P)$. We therefore have the following.

**Lemma 4.7.** The maximum number of useful processors given $P$ for a loop $L$ is the minimum $Q$, such that $1 \leq Q \leq P$ and $G_1^L(Q) = G_1^L(P)$.

**Theorem 4.8.** For any loop $L$ of maximum nest depth $k$, and any integer $P$, OPTAL terminates after $k$ iterations and generates the optimal assignment of $P$ processors to $L$.

The complexity of the algorithm can be easily determined. The assignment function $G_i^j$ is computed $P$ times for each node (loop) in the tree, or a total of $\lambda P$ times. Each evaluation of the assignment function also involves finding the minimum of an average of $P/2$ terms. The complexity therefore (without counting additions) is $O(\lambda P^2 / 2)$. The complexity can be reduced to $O(\lambda P \log P)$, and OPTAL can be used to implement a systolic array control unit that consists of $P \log P$ nodes and determines the optimal assignment of $P$ processors to a given loop in $\lambda$ steps [11]. The speedup resulting from the optimal assignment of $P$ processors to a loop $L$ is given by, $S_P \;=\; G_1^L(1) \;/\; G_1^L(P)$.

An interesting point of this approach is that although loops at the same nest level are allocated the same total number of processors, each loop manages (assigns) its own processors in an independent way. For example, suppose that loops 3 and 6 of Figure 3 are allocated 8 processors each. A possible assignment then may assign 1 processor to loops 3 and 4, and 8 processors to loop 5, while in the second case we may have 2 processors assigned to loop 6, and 4 processors to each of the loops 7 and 8. It is clear that loops on the same nest level must be assigned the same total number of processors when executing on a parallel processor system. Otherwise we have suboptimal parallel execution times since some processors will be forced to remain idle.

## 5. EXPERIMENTS

We implemented this processor assignment algorithm as a pass in the Parafrase compiler. Processor assignment is performed after DOALL and DOACR loops are recognized and delays computed. In our experiments we measured speedup values for some subroutines of the EISPACK and IEEE DSP packages, even though analysis of the entire packages is under way.

Speedup values were computed as discussed in Section 2. In our case $T_P$, the parallel execution time, was measured for $P$=32, $P$=256, and $P$=2048 processors, and for loop bounds set to 40. In some EISPACK subroutines where loop bounds correspond to the bandwidth of band-matrices, we used loop bounds of 1 or 4. The speedup values measured for the three different numbers of processors are shown in Tables 1 and 2. The subroutines from the two packages used in these experiments were randomly selected.

From the speedup values we observe that for 32 processors the average speedup is almost linear for both EISPACK and IEEE subroutines. For 256 processors the average speedup for EISPACK subroutines is about 137, or more than $P/2$. In other words, we have an efficiency of more than 50% for $P$=256. For the IEEE subroutines we observe an even higher average efficiency for the same number of processors. The third column in the tables corresponds to an unlimited number of processors. Since most of the EISPACK subroutines deal with square matrices, for 40X40 arrays the the maximum expected speedup is 1600. Taking into account several loops with bounds of 1 or 4 and the number of one-dimensional loops, the average maximum speedup should be expected to be considerably lower than 1600. The average speedup of the third column of Table 1 is about 310, which corresponds to an average efficiency of about 15%. Since at most 1600 processors would be useful for most of the EISPACK routines, in reallity we would have an efficiency of about 20%. The corresponding values for the third column of Table 2 are quite higher than those of EISPACK. Generally, supercomputers deliver a wide range of performances from program to program. This is true of real machines [16], and has been observed in our earlier experimental work [7]. It appears, from the experiments we

| Subroutine Name | $S_{32}$ | $S_{256}$ | $S_{2048}$ |
|---|---|---|---|
| ELMBAK | 31.9 | 242.0 | 668.0 |
| ELMHES | 31.7 | 33.6 | 33.6 |
| ELTRAN | 29.3 | 71.3 | 84.5 |
| HQR2 | 18.0 | 26.0 | 28.0 |
| TRED1 | 31.0 | 235.0 | 240.0 |
| MINFIT | 28.0 | 130.0 | 181.0 |
| TRED2 | 18.3 | 36.5 | 39.5 |
| CBABK2 | 30.0 | 53.5 | 57.4 |
| CH | 25.0 | 66.9 | 85.0 |
| COMBAK | 31.9 | 248.5 | 721.0 |
| CORTB | 32.0 | 254.0 | 1250.0 |
| CORTH | 32.0 | 252.0 | 501.0 |
| BANDV | 31.0 | 98.0 | 98.0 |

| Subroutine Name | $S_{32}$ | $S_{256}$ | $S_{2048}$ |
|---|---|---|---|
| INISHL | 32.0 | 255.8 | 1021.0 |
| WFTA | 32.0 | 255.8 | 2036.9 |
| TRBIZE | 30.8 | 128.6 | 128.6 |
| PCORP | 31.9 | 246.3 | 537.0 |
| POWER | 26.4 | 68.2 | 79.6 |
| COSYFP | 22.2 | 54.4 | 65.7 |
| FREDIC | 31.9 | 162.0 | 190.0 |
| FLPWL | 30.9 | 169.4 | 363.0 |
| DIINIT | 28.0 | 89.5 | 119.4 |
| SRINIT | 21.3 | 48.6 | 56.8 |
| SMINVD | 31.9 | 120.6 | 186.0 |
| DEFIN4 | 19.2 | 37.6 | 37.6 |
| FFT | 30.8 | 191.0 | 505.0 |
| LOAD | 22.0 | 36.3 | 36.3 |
| COVAR1 | 31.5 | 68.0 | 76.5 |
| CLHARM | 27.7 | 91.8 | 120.0 |
| FLCHAR | 31.0 | 188.3 | 458.8 |
| REMEZ | 10.0 | 12.1 | 12.3 |
| D | 31.3 | 210.0 | 670.0 |
| LPTRN | 11.0 | 13.9 | 14.8 |

Table 1: Speedup values for 32, 256, and 2048 processors for EISPACK (Table 1), and IEEE DSP subroutines (Table 2).

have conducted so far, that when OPTAL is used there is very little variation when programs are run with limited number of processors (i.e, when the number of processors is proportional to array sizes).

Considering the fact that efficiencies in the range of 20% are characterized very satisfactory in modern supercomputers, we can claim that optimal assignments to parallel loops result in high speedups for most cases. Processor allocations to independent code segments can increase the average speedup at least by a factor of two [14], [11].

## 6. CONCLUSION

The problem of coordinating several parallel processors to execute a parallel program as fast as possible is a key problem to the efficient use of large parallel processor machines, and the design of multiprocessors with hundreds or even thousands of processors. It is very important to be able to identify parallelism in a program and use all the available processors to maximize speedup. This problem however is complex and should not be shifted entirely to the user. Compilers, operating systems, and special hardware modules can be used to solve the problem satisfactorily with little or no user assistance.

In this paper we discussed issues of program parallelism and scheduling for parallel processor systems. We divided the scheduling process into three basic activities and focused on processor assignment to parallel loops. An algorithm was presented that gives an optimal solution at compile-time for processor assignments to complex parallel loops. This algorithm can be also implemented as a hardware device to perform processor assignment at run-time. We also presented some speedup measurements for EISPACK and IEEE subroutines, that result from the optimal assignment of processors to parallel loops. These measurements indicate that optimal processor assignments result in almost linear speedups on medium scale parallel processor machines.

## REFERENCES

[1] R.G. Cytron, "Compile-Time Scheduling and Optimizations for Multiprocessor Systems," Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Sept., 1984.

[2] "Programs for Digital Signal Processing," Edited by Digital Signal Processing Committee, IEEE Press, New York, 1979.

[3] D. J. Kuck, D. Lawrie, R. Cytron, A. Sameh, and D. Gajski, "The Architecture and Programming of the Cedar System," *Proceedings of the 1983 LASL Workshop on Vector and Parallel Processing,* Los Alamos, NM, August, 1983.

[4] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proceedings of the 8-th ACM Symposium on Principles of Programming Languages,* January 1981.

[5] D. J. Kuck, *The Structure of Computers and Computations,* Volume 1, John Wiley and Sons, New York, 1978.

[6] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference,* October, 1980.

[7] D.J. Kuck et. al., "The Effects of Program Restructuring, Algorithm Change and Architecture Choice on Program Performance," *Proceedings of the 1984 International Conference on Parallel Processing,* August, 1984.

[8] S. P. Midkiff, D. A. Padua, "Compiler Generated Synchronization for DO Loops," *Proc. of the 1986 Inter. Conference on Parallel Processing.*

[9] D. A. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-990, November, 1979.

[10] D. A. Padua, D. J. Kuck, D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Computers,* Vol C-24, No. 9, pp.. 763-776, Sept., 1980.

[11] C. D. Polychronopoulos, Ph.D. Thesis in preparation, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1986.

[12] C. D. Polychronopoulos, U. Banerjee, "Speedup Bounds and Processor Allocation of Parallel Programs on Multiprocessor Systems," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, August, 1986.

[13] B. J. Smith et. al., "Matrix Eigensystem Routines-Eispack Guide," Springer-Verlag, Heidelberg, 1976.

[14] Alex Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," Ph.D. Thesis, UILU-ENG-85-8012, Center for Supercomputing R&D, University of Illinois at Urbana-Champaign, 1985.

[15] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1982.

[16] J. J. Dongarra, "Comparison of the Cray X-MP-4, Fujitsu VP-200, and Hitachi S-810/20: An Argonne Perspective," Argonne National Laboratory, ANL-85-19, October, 1985.

# PROCESSOR SELF-SCHEDULING FOR MULTIPLE-NESTED PARALLEL LOOPS

Peiyi Tang and Pen-Chung Yew

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

## Abstract

Processor self-scheduling is a useful scheme in a multiprocessor system if the execution time of each iteration in a parallel loop is not known in advance and varies substantially, or if there are multiple nestings in parallel loops which makes static scheduling difficult and inefficient. By using efficient synchronization primitives, the operating system is not needed for loop scheduling. The overhead for the processor self-scheduling is small.

We presented a processor self-scheduling scheme for a single-nested parallel loop, and extend the scheme to multiple-nested parallel loops. Barrier synchronization mechanisms in the processors self-scheduling schemes are also discussed.

## 1. Introduction

Processor scheduling is a problem that must be solved in order to run a program on a multiprocessor system efficiently. The multiprocessor system considered here is a collection of identical processors with a global shared memory. Cray X-MP [1], Cedar [2], Ultracomputer [3] and RP3 [4] are some examples of such systems.

Parallel loops in a program, whose iterations can be executed concurrently on different processors, provide the greatest potential of parallelism to be exploited by multiprocessor systems. It is called a DOALL-loop, if the iterations of a parallel loop are independent. If there are data dependences across iterations of a DO-loop, its iterations can still be executed concurrently on different processors provided that the data dependences are enforced by synchronization across the processors during the execution [8]. This kind of parallel loop is called a DOACROSS-loop [11]. Both DOALL-loops and DOACROSS-loops can be nested in many levels. In this paper, we only consider DOALL-loops.

DOALL-loops can either be recognized by an optimizing compiler like Paraphrase [10] or explicitly specified by a programmer. Processors need to be scheduled properly so that execution time of a DOALL-loop is minimized. A *task* of scheduling here is one or several iterations of a DOALL-loop. We will only consider non-preemptive scheduling schemes, i.e., once a processor is assigned an iteration, it will continue to execute the iteration until its completion.

If execution time of each iteration of a DOALL-loop is the same, the optimal scheduling is to assign iterations evenly among processors. This scheduling can be done before program execution and is called *static* scheduling.

If execution time of each iteration is different and is not known until program execution, the scheduling of iterations to processors is more efficient if it can be done dynamically during the program execution, i.e., assign a new iteration to a processor whenever it becomes available. The total number of iterations assigned to each processor may not be equal, but the workload of each processor tends to be balanced. This scheduling is called *dynamic* scheduling. In this paper, we only consider dynamic scheduling.

Dynamic scheduling will incur scheduling overhead at run time. One technique to reduce overhead is to schedule several iterations (called a *chunk* of iterations) to a processor at a time. As long as the number of chunks is large enough, workload among processors can still be balanced.

Scheduling overhead can be very large if dynamic scheduling is done by system calls to the operating system. One way to reduce scheduling overhead is to use processor self-scheduling [5] [6] [7]. Rather than issuing a system call to the operating system for scheduling, processors can schedule themselves by fetch-and-adding a shared variable to get loop indices of a chunk of iterations. If a multiprocessor system has efficient hardware-implemented synchronization primitives like those in Cedar [8], Ultracomputer [9] and RP3 [4], scheduling overhead for a chunk of iterations can be reduced quite significantly.

However, so far, all processor self-scheduling schemes only deal with the outermost parallel DO-loop [5] [6] [7]. The rest of the parallel loops nested inside are treated as serial DO-loops. Hence, parallelism of the nested DOALL-loops is not fully exploited.

In this paper, we present a self-scheduling scheme for nested DOALL-loops using Cedar synchronization instructions. Cedar synchronization instructions [8] are briefly introduced in section 2. In section 3, we describe barrier synchronization mechanism needed in processor self-scheduling and processor self-scheduling schemes for single-nested DOALL-loops. In section 4, we present a processor self-scheduling scheme for multiple-nested DOALL-loops. An estimation of overhead for scheduling an iteration and the performance of the proposed processor self-scheduling schemes are given in section 5. In section 6, we have some concluding remarks.

This work was supported in part by the **National Science Foundation** under Grant Nos. US NSF DCR84-06916 and US NSF DCR84-10110, and the **US Department of Energy** under Grant No. US DOE DE FG02-85ER25001.

## 2. Cedar Synchronization Primitives

In a Cedar Multiprocessor System, a variable can be declared as a synchronization variable. A synchronization variable x has two fields: KEY and DATA. The KEY field is for storing synchronization information (which is an integer) and the DATA field is for storing the value of the variable (which can be a floating point number). The format of a Cedar synchronization instruction is as follows:

```
{x; test on KEY;
  operation on KEY; operation on DATA}.
```

Here x is the name (or the address) of the synchronization variable. The "test on x.KEY" specifies the condition to be tested between the KEY field of x (denoted by x.KEY) and a key provided by the instruction (denoted by i.KEY). The test includes $>$, $\geq$ $<$, $\leq$ $=$, $\neq$and NULL. The NULL test means that no test is needed and therefore the result of the test is always true. The operation on x.KEY can be Increment, Decrement, Add, Fetch, Fetch & Add, Store, Fetch & Increment, Fetch & Decrement and No Action. The operation on DATA can be Fetch, Store and No Action. The execution of the whole synchronization instruction, i.e. test on x.KEY, operations on x.KEY and x.DATA, is done in globally shared memory modules and is indivisible [8]. The operation on x.KEY and the operation on x.DATA are executed only when the result of the test is true. The memory module will inform the processor of a "failure" if the condition of the test is not satisfied, or a "success" if the condition is satisfied and execution of the instruction is completed. For some application, test on x.KEY has to be done repeatedly until the test condition becomes true. A star on the test condition (as shown below) is used to indicate this situation. In other words,

```
{x; (test on KEY)*;
  operation on KEY; operation on DATA }
```

is equivalent to

```
1: {x; test on KEY;
     operation on KEY; operation on DATA }
   if failure then goto 1
```

Cedar synchronization primitives are very effective in handling low level synchronizations required in numerical computations like enforcing data dependences across loop iterations [8]. In those applications, KEY field stores loop iteration numbers and DATA field usually stores the value of the data (which is usually a floating point number). In this paper, we use this synchronization primitives mostly for non-numerical scheduling problems. Hence, only the KEY field is used.

## 3. Barrier Synchronization

Following are several assumptions used in our processor self-scheduling schemes:

(1) We assume that the operating system assigns certain number of processors, say P processors, to a program before its execution. After that, the operating system will not be involved in scheduling.

(2) Code for processor self-scheduling is embedded in a user's program, and each processor will execute the same program.

(3) For the simplicity of the discussion, we assume that a task in our scheduling is only one iteration. A task whicb has several iterations can be similarly implemented.

The iterations of a DOALL-loop are scheduled through a shared variable, and after all of the iterations are scheduled, a *barrier synchronization* is needed for the completion of the DOALL-loop. In this section, we present two barrier synchronization mechanisms and processor self-scheduling schemes for a single-nested DOALL-loop.

In the first barrier synchronization mechanism, processors will be blocked at the *barrier* until all of the processors complete their tasks and arrive at the barrier. After that, the barrier will open and all of the processors can pass through. There is a counter which counts the number of arriving processors, and it must be reinitialized before the barrier can be reused for the second time. Algorithm 3.1 is a processor self-scheduling scheme for a DOALL-loop using this barrier synchronization mechanism.

**Algorithm 3.1**

```
/* J: loop index for the DOALL-loop with M iterations
   A: a counter to count the number of processors
      that have arrived at the barrier
   B: a variable acting as a barrier */

/* Initially J=1, A=P and B=0, where P is the
   total number of the processors assigned to
   execute the program. */

$L_1$: {J; $\leq$M; Fetch(LOCJ)&Increment}
   if failure then goto $L_2$
     .
     .
     . /* Original Do loop-body with index LOCJ */
     .
     .
   goto $L_1$
   /* Processors go back to $L_1$
      to get another iteration. */

$L_2$: {A; $>$1; Decrement}
   if failure then
        begin
         J:=1; A=P; B:=P
        end
   {B; ($>$0)*; Decrement}
   /* B acts as a barrier and is reinitialized
      after P processors pass through. */
```

If a DOALL-loop is enclosed in an outer serial loop as shown in Figure 3.1, the implementation of the outer serial loop is quite simple. Since all of the processors are synchronized at the barrier of the second DOALL-loop $J_2$, each processor can have a local copy of loop index variable LOCI and update it after the execution of the DOALL-loop $J_2$. The processor self-scheduling code for the whole program is illustrated in Figure 3.2.

However, a barrier synchronization based on the number of arriving processors has a problem. If the actual number of processors assigned to the program is less than that specified in the program (i.e. P processors), the barrier of $J_1$ will never open and all of the processors will be stuck there. In other words, the operating system must assign exactly the same number of processors as specified in the program; otherwise, we will have a deadlock.

However, the precedence relation only requires that all of the iterations of the first DOALL-loop be completed before the second DOALL-loop can be started. In the following, a barrier is controlled by the number of completed iterations of the DOALL-loop instead of by the number of arriving processors. Counter A is initialized to M, the total number of iterations, instead of P. When all of the iterations are completed, the barrier will be opened. Thus, this barrier synchronization mechanism will work even if the operating system assigns fewer than P processors to the program. The processor self-scheduling scheme using this barrier synchronization mechanism is shown in Algorithm 3.2

**Algorithm 3.2**

```
/* C: a lock to block processors after P processors
      have entered the DOALL-loop
   J: loop index for the DOALL-loop with M iterations
   A: a counter to count the number of completed iterations
   B: a variable acting as a barrier
   T: a counter to detect the last processor,
      which is responsible for reinitializing
      all of the synchronization variables */
/* Initially C=P, J=1, A=M, B=0 and T=P.
   P is the number of processors assigned
   to the program.                 */

  {C; (>0)*; Decrement}
  /* This is to block processors after P processors
      have entered the DOALL-loop. */
$L_1$: {J; ≤M; Fetch(LOCJ)&Increment}
  if failure then goto $L_2$
     .
     .
     . /* Original Do loop-body with index LOCJ */
     .
     .
  {A; >1; Decrement}
  if failure then B:=1
  /* Counter A counts the number of completed
     iterations and controls the opening of
     the barrier.            */
  goto $L_1$
  /* Processors go back to $L_1$ to get
     another iteration. */
 $L_2$: {B; (>0)*; No Action}
  /* B acts as a barrier.    */
  {T; >1; Decrement}
  if failure then
  /* This is to reinitialize all of the
     synchronization variables. */
    begin
      T:=P; B:=0; A:=M
      J=1; C:=P
    end
```

Note that all of the synchronization variables in the algorithm will be reinitialized after P processors have passed the variable T. If there are fewer than P processors assigned to the program, some processors must come back and make up the discrepancy, and the synchronization variables will be reinitialized eventually.

Lock C in the beginning of the code is used to prevent the race problem caused by the delay in reinitializing the barrier. Note that there is a delay between the time when all of the P processors pass the barrier and the time it is reinitialized. During this period of time, it is possible that some processors may revisit this DOALL-loop again for the next iteration of the outer serial loop. If there is no such a lock to block those fast processors, they will pass the barrier for the second time and start executing the following DOALL-loop before this DOALL-loop is restarted.

The disadvantage of the barrier synchronization mechanism based on the number of completed iterations is that the implementation of the outer serial DO-loop is more complicated. Because processors are no longer synchronized after each iteration of the outer serial DO-loop, the number of the times each processor will traverse the outer serial DO-loop may be different. A shared index variable for the outer serial DO-loop is thus needed. The branch node needed to implement the outer serial DO-loop is given in Algorithm 3.3. Figure 3.3 illustrates the processor self-scheduling code for the entire program in Figure 3.1 using this barrier synchronization mechanism.

**Algorithm 3.3**

```
/* C: a lock to block processors after P processors
      have entered the branch node.
   S: a semaphore to guarantee that only one
      processor can control the outer serial loop
   B: a variable acting as a barrier
   T: a counter to detect the last processor,
      which is responsible for reinitializing
      all of the synchronization variables */

/* Initially C=P, S=1,B=0 and T=P, where P is
   the number of processors assigned to the program. */

  {C; (>0)*; Decrement}
  /* This is to block processors after P processors
      have entered the branch node. */
$L_1$: {S; >0; Decrement}
  if failure then goto $L_2$
  /* This is to allow only one processor
      to execute the following code. */
  if  I<N then
      begin
        COND:= true; I:=I+1
      end
     else
      begin
        COND:= false; I:= 1
      end
  /* Update the shared index variable I
      and resolve the branch condition,
      which is stored in COND */
  B:=1
  /* Open the  barrier B */
$L_2$: {B; (>0)*; No Action}
  MYCOND:=COND
  /* Save the branch condition */
  {T; >1; Decrement}
  if failure then
     begin
```

```
    T:=P; B:=0; S:=1; C:=P
  end
/* This is to reinitialize all of the
   synchronization variables. */
if MYCOND=true then goto START
/* START is the head of the serial DO-loop. */
```

## 4. Processor Self-Scheduling for Nested DOALL-loops

In this section, we extend the processor self-scheduling schemes to nested DOALL-loops. Before going further, we have to define a few terms which will be used later.

If each inner DOALL loop is surrounded immediately by an outer DOALL loop with no scalar code between them, this multiple DOALL loop nestings is a **perfectly-nested DOALL structure**.

Otherwise, a nested DOALL structure is called a **non-perfectly-nested DOALL structure.** Figure 4.1 shows an example of non-perfectly-nested DOALL structure. There, we use a left bracket "[" to denote each DOALL-loop nesting. In a non-perfectly-nested DOALL structure, each innermost DOALL loop nesting will contain a loop body. The loop-body with all of its surrounding DOALL-loop nestings form a **nested DOALL component**. For example, the non-perfectly-nested DOALL structure in Figure 4.1 has three nested DOALL components. A perfectly-nested DOALL structure contains only one nested DOALL component.

We only discuss self-scheduling scheme of a non-perfectly-nested structure. A perfectly-nested DOALL structure is a special case of a non-perfectly-nested DOALL structure.

Assume that a non-perfectly-nested DOALL structure like the one in Figure 4.1 consists of $m$ nested DOALL components. Each nested DOALL component can be characterized by two vectors: an index variable vector and a loop-bound vector. The index variable vector of a nested DOALL component $(I_1,I_2,...,I_d)$ is a vector which consists of the index variables of the loop nestings. Here, $I_1$ is the index variable of the outermost nesting and d is the depth of the loop nestings. The loop-bound vector $(N_1,N_2,...,N_d)$ is a vector of the corresponding loop-bounds. For example, the index variable vectors for the three nested DOALL components in Figure 4.1 are $(I_1,I_2)$, $(I_1,J_2,J_3)$ and $(I_1,J_2,K_3)$, respectively, and their corresponding loop-bound vectors are (3,7), (3,2,3) and (3,2,4).

For a nested DOALL component with index variable vector $(I_1,I_2,...,I_d)$ and loop-bound vector $(N_1,N_2,...,N_d)$, there are total of $N_1 \cdots N_d$ iterations for the component. Each of them can be identified by a sequence number from 1 to $N_1 \cdots N_d$. If we split the d loop nestings at level k $(1 \leq k < d)$ into $(I_1,...,I_k)$ and $(I_{k+1},...,I_d)$, we can notice that there are $N_{k+1} \cdots N_d$ iterations of the component that share common loop indices of the k outermost nestings $(I_1,...,I_k)$. An iteration with $I_1=i_1$, $I_2=i_2$, ..., $I_k=i_k$ is called an **iteration at level k.** The sequence number for the iteration is defined to be

$$i=(i_1\text{-}1)N_2 \cdots N_k + \cdots + (i_{k-1}\text{-}1)N_k + i_k.$$

There are $N_1 \cdots N_k$ iterations at level k, each of which can be identified by a sequence number from 1 to $N_1 \cdots N_k$.

Let $(I_1,I_2,...,I_{d_j})$ and $(N_1,N_2,...,N_{d_j})$ be the index variable vector and the loop-bound vector of the $j$-th component. It has two parameters $k_j$ and $l_j$ $(1 \leq j \leq m,\ 1 \leq k_j \leq d_j,\ 1 \leq l_j \leq d_j)$, defined as follows: if its deepest common loop nesting with the $(j\text{-}1)$-th component is at $(p\text{-}1)$ level, we will have $k_j=p$. Similarly, if the deepest common loop nesting it shares with the $(j+1)$-th component is at $(q\text{-}1)$ level, we have $l_j=q$. From the definition, we have $l_j=k_{j+1}$, $1 \leq j < m$. We also define that $k_1=1$ and $l_m=1$. For the non-perfectly-nested DOALL structure in Figure 4.1, we have $k_1=1$, $l_1=2$, $k_2=2$, $l_2=3$, $k_3=3$, $l_3=1$.

Assume that the deepest common loop nesting for the $(j\text{-}1)$-th and the $j$-th components is at level $p\text{-}1$, i.e. $l_{j-1}=k_j=p$ (See Figure 4.2). According to the semantics of nested DOALL-loops, an iteration at level p-1 for the j-th component can not be started until the corresponding iteration for the (j-1)-th component is completed. We use a variable $B^{j-1}$ to enforce this precedence constraint. When $B^{j-1}=t$, the first t iterations at level p-1 for the (j-1)-th component are completed, and, hence, we can start to schedule the j-th component for those iterations.

The components of a DOALL loop structure will be scheduled in sequence, and the iterations of each component will be scheduled in the order of their sequence numbers. Thus, the processor self-scheduling schemes proposed here for nested DOALL-loops can also be applied to nested DOACROSS-loops or mixture of nested DOALL-loops and DOACROSS-loops without causing deadlocks.

Processor self-scheduling for the j-th nested DOALL component is realized by fetch-and-incrementing an **index control variable $I^j$.** The initial value of $I^j$ is 1, which corresponds to the first index vector (1,1,...,1). The maximum value of $I^j$ is $M_j=N_1N_2 \cdots N_{d_j}$, which corresponds to the last index vector $(N_1,N_2,...,N_{d_j})$. Let $F_{N_1,N_2,\cdots,N_{d_j}}$ be a function that maps the sequence number of an iteration into its index vector. Table 4.1 shows an example of function $F_{2,3,2}$. Function $F_{N_1,N_2,\cdots,N_{d_j}}$ can be formulated as follows:

$$F_{N_1,N_2,\cdots,N_{d_j}}[x] \rightarrow (i_1,i_2,\cdots,i_{d_j}),$$

where for $1 \leq k \leq d_j$, we have

$$i_k = \left\lfloor \frac{x\text{-}1}{N_{k+1} \cdots N_{d_j}} \right\rfloor mod\ N_k + 1.$$

$\lfloor t \rfloor$ is the largest integer smaller than $t$. This computation can be done locally in a processor after it fetches a sequence number from $I^j$. However, as mentioned before, if the sequence number of an iteration at level p-1 for the j-th component is larger than $B^{j-1}$, it can not be started. Given a sequence number x of an iteration, the sequence number of the iteration at level p-1 is

$$y = \left\lfloor \frac{x\text{-}1}{N_p \cdots N_{d_j}} \right\rfloor + 1.$$

So, before starting to execute an iteration of the component, a processor needs to check if $B^{j-1} \geq y$. If $B^{j-1} < y$, a processor has to wait until $B^{j-1}$ becomes larger.

The processor self-scheduling code for the non-perfectly-nested DOALL structure is given in Algorithm 4.1.

**Algorithm 4.1**

```
{C; (>0)*; Decrement}
    .
    .
    .
/* The code for the j-th component starts here.
  Initially $I^j$=1, $B^j$=0,
  $A^j(i_1,i_2, \ldots, i_{q-1})=N_q N_{q+1} \cdots N_{d_j}$
       $(1\leq i_1\leq N_1, 1\leq i_2\leq N_2, \ldots, 1\leq i_{q-1}\leq N_{q-1})$ */
$L^j$: {$I^j$; $\leq M_j$; Fetch(x)&Increment}
   /* $M_j=N_1 \cdots N_{d_j}$. */
   if failure then goto $L^{j+1}$
   /* $L^{j+1}$ is the beginning of the next component */
   y:= $\lfloor (x-1)/(N_p \cdots N_{d_j}) \rfloor + 1$
   {$B^{j-1}$; $(\geq y)$*; No action}
   for k= 1 to $d_j$ do
     $LOCI_k := \lfloor (x-1)/(N_{k+1} \cdots N_{d_j}) \rfloor \bmod N_k + 1$
   /* Compute the index vector from the sequence
      number x using function $F_{N_1,\ldots,N_{d_j}}$ */
    .
    .
    . /* original Loop-body with index vector
    .    $(LOCI_1, LOCI_2, \cdots, LOCI_{d_j})$ */
    .
    .
   {$A^j(LOCI_1, LOCI_2, \ldots, LOCI_{q-1})$; >1; Decrement}
    if failure then
      begin
        $A^j(LOCI_1, \cdots, LOCI_{q-1}):=N_q \cdots N_{d_j}$
        z := $\lfloor (x-1)/(N_q \cdots N_{d_j}) \rfloor + 1$
        {$B^j$; (=z-1)*; Increment}
      end
    goto $L^j$
 /* The j-th component ends here.  */
    .
    .
$L^{m+i}$: {$B^m$; (=1)*; No action}
    /* $B^m$ acts as a barrier. */
    {T; >1: Decrement}
    if failure then
     begin
      T:=P
      $B^1$:=0; $I^1$:=1
      $B^2$:=0; $I^2$:=1
       .
       .
       .
      $B^m$:=0; $I^m$:=1
      C:=P
    end
```

Algorithm 4.1 consists of three portions. First portion is just a single statement

{C; (>0)*; Decrement}

which functions as a lock in the beginning of the code to prevent race problem. Lock C is initialized to P, the number of processors assigned to the program. The second portion is the main self-scheduling code for each nested DOALL component. They are the same except for the different parameters such as loop-bound vector, $k_j$ and $l_j$. We only present the code for the j-th component.

The main self-scheduling code for each component consists of three parts: loop self-scheduling, the original loop-body and bookkeeping of the completed iterations. We already discussed the "self-scheduling" part. In bookkeeping of the completed iterations, recall that for each iteration at level q-1, there are total of $N_q \cdots N_{d_j}$ iterations for the $(d_j-q+1)$ innermost loop nestings of the component. We use an element of array $A^j$ to record the total remaining iterations. It will be decremented by 1 after each iterations of the innermost $(d_j-q+1)$ loops has been completed.

When $N_q \cdots N_{d_j}$ iterations of the innermost $(d_j-q+1)$ nestings are completed, the element of $A^j$ is reinitialized. Given the sequence number x of the iteration fetched from $I^j$, the sequence number z of the corresponding iteration at level q-1 is

$$z=\left\lfloor \frac{x-1}{N_q \cdots N_{d_j}} \right\rfloor +1.$$

$B^j$, which is used to control the execution of the (j+1)-th component, is then updated. Note that $B^j$ is incremented only when its value is one smaller than the calculated sequence number. Thus, when $B^j$=s, it is guaranteed that the iterations at level q-1 with the sequence number from 1 to s have been completed.

The last portion of Algorithm 4.1 is for barrier synchronization and reinitialization. $B^m$ acts as a barrier for the entire non-perfectly-nested DOALL structure. In fact, since $l_m$=1, $B^m$ has only two values: 0 and 1. As in Algorithm 3.2, T is used to detect the last processor leaving the entire DOALL structure. That processor is responsible for reinitializing variables $B^j$, $I^j$ $(1\leq j\leq m)$, T and C. Array $A^j$ $(1\leq j\leq m)$ is reinitialized in the code for each component in the second portion. The barrier synchronization used here is based on the number of completed iterations. If the nested DOALL structure is within an outer serial loop, one has to use algorithm 3.3 to implement branch node.

Notice that iterations for the first nested DOALL component in a non-perfectly-nested DOALL structure can be scheduled without restriction. It is not necessary to check the value of $B^0$. This is why we set $k_1$=1 and $B^0$ is not used.

The entire processor self-scheduling code for the whole nested DOALL structure in Figure 4.1 is shown in Figure 4.3.

## 5. Overhead and Performance

Processor self-scheduling allows us to have better utilization of processors. During the program execution time, whenever a processor becomes available, it will grab another iteration and start executing the new iteration as long as the precedence relation in the program is not violated. Hence, the program execution time can actually be improved.

In a large multiprocessor system, global memory accesses will take very significant amount of time, hence, we will only consider global memory accesses (remember that Cedar synchronization primitives are global memory accesses) when we estimate the scheduling overhead. In Algorithm 3.1 and Algorithm 3.2, the overhead of scheduling an iteration of a single-nested parallel loop is only one Cedar synchronization instruction, which is one global memory access to the index control variable J. The overhead of scheduling an iteration of multiple-nested parallel loops is two global memory accesses as shown in Algorithm 4.1: one for testing and fetch-and-adding index control variable $I^j$ and one for checking variable $B^{j-1}$. The later is not needed in the case of perfectly nested parallel loops, i.e. the scheduling overhead for an iteration of a perfectly nested parallel loop is only one global memory access. This overhead is very small compared to the overhead that would incur if it is done by the operating system.

It is very important to note that the barrier synchronizations in the end of parallel loops are also needed in the static scheduling schemes. Hence, they are not extra overhead for the self-scheduling schemes.

Let us compare the self-scheduling scheme with the static scheduling scheme proposed in [12]. Assume that the execution times for each iteration of the nested DOALL structure is the same.

For a single-nested DOALL loop with N iterations, both schemes will schedule N iterations over P processors evenly. The program execution time will be the same for both schemes, i.e. it will be $\lceil N/P \rceil$ times the execution time of an iteration. However, processor self-scheduling will require one extra global memory access and, hence, it will have slightly more overhead than static scheduling scheme.

However, for multiple-nested DOALL loops, things can be quite different. Assume that we have a perfectly-nested DOALL-loop structure with loop-bound vector $(N_1, N_2, \cdots, N_d)$. Using static scheduling scheme, we have to find the optimal decomposition of $P = P_1 \cdots P_d$, where $P_i$ is the number of processors assigned to the i-th loop nesting, such that $\lceil N_1/P_1 \rceil \cdots \lceil N_d/P_d \rceil$ is minimized [12]. The completion time for this static scheduling is

$$T_{st} = \tau \lceil N_1/P_1 \rceil \cdots \lceil N_d/P_d \rceil \tag{5.1}$$

where $\tau$ is the execution time of an iteration. Using processor self-scheduling in Algorithm 4.1, we have

$$T_{sf} = (\tau + \sigma) \lceil N/P \rceil, \tag{5.2}$$

where $\sigma$ is the extra time needed for 1 extra Cedar synchronization instruction, and $N = N_1 N_2 \cdots N_d$. Note that

$$\lceil N/P \rceil \leq \lceil N_1/P_1 \rceil \cdots \lceil N_d/P_d \rceil \tag{5.3}$$

The equality in (5.3) holds only for some special cases. Let us ignore those special cases and assume that $(\lceil N_1/P_1 \rceil \cdots \lceil N_d/P_d \rceil) - \lceil N/P \rceil = k$, $k \neq 0$. $T_{sf}$ is less than $T_{st}$ when

$$\tau > \frac{\sigma}{k} \left\lceil \frac{N}{P} \right\rceil.$$

For the non-perfectly-nested DOALL-loop structure in Figure 4.1, assume that there are P=8 processors. Using the static scheduling scheme in [12], we get the optimal decomposition $P = 1 \times 2 \times 4$, i.e., using 1 processor for the outermost loop nesting, 2 processors for the second outermost loop nesting and 4 processors for the innermost loop nesting. The completion time is $9\tau$. Using self-scheduling, the completion time is $8(\tau + 2\sigma)$. In self-scheduling scheme, a processor is not tied to any specific loop nesting, thus, processors can be better utilized.

## 6. Concluding Remarks

We present processor self-scheduling scheme for both single- and multiple-nested parallel loops. Two different barrier synchronization mechanism are discussed. The scheduling overheads for these schemes are quite small if synchronization primitives are supported in the system as in Cedar [2].

Processor utilization can be improved over static scheduling scheme, which can lead to better program execution time if the execution time of each loop iteration varies substantially, or if there are multiple loop nestings.

## REFERENCES

(1) S.C. Chen, "Large-Scale and High-Speed Multiprocessor System for Scientific Applications--Cray X-MP-2 Series", Proc. NATO Advanced Research Workshop on High Speed Computation, J. Kowalik, ed., Springer Verlag, Julish, West Germany, June 1983.

(2) David J. Kuck, et al, "CEDAR", Proc. of COMPCON, Spring 1984.

(3) A. Gottlieb, et al., "The NYU Ultracomputer - Designing an MIMD Shared Memory parallel Computer", IEEE Trans. on Computers, Feb. 1983.

(4) G. F. Pfister, et al., "The IBM RP3 Introduction and Architecture", Proc. of 1985 International Conf. on parallel Processing, Aug. 1985.

(5) Peiyi Tang, Pen-Chung Yew and Chuan-Qi Zhu,"Processor Self-scheduling in large Multiprocessor Systems", Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Rpt No. 536 (Oct. 1985), presented in Second SIAM Conference on Parallel Processing for Scientific Computing, Nov. 18-21, 1985.

(6) F. Darema-Rogers, D. A. George, V. A. Norton and G. F. Pfister, "A VM parallel Environment", RC 11225(#49161), IBM T.J. Watson Research Center, Yorktown Heights, Jan. 23, 1985.

(7) E. L. Lusk and R. A. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and other parallel Processors", Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois (December 1983).

(8) Chuan-Qi Zhu and Pen-Chung Yew, "A Synchronization Scheme and its Applications for Large Multiprocessor Systems", Proc. 4th International Conference on Distributed Computing Systems, May 14-18, 1984, pp. 486-493.

(9) A. Gottlieb, B. D. Lubachevsky and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processes", ACM Trans. on Programming Language and Systems, Vol.5, No.2, April 1983, pp. 164-189.

(10) David J. Kuck, et al., "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance", Proceedings of the 1984 International Conference on parallel Processing, August 21-24, 1984, pp. 129-138.

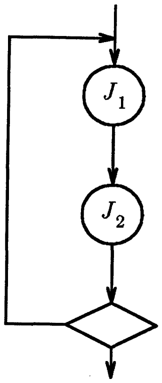
(11) R. G. Cyntron, "Compile-Time Scheduling and Optimization for Asynchronous Machines", Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-84-1177. (OCT. 1984)

(12) Constantine D. Polychronopoulos, David J. Kuck and David

A. Padua, "Execution of Parallel Loops on Parallel Processor Systems", Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Rpt. No. 552, Jan. 1986, to appear in Proceedings of 1986 International Conference on Parallel Processing.

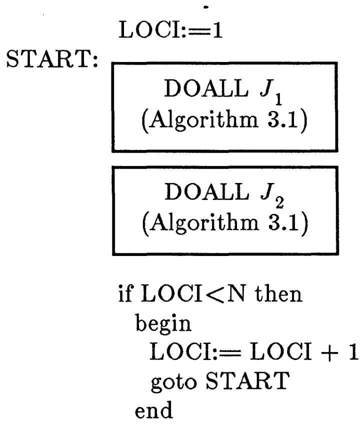
```
DOSERIAL I=1,N
 DOALL J1=1, M1
 .
 .
 ENDDOALL

 DOALL J2=1, M2
 .
 .
 ENDDOALL
ENDDOSERIAL
```

(a)



(b)

Figure 3.1 An example of parallel program



```
        LOCI:=1
START:  DOALL J1 (Algorithm 3.1)
        DOALL J2 (Algorithm 3.1)

        if LOCI<N then
          begin
            LOCI:= LOCI + 1
            goto START
          end
```

Figure 3.2 Outer serial loop control (scheme 1)

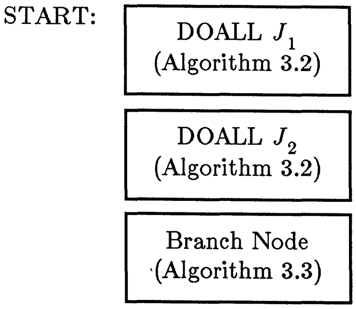


Figure 3.3 Outer serial loop control (scheme 2)

| x | $F_{2,3,2}(x)$ | | |
|---|---|---|---|
| | $i_1$ | $i_2$ | $i_3$ |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 |
| 3 | 1 | 2 | 1 |
| 4 | 1 | 2 | 2 |
| 5 | 1 | 3 | 1 |
| 6 | 1 | 3 | 2 |
| 7 | 2 | 1 | 1 |
| 8 | 2 | 1 | 2 |
| 9 | 2 | 2 | 1 |
| 10 | 2 | 2 | 2 |
| 11 | 2 | 3 | 1 |
| 12 | 2 | 3 | 2 |

Table 4.1 Function $F_{2,3,2}$

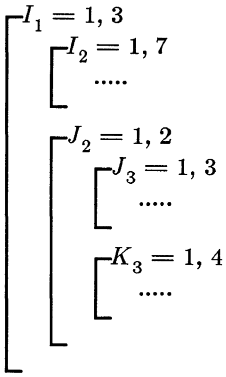


Figure 4.1 A non-perfectly-nested DOALL structure

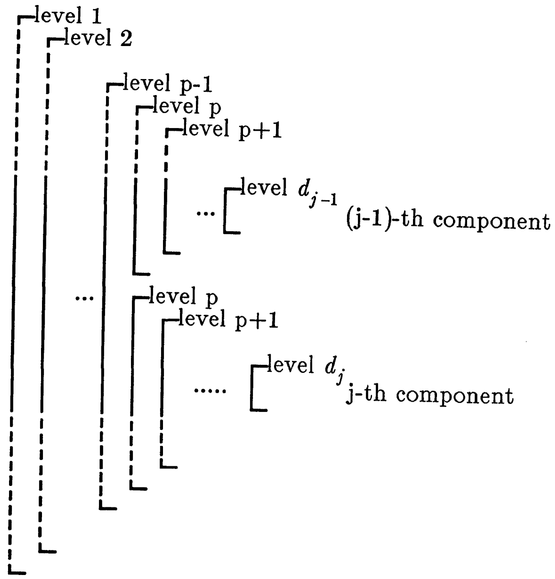


Figure 4.2 p-1 outermost common loop nestings

```
{C; (>0)*; Decrement}

$L^1$: {$I^1$; $\leq 3\times 7$; Fetch(x)&Increment}
  if failure then goto $L^2$
  $(LOCI_1, LOCI_2) \leftarrow F_{3,7}(x)$
   .
   . /*loop body 1 with $(LOCI_1, LOCI_2)$*/
   .
  {$A^1(LOCI_1)$; $>1$; Decrement}
  if failure then
   begin
    $A^1(LOCI_1) := 7$
    z := $\lfloor (x - 1) / 7 \rfloor + 1$
    {$B^1$; (=z-1)*; Increment}
   end
  goto $L^1$

$L^2$: {$I^2$; $\leq 3\times 2\times 3$; Fetch(x)&Increment}
  if failure then goto $L^3$
  y := $\lfloor (x - 1) / (2\times 3) \rfloor + 1$
  {$B^1$; ($\geq$y)*; No action}
  $(LOCI_1, LOCJ_2, LOCJ_3) \leftarrow F_{3,2,3}(x)$
   .
   . /*loop body 2 with $(LOCI_1, LOCJ_2, LOCJ_3)$*/
   .
  {$A^2(LOCI_1, LOCJ_2)$; $>1$; Decrement}
  if failure then
   begin
    $A^2(LOCI_1, LOCJ_2) := 3$
    z := $\lfloor (x - 1) / 3 \rfloor + 1$
    {$B^2$; (=z-1)*; Increment}
   end
  goto $L^2$

$L^3$: {$I^3$; $\leq 3\times 2\times 4$; Fetch(x)&Increment}
  if failure then goto $L^4$
  y := $\lfloor (x - 1) / 4 \rfloor + 1$
  {$B^2$; ($\geq$y)*; No action}
  $(LOCI_1, LOCJ_2, LOCK_3) \leftarrow F_{3,2,4}(x)$
   .
   . /*loop body 3 with $(LOCI_1, LOCJ_2, LOCK_3)$*/
   .
  {$A^3$; $>1$; Decrement}
  if failure then
   begin
    $A^3 := 3\times 2\times 4$
    z := 1
    {$B^3$; (=z-1)*; Increment}
   end
  goto $L^3$

$L^4$:{$B^3$; (=1)*; No action}
  {T; $>1$; Decrement}
if failure then
  begin
   T:=P
   $B^1$:=0; $I^1$:= 1
   $B^2$:=0; $I^2$:= 1
   $B^3$:=0; $I^3$:= 1
   C:=P
  end
```

Figure 4.3 Self-scheduling code for DOALL structure in Figure 4.1

# Advanced Loop Interchanging

Michael Wolfe

Kuck and Associates, Inc.
1808 Woodfield Drive
Savoy, IL 61874

## Abstract

This paper discusses some advanced topics related to interchanging DO loops in numerical programs. DO loop interchanging is used for many reasons; one well-known use of loop interchanging is to vectorize an outer DO loop when the inner DO loop must be left serial. In this paper we study loop interchanging as an end in itself. First, we describe the KAP/Design, a special program which, when given a nest of DO loops, prints out all the ways in which those DO loops can be interchanged. We describe some of the special transformations that are used by the KAP/Design. Second, a new type of loop interchanging, interchanging of imperfectly nested loops, is described.

## 1. Introduction

This paper discusses interchanging of DO loops in numerical programs. Loop interchanging is often used to uncover parallel operations in nested DO loops; when an inner DO loop cannot be vectorized, for instance, loop interchanging is used to bring a different loop to the innermost nest level which perhaps can be vectorized [1,2]. Loop interchanging has also been shown to be useful to modify the way arrays are accessed (to reduce bank conflicts or to reduce page faults [1,3,4]) or to change the way registers are allocated (vector registers in particular, allowing "super-vector" performance on vector pipeline machines [2]).

The KAP is a family of powerful retargetable vectorizers which use loop interchanging heavily [5,6,7]. We have studied loop interchanging and its applications extensively. This paper presents some of the results of this research.

The second section describes a special version of the KAP, called the KAP/Design, which prints out all legal ways to interchange a nest of DO loops. Examples of its use are given with an explanation of the problems we encountered while developing the KAP/Design. The third section describes a technique for interchanging imperfectly nested DO loops; this technique is not as simple as distributing the outer DO loop, but is something totally new.

## 2. KAP/Design

As pointed out in [8], different formulations of an algorithm sometimes differ only in having the DO loops interchanged. Some algorithm designers even stated that it might be useful to have a program source translator that printed out all the ways in which a DO loop nest might be interchanged. The KAP vectorizer includes a powerful loop interchanging algorithm, and it was a small matter to make a version that would interchange a nest of loops all possible ways and print each interchanged loop nest out. This became KAP/Design; an example of KAP/Design output on a simple matrix multiply program is shown in Figure 1. Normally the KAP/Design would be hardly noteworthy, but some of the problems that were encountered while designing and using the KAP/Design are interesting.

### 2.1 Simple Loop Interchanging

This section explains briefly how the KAP/Design tests for the legality of loop interchanging. A nest of DO loops, such as the two-nested loop in Figure 2(a), can be thought of as traversing a two-dimensional iteration space, shown in Figure 2(b). The arrows in Figure 2(b) represent how the serial DO loops execute the statements in the loop for iteration (I=1,J=1) first, then (1,2), (1,3), ..., (1,5), then incrementing the I index to go to (2,1), (2,2), ..., (2,5), ..., (5,1), ..., (5,5). Interchanging these two DO loops means changing the order in which the iteration space is traversed; by interchanging the loop as in Figure 3(a), the iteration space would be executed in the order shown in Figure 3(b).

There may also be data dependence relations between the iterations of a DO loop. Simple data dependence relations in scalar code are represented in the following program segment:

```
S1:    X = Z
S2:    Y = X
S3:    Z = Z + 1
S4:    X = 9.
```

In this short program segment, we say that statement S2 depends on S1 (flow-dependence) since the value of X used in S2 is assigned in S1; this means that statement S2 cannot be moved above S1 without changing the answers.

0190-3918/86/0000/0536 $01.00 © 1986 IEEE

We say that S3 depends on S1 (anti-dependence) since the value of Z assigned in S3 is not the value used by S1; this means that S3 cannot be moved above S1 without changing the answers. Finally, we say that S4 depends on S1 (output-dependence) since the value assigned to X in S4 is assigned after the value in S1 is assigned; again, S4 cannot be moved above S1 without changing the program. More on data dependence can be found in [9,10,11,12,13].

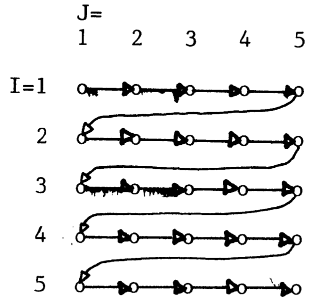
For interchanging loops, the KAP/Design is not so much interested in the data dependence relations between the statements in a loop as between the iterations of a loop. In the loop in Figures 2 and 3, the value of A(1,2) used in iteration (I=1,J=2) was assigned in iteration (I=1,J=1). In fact, the value used in iteration (i,j) was assigned in iteration (i,j-1) (except for boundary values); this relation is shown in the iteration space dependence graph in Figure 4. Notice that the pattern of the dependence flow in the iteration space can be characterized by the direction or the distance of the flow with respect to the loop index variables. In this example, the dependence distance would be called (0,-1), since the distance in the I loop is zero, and the distance in the J loop is -1. The KAP/Design saves the sign of the dependence distance; here it would save (0,-), or (=,<), to characterize the data dependence in the loop ((=,<) is the data dependence direction vector).

```
      SUBROUTINE MATMUL2( A,B,C,N,M,P )
      REAL A(N,M), B(N,P), C(P,M)
      INTEGER N,M,P,I,J,K
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 100 I = 1,N
      DO 100 J = 1,M
      DO 100 K = 1,P
100    A(I,J) = A(I,J) + B(I,K)*C(K,J)
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 101 I=1,N
      DO 101 K=1,P
      DO 101 J=1,M
101    A(I,J) = A(I,J) + B(I,K) * C(K,J)
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 102 J=1,M
      DO 102 I=1,N
      DO 102 K=1,P
102    A(I,J) = A(I,J) + B(I,K) * C(K,J)
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 103 J=1,M
      DO 103 K=1,P
      DO 103 I=1,N
103    A(I,J) = A(I,J) + B(I,K) * C(K,J)
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 104 K=1,P
      DO 104 I=1,N
      DO 104 J=1,M
104    A(I,J) = A(I,J) + B(I,K) * C(K,J)
C
C     MATRIX MULTIPLICATION LOOP
C
      DO 105 K=1,P
      DO 105 J=1,M
      DO 105 I=1,N
105    A(I,J) = A(I,J) + B(I,K) * C(K,J)
      END
```

Figure 1.

```
      DO 100 I = 1,5
      DO 100 J = 1,5
100    A(I,J+1) = A(I,J) + B(I,J)
```

(a)

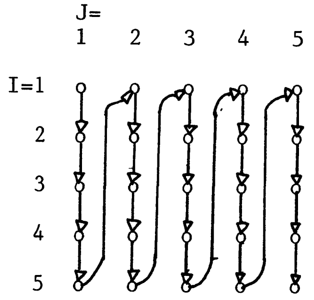


(b)

Figure 2.

```
      DO 100 J = 1,5
      DO 100 I = 1,5
100    A(I,J+1) = A(I,J) + B(I,J)
```

(a)



(b)

Figure 3.

The possible directions in a two dimensional iteration space (corresponding to a doubly-nested DO loop) are shown in Figure 5. Figure 5(a) shows the data dependence directions that are preserved by loop interchanging; Figure 5(b) shows the data dependence directions that prevent loop interchanging. A pair of loops with a (<,>) data dependence direction vector cannot be interchanged (without interaction from outer loops; see [1,2]).

The KAP/Design discovers all the data dependence relations in the DO loop, and saves the corresponding data dependence direction vectors. To perform DO loop interchanging on more than two loops, it does repeated pairwise interchanging.

## 2.2 Triangular Loops

One problem is that most studies of loop interchanging ignore the loop bounds [1,2,3]. Detailed discussions of how to test whether two loops can be interchanged explain the data dependence conditions, but assume that the inner DO loop bound is invariant in the outer loop. Many programs include triangular DO loop bounds, such as the loop nest in Figure 6(a). Here, the upper bound of the inner DO J loop is a simple function of the outer loop index. The iteration space traversed by this DO loop pair is drawn in Figure 6(b); it is easy to see why this is called a triangular loop. To properly interchange these loops it is necessary to modify the loop bounds, as in Figure 6(c). This is no more difficult than modifying the bounds of a double summation when interchanging the summations. The types of triangular loop bounds are classified by the shape of the triangle in the iteration space; the triangle in Figure 6(b) is a lower-left triangle. The four types of triangles are the lower/upper-left/right; four more types may be distinguished by including or excluding the diagonal (Figure 6(b) excludes the diagonal). These are characterized by the loop bounds:

```
              with diagonal:       without diagonal:

  lower left:
              DO 100 I = M,N       DO 100 I = M,N
              DO 100 J = M,I       DO 100 J = M,I-1

  upper right:
              DO 100 I = M,N       DO 100 I = M,N
              DO 100 J = I,N       DO 100 J = I+1,N

  upper left:
              DO 100 I = M,N       DO 100 I = M,N
              DO 100 J = M,N-I+M   DO 100 J = M,N-I+M-1

  lower right:
              DO 100 I = M,N       DO 100 I = M,N
              DO 100 J = N-I+M,N   DO 100 J = N-I+M+1,N
```

Interchanging a triangular loop can be thought of as transposing the iteration space about its major diagonal. Thus, a lower left triangle would be transposed into an upper right triangle, and vice versa. The lower right and upper left triangles would be transposed into themselves.

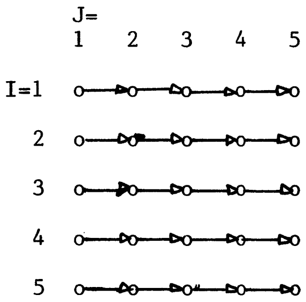


Figure 4.

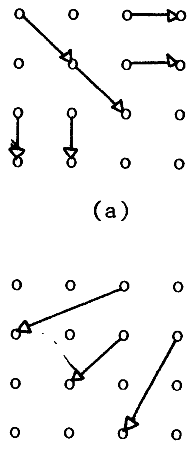


Figure 5.

```
      DO 100 I = 1,N
      DO 100 J = 1,I-1
  100 A(I,J) = A(I,J) + B(I,J)
```

(a)

```
      J=
      1   2   3 ...     N

  I=1

    2 o

    3 o   o

    4 o   o   o

    .
    .
    .

    N o   o   o ...     o
```

(b)

```
      DO 100 J = 1,N
      DO 100 I = J+1,N
  100 A(I,J) = A(I,J) + B(I,J)
```

(c)

Figure 6.

## 2.3 Data Dependence in Triangular Loops

Most triangular loops in real programs, however, are not so easy to interchange. Most triangular loops refer to both the triangle and the diagonal, as in Figure 7(a), or to the triangle and its transpose, as in Figure 7(b). In these cases, the fact that the loop bounds are triangular must be taken into account when the data dependence graph is built. If the inner loop bound were replaced by N in these two loops, for instance, then there would be a data dependence cycle which would prevent the loops from being interchanged. With the upper bound of "I-1", there is no dependence cycle, since J is always less than I for any I. A harder example, from a kernel of a real program, is shown in Figure 8(a). To find the dependence from A(I,J) to A(K,J), the I and K indices must be considered together with the J index, since both I and K are triangular in J.

To handle cases like this, the KAP/Design includes an exact data dependence algorithm; this algorithm either proves that a data dependence relation exists or proves independence, if the following common conditions are satisfied:

1. Loop increments are constant,
2. Array subscripts are a linear expression involving loop index variables and constants, and
3. Loop upper and lower bounds are linear expressions involving outer loop index variables and constants, or are unknown.

When loop bounds are unknown, this test may compute a data dependence relation that does not exist for the loop bounds used in a particular execution of the loop, but the dependence can be proven to exist for some possible values of the loop bounds. The new data dependence algorithm is an extension of the test in [9]. Figure 8(b) shows all the ways to interchange the loop from Figure 8(a), as discovered using this perfect data dependence test.

## 2.4 Trapezoidal Loops Interchanging

The triangular loop bounds shown above actually appear quite often in numerical algorithms. Less frequently, other loop bounds which are functions of outer loops appear; some of these have little hope of loop interchanging, as the one in Figure 9. Others can be interchanged, but not with the simple rules for triangular loops. Such loop bounds may not appear in the original formulation of the algorithm, but may show up after some triangular loops have been interchanged. For instance, the loop in Figure 10(a) is coded with some simple triangular loop bounds.

```
      DO 100 I = 1,N
      DO 100 J = 1,I-1
100   A(I,J) = A(I,J) / A(I,I)
```

(a)

```
      DO 100 I = 1,N
      DO 100 J = 1,I-1
100   A(I,J) = A(J,I)
```

(b)

Figure 7.

```
      DO 100 J = 1,N
      DO 100 I = J+1,N
      DO 100 K = 1,J-1
100    A(I,J) = A(I,J) + A(I,K)*A(K,J)
```

(a)

```
      REAL A(N,N)
      INTEGER N,I,J,K
C
C
      DO 100 J = 1,N
      DO 100 I = J+1,N
      DO 100 K = 1,J-1
100    A(I,J) = A(I,J) + A(I,K)*A(K,J)
C
C
      DO 101 J=1,N
      DO 101 K=1,J-1
      DO 101 I=J+1,N
101    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 102 I=1,N
      DO 102 J=1,I-1
      DO 102 K=1,J-1
102    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 103 I=1,N
      DO 103 K=1,I-2
      DO 103 J=K+1,I-1
103    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 104 K=1,N-1
      DO 104 J=K+1,N
      DO 104 I=J+1,N
104    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 105 K=1,N-1
      DO 105 I=K+1,N
      DO 105 J=K+1,I-1
105    A(I,J) = A(I,J) + A(I,K) * A(K,J)
      END
```

(b)

Figure 8.

After interchanging the DO K and DO J loops, the bounds are modified as shown in Figure 10(b); the inner DO I loop, which was originally triangular with respect to the DO K loop, no longer is. The DO K loop bounds were changed when it was interchanged with DO J. Even through DO I and DO K do not form a triangular loop nest, the loop bounds can be modified to allow the loops to be interchanged; the result is shown in Figure 10(c). Note the use of the MIN function in the upper bound of DO K (had the lower bound needed to be changed, a MAX function would have been used). The iteration space described by the inner two loops of Figure 10(b) (assume that J is fixed) is shown in Figure 10(d); this is a trapezoid. In order to interchange trapezoidal loop bounds, the same type of loop bound modifications as used in triangular loop interchanging is used, except that a MIN or MAX function is used to cut off the point of the triangle and make the trapezoid. The KAP/Design includes triangular loop bound interchanging and trapezoidal loop interchanging; the KAP/Design output for the loop in Figure 10(a) is shown in Figure 11.

```
      DO 100 I = 1,N
      DO 100 J = 1,LP(I)**2
  100  A(I) = A(I) + B(I,J)
```

Figure 9.

```
      DO 100 K = 1,N
      DO 100 J = K+1,N
      DO 100 I = K+1,N
  100  A(I,J) = A(I,J) + A(I,K)*A(K,J)
```

(a)

```
      DO 100 J=1,N
      DO 100 K=1,J-1
      DO 100 I=K+1,N
  100  A(I,J) = A(I,J) + A(I,K) * A(K,J)
```

(b)

```
      DO 103 J=1,N
      DO 103 I=2,N
      DO 103 K=1,MIN (J-1, I-1)
  100  A(I,J) = A(I,J) + A(I,K) * A(K,J)
```

(c)

```
         I=
      1  2  3 ... J-1 J J+1 ... N

K=1   o  o  o ...  o  o  o  ... o

  2      o  o ...  o  o  o  ... o

  3         o ...  o  o  o  ... o
  .
  .
  .

J-1                   o  o  ... o
```

(d)

Figure 10.

3. Imperfectly Nested DO Loops

When imperfectly nested DO loops are to be interchanged, such as the loops in Figure 12(a), the usual process is to distribute the outer loop as in Figure 12(b); this creates perfectly nested DO loops, which can then be interchanged normally, as in Figure 12(c). The iteration space of the original loop can be drawn as shown in Figure 12(d); notice that there are two loop bodies that are important: statement s1, which is enclosed only in the DO I loop, and statement s2, which is enclosed in both loops. The execution order of the iterations is shown. Distributing and interchanging the loops as described modifies the execution ordering to the one in Figure 12(e); instead of execution flowing from left

```
      REAL A(N,N)
      INTEGER N,I,J,K
C
C
      DO 100 K = 1,N
      DO 100 J = K+1,N
      DO 100 I = K+1,N
100    A(I,J) = A(I,J) + A(I,K)*A(K,J)
C
C
      DO 101 K=1,N
      DO 101 I=K+1,N
      DO 101 J=K+1,N
101    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 102 J=1,N
      DO 102 K=1,J-1
      DO 102 I=K+1,N
102    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 103 J=1,N
      DO 103 I=2,N
      DO 103 K=1,MIN (J-1, I-1)
103    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 104 I=1,N
      DO 104 K=1,I-1
      DO 104 J=K+1,N
104    A(I,J) = A(I,J) + A(I,K) * A(K,J)
C
C
      DO 105 I=1,N
      DO 105 J=2,N
      DO 105 K=1,MIN (I-1, J-1)
105    A(I,J) = A(I,J) + A(I,K) * A(K,J)
      END
```

Figure 11.

to right, then top to bottom, it flows from top to bottom, then left to right. This is exactly the same as the modification to the execution ordering for interchanging of perfectly nested loops, as was shown in Figures 2 and 3.

Sometimes, however, the loop distribution that is required by this method is not legal. Take, for example, the loop in Figure 13(a); the value of A(I,K) used in s2 is assigned is sl, and the value of A(I-1,K+1) used in sl is assigned on the previous I iteration in sl. The iteration space dependence graph for this loop is shown in Figure 13(b); distributing the outer DO I loop would violate the data dependence conditions, since all the iterations of sl would be executed before any iterations of s2, changing the value used for A(I-1,K+1). These two loops can be interchanged, however; if the execution of statement sl could somehow be moved so it ran along the DO J axis, then the iteration space dependence graph would be as shown in Figure 13(c). In this iteration space, the DO I loop would have to be the inner loop; this corresponds to the program in Figure 13(d). Notice that not only have the DO loops been interchanged, but the references to the I loop index in sl have been replaced by J.

This type of loop interchanging is legal when no data dependence relations are violated. The data dependence relations that are preserved are:

(a) sl(i)   --> sl(i')      where i<=i'

(b) s2(i,j) --> s2(i',j') where i<=i', j<=j'

(c) sl(i)   --> s2(i',j') where i<=i', i<=j'

(d) s2(i,j) --> sl(i')      where i<i', j<i'

```
         DO 100 I = 1,N
sl:       X(I) = X(I) + 1.
         DO 100 J = 1,N
s2:  100  B(I,J) = X(I) + A(I,J)
```

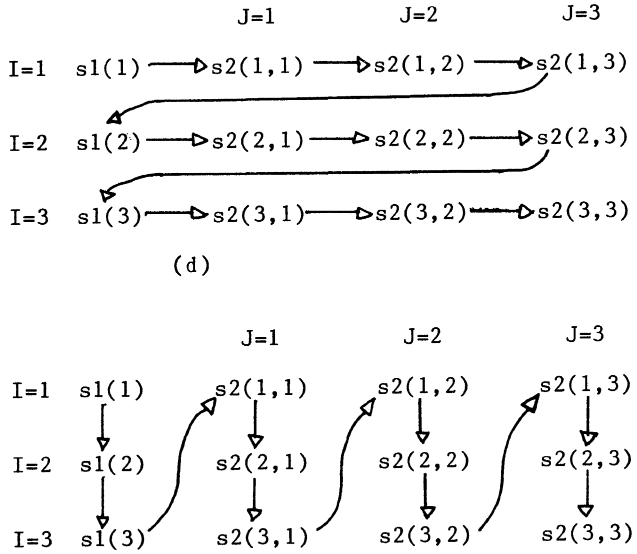
(a)

```
         DO 101 I = 1,N
sl:  101  X(I) = X(I) + 1.
         DO 100 I = 1,N
         DO 100 J = 1,N
s2:  100  B(I,J) = X(I) + A(I,J)
```

(b)

```
         DO 101 I = 1,N
sl:  101  X(I) = X(I) + 1.
         DO 100 J = 1,N
         DO 100 I = 1,N
s2:  100  B(I,J) = X(I) + A(I,J)
```

(c)



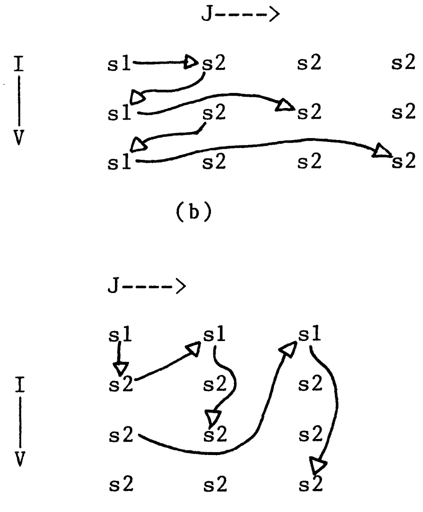
(d)

(e)

Figure 12.

```
         DO 100 I = K+1,N
sl:       A(I,K) = A(I-1,K+1) * A(K+1,K+1)
         DO 100 J = K+1,N
s2:  100  A(I,J) = A(I,J) + A(I,K)*A(K,J)
```

(a)



(b)

(c)

```
         DO 100 J = K+1,N
sl:       A(J,K) = A(J-1,K+1) * A(K+1,K+1)
         DO 100 I = K+1,N
s2:  100  A(I,J) = A(I,J) + A(I,K)*A(K,J)
```

(d)

Figure 13.

Test (a) above is obvious, and test (b) is the same as the test for normal interchanging.

These conditions divide the iteration space into two types of regions, as shown in Figures 14(a) and 14(b). These regions are subsets of each other: R1(2) includes R1(1) and is a subset of R1(3), and so on; for the R2 regions, R2(2) includes R2(3) and is a subset of R2(1), and so on. This new type of imperfectly-nested loop interchanging will preserve dependences in the iteration space dependence graph from s1(k) to any iteration of s2 in region R2(k), and dependences to s1(k) from any iteration of s2 in region R1(k-1). Figures 15(a) and 15(b) shows data dependences in the iteration space that will be preserved by this type of loop interchanging, and Figures 15(c) and 15(d) shows data dependences that will be violated.

Testing for legality of this type of loop interchanging is easier when the loop bounds are triangular, as in Figure 16(a); the iteration space for this loop is shown in Figure 16(b). Since the relation I < J is always true for any iteration of s2 (since the DO J loop starts at I+1), the interchange conditions (c) and (d) above becomes

(c') s1(i) --> s2(i',j') where i<=i'
(since i'<j', i<=i' implies i<=j')

(d') s2(i,j) --> s1(i') where j< i'
(since i<j, j<i' implies i<i')

```
               J---->

             R1(1)| R1(2)| R1(3)| R1(4)...
I      s1     s2  |  s2  |  s2  |
|   --------------+      |      |
|      s1     s2     s2  |  s2  |
V   ---------------------+      |
       s1     s2     s2     s2  |
    ----------------------------+
```

(a)

```
               J---->

           +--------------------------
I      s1  |  s2     s2     s2
|          |      +-------------------
|      s1  |  s2  |  s2     s2
V          |      |      +------------
       s1  |  s2  |  s2  |  s2
           |      |      |      +-----
           | R2(1)| R2(2)| R2(3)| R2(4)...
```

(b)

Figure 14.

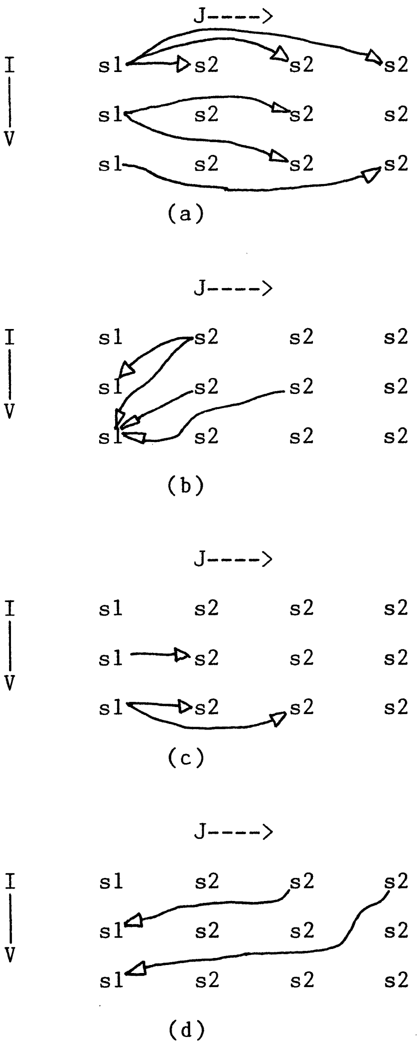


Figure 15.

```
              DO 100 I = 1,N
s1:            .....
              DO 100 J = I+1,N
s2:     100    .....
```

(a)

```
              J---->

I      s1     s2     s2     s2     s2
|      s1            s2     s2     s2
V      s1                   s2     s2
       s1                          s2
```

(b)

Figure 16.

A future version of KAP/Design will include interchanging of imperfectly nested DO loops.

In conclusion, loop interchanging is an interesting subject for its own sake. Loop interchanging has many applications in optimizing compilers for high speed computers, and further studies in this field will certainly prove useful in the long run.

## References

[1] Wolfe, M. J., "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 82-1009, Oct. 1982.

[2] Allen, J. R. and K. Kennedy, "Automatic Loop Interchange," in Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction, pp. 233-246, June, 1984.

[3] Abu-Sufah, W. A., "Improving the Performance of Virtual Memory Computers," Ph.D. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 78-945, Nov. 1978.

[4] W. A. Abu-Sufah, D. J. Kuck, D. H. Lawrie "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," IEEE Trans. on Computers, Vol. C-30, No. 5, pp. 341-356, May 1981.

[5] Huson, C. et al, "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," Proc. of the 1986 International Conference on Parallel Processing, Aug., 1986.

[6] Macke, T. et al, "The KAP/ST-100: A Fortran Translator for the ST-100 Attached Processor," Proc. of the 1986 International Conference on Parallel Processing, Aug., 1986.

[7] Davies, D. et al, "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," Proc. of the 1986 International Conference on Parallel Processing, Aug., 1986.

[8] Dongarra, J. J., F. G. Gustavson and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," in SIAM Review, Vol. 26, No. 1, pp. 91-112, Jan. 1984.

[9] Banerjee, U., "Speedup of Ordinary Programs," Ph.D. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 79-989, Oct. 1979.

[10] Banerjee, U., "Data Dependence in Ordinary Programs," M.S. Thesis, Univ. of Ill. at Urb.-Champ., Dept. of Comp. Sci. Rpt. No. 76-837, Nov. 1976.

[11] Banerjee, U., S. C. Chen, D. J. Kuck and R. A. Towle "Time and Parallel Processor Bounds for Fortran-Like Loops", IEEE Trans. on Computers, Vol. C-28, No. 9, pp. 660-670, Sep. 1979.

[12] Allen, J. R., K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report COMP TR84-9, Rice University, Houston, July, 1984.

[13] Kuck, D., The Structure of Computers and Computations, Vol. I, John Wiley and Sons, Inc., New York, NY, 1978.

# COMPILER GENERATED SYNCHRONIZATION FOR DO LOOPS

*Samuel P. Midkiff and David A. Padua*

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- This paper presents methods for the compile time generation of synchronization instructions for parallel loops running on a multiprocessor. A synchronization instruction set and architecture are defined, and using these it is shown how to synchronize FORTRAN source loops so that dependences on the parallel loop are enforced. Construction of an applicable dependence graph is covered. A program transformation is given that increases the parallelism that can be utilized in a parallel loop containing nested loops. Finally, a generalizable technique is presented for reducing the number of dependences needing synchronizing in a parallel loop.

## Introduction

When programs are executed on a multiprocessing computer, parallelism can be supported at several levels. The University of Illinois Cedar Project machine [7], the CRAY XMP series [3] (using micro-tasking) and the NYU Ultra-Computer [6] among others allow intra-loop parallelism, i.e. parallelism that exists between different iterations of a loop, to be exploited. This parallelism is realized by allowing different iterations of a loop to run simultaneously on different processors of the multiprocessor. When one iteration of a loop produces data that another iteration uses, or there is a danger of an iteration overwriting data prematurely, it is necessary to constrain the execution order of iterations so that the hazards are avoided. In this paper, we show how this can be done automatically, at compile time, using synchronization instructions. Strategies are presented for the placement of synchronization instructions in a FORTRAN source program loop, to increase the amount of parallelism realized from the source program, and to reduce the amount of synchronization necessary.

It is assumed that programs that are synchronized by the methods presented here have been passed through an automatic program restructurer such as Parafrase [17], [11] and [10]. We do not discuss parallel loop detection, but rather the synchronization of dependences in loops that have previously been recognized as parallel.

Although some of the transformations and optimizations mentioned in this paper are applicable to a wide range of architectures, for brevity's sake we restrict ourselves to a single architecture.

The target architecture, and the sorts of parallelism that can be exploited by it are the first topic of this paper. We then turn to a discussion of *dependences* and *dependence graphs* (DG), which form the basis of the actions of this paper. Since we are concerned only with synchronization within a loop, a DG will be built in turn for each loop to be synchronized in the program. The loop is then transformed, and the next loop in the program is dealt with until the entire program is transformed.

Next we define the synchronization instruction set used, and show how it can be used to synchronize dependences. Because of the semantics of the synchronization instructions, the DG can now be modified to reduce the amount of parallelism in inner loops lost to the synchronization scheme.

After nested loops have been dealt with, the DG is appropriate to guide the generation of synchronization instructions, and we give an example of a loop nest with synchronization instructions added. We can choose, however, to remove from the DG redundant dependence arcs, thereby reducing the amount of synchronization needed. We give a method to reduce the number of dependence arcs, and by examples show its effects on synchronization.

Although we use a single synchronization instruction set in this paper, the methods used can be extended to other instruction sets. In particular, the method presented in this paper for removing redundant dependence arcs is extensible to any architecture.

## Architecture

The architecture of this paper is a multiprocessor. Figure 1 shows a block diagram of the target architecture. Neither processor nor memory modules are distinguishable from their neighbors.

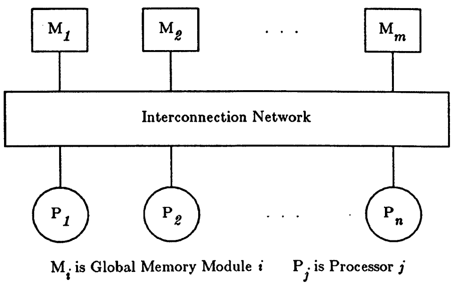


Figure 1. System Architecture

Each memory module is capable of being accessed by any processor through the interconnection network. Access is assumed to be deterministic in that either fetches and stores from a single processor are processed in the order they are issued, This means that either fetches and stores within a processor are done serially, or that an equivalent ordering is imposed on data accesses.

Parallelism is realized by spreading iterations of a loop across multiple processors. A loop so executed will be referred to as a *dospread* loop. All other loops will be referred to as DO loops. A *dospread* loop can be either a *doall* [5], [13], [15], or a *doacross* loop [4], [15], [16]. We place the following restriction on *dospread* loops: they may contain no nested dospread loop. In [14] and [16] algorithms are given for deciding what loop in a nest of potential *dospread* loops should be run as a *dospread* loop.

*Dospread* loop iterations are either spread horizontally [8] across processors or self scheduled. Horizontal spreading means that given $P$ processors, processor $p$ will execute iterations $p, p+P, p+2 \cdot P, \cdots$. Figure 2 shows an example of horizontal spreading on a loop with sixteen iterations. Horizontal scheduling can be accomplished through loop blocking [8]. Self scheduling means that whenever a processor becomes idle it will execute the next iteration of the *dospread* loop.

This work was supported in part by the National Science Foundation under Grant No. US NSF DCR84-10110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and the IBM Donation.

0190-3918/86/0000/0544 $01.00 © 1986 IEEE

| processor = | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| i = | 1 | 2 | 3 | 4 |
| | 5 | 6 | 7 | 8 |
| | 9 | 10 | 11 | 12 |
| | 13 | 14 | 15 | 16 |

Figure 2. Horizontal Spreading of *dospread* Iterations

## Dependences and Dependence Graphs

The restrictions on program execution mentioned previously are necessary to enforce dependences. We now informally define four types of dependences. For more formal definitions and discussion of dependence testing, see [2], [9] and [17]. In the definitions that follow, $S_i$ refers to a statement in the program. If there are two statements, $S_i$ and $S_j$, and $i < j$, then $S_i$ lexically precedes $S_j$.

i *flow* dependence: If data fetched by $S_{si}$ may be generated by $S_{so}$, then a flow dependence exists from $S_{so}$ to $S_{si}$.

ii *output* dependence: If $S_{si}$ may write to some memory location after $S_{so}$ has written to that same memory location, then an output dependence exists from $S_{so}$ to $S_{si}$.

iii *anti* dependence: If $S_{so}$ fetches from a memory location that is may subsequently be overwritten by $S_{si}$, then $S_{si}$ is anti dependent on $S_{so}$.

iv *control* dependence: If whether $S_{si}$ executes depends on the outcome of the execution of $S_{so}$, then a control dependence exists from $S_{so}$ to $S_{si}$.

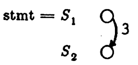
In each of the four cases, $S_{so}$ is called the *source* of the dependence and $S_{si}$ the sink. If the sink of a dependence is after the source, i.e. $so < si$, the dependence is a *lexically forward dependence* (LFD). Otherwise the dependence is a *lexically backward dependence* (LBD).

Dependences extend over zero or more iterations. The number of iterations a dependence extends across is known as the dependence *distance*, $\Delta$ [17]. Figure 0a shows a loop nest. Values of $A$, in $S_1$, written in an iteration are read three iterations later by $S_2$. Therefore a flow dependence exists from $S_1$ to $S_2$, and $\Delta = 3$.

A dependence can be thought of as a relation between two statements. This relation can be represented graphically on a DG, where nodes are statements and a directed arc from $S_{so}$ to $S_{si}$ represents a dependence. The arcs are labeled with the value of $\Delta$ for the dependence. Nested DO loops are considered to be a single statement, and are represented by a single node in the DG. Control cycles caused by backwards GOTOs are also represented by a single node in the DG. Figure 3b gives a DG for the loop of Figure 3a. If two or more dependences, all with the same source and sink, but with different distances, exist, we can remove all the dependence arcs except for the one with the minimum distance. The reason for this is that all arcs but the one with the shortest distance are redundant when using the synchronization instructions of this paper. An example showing this is given in the section Dependence Arc Elimination. An arc is considered redundant if the dependence it enforces will be honored even if it is not explicitly synchronized. What arcs are redundant is dependent on the synchronization instruction set and control structure of the architecture.

```
        DOSPREAD 10 I = 1, N
                A(I) = B(I) + C(I)      S1
                D(I) = A(I-3) + E(I)    S2
10      CONTINUE
```

(a) A Loop Nest



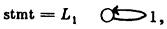
(b) DG for the Loop of (a)

Figure 3. A Loop and Its DG

Dependences can have $\Delta$s that are greater than zero, less than zero, and equal to zero. If $\Delta > 0$, as in Figure 3b, the dependence is from an earlier iteration to a later iteration. If $\Delta = 0$, then the dependence is from the same iteration to a later one. If $\Delta < 0$, the dependence is from a later iteration to an earlier one. This can only happen if the dependence is on a loop nested within another loop. Figure 4a shows such a loop nest, and Figure 4b shows the DG. There are two distances attached to the dependence arc. The first, 1, is the distance of the dependence on the outer $I$ loop and the second, -2, is the distance on the inner $J$ loop.

```
        DOSPREAD 20 I = 1, N
                DO 10 J = 1, N                          L1
                        A(I,J) = B(I,J) + C(I,J)
                        D(I,J) = A(I-1, J+2) - E(I,J)
10              CONTINUE
20      CONTINUE
```

(a) A Loop Nest



(b) The DG for (a)

Figure 4. A Loop Nest with >0 and <0 Dependences

The DG is the major data structure for determining the location of synchronization instructions in a program. For our architecture, when the dependence graph is built only dependences with $\Delta > 0$ are included. $\Delta$ less than zero has to be considered only if two nested loops are to execute as *dospread* loops. Since *dospread* loops cannot be nested, these types of dependences will not need to be synchronized. If $\Delta = 0$, then the dependence is within an iteration. Since data stores and fetches are serial within an iteration, these dependences are forced to be honored by the hardware, and need not be synchronized.

All control dependences have $\Delta = 0$, except for control dependences caused by branches out of the loop, called *exit-if*s. If an exit-if occurs, the loop terminates. Therefore, in loops with an exit-if, each statement of an iteration is dependent on the execution of the exit-if in the previous iteration. Thus a control dependence exists between the exit-if and every statement in the graph. Every dependence arc but the one from the exit-if to the first statement of the loop body is redundant and can be removed. Therefore we only include this dependence arc in the DG. Which exit-if dependence arcs are redundant is strongly dependent on the the control structure of the architecture, so adding only one control arc for an exit-if is not generally valid. Figure 5a shows a loop nest that will be used throughout this paper as an example, and Figure 5b the DG for the loop. Note that a data dependence with distance zero exists from $S_4$ to $S_5$ and is not included on the DG.

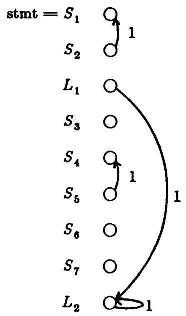
## Synchronization Instructions

Three synchronization instructions are needed to synchronize *dospread* loops. We first give the semantics of these instructions, and then show how they are used to synchronize loops. All three instructions operate on a fixed set of synchronization registers accessible to all processors. The synchronization instructions assume that the DO loops being synchronized are normalized [2], i.e. they run from 1 to some upper bound in increments of 1. This can be done automatically.

The *testset* instruction has the syntax *testset*$(r)$, where $r$ is a synchronization register. It performs two functions. The first is iteration. Having done so, it alters the value of *r* to signal later iterations it has executed. Figure 6a gives the semantics of the ***testset*** statement. One important feature of the ***testset*** instruction is that all executions of a ***testset*** after the first, for some synchronization register ***r***, within an iteration, act as no-ops.

```
          DOSPREAD 50 I = 1, N
                  S(I) = S(I) + 1.0                        S1
                  IF (S(I).GT.0.0) GOTO 60                 S2
                  DO 10 J = 1, N                           L1
                          A(I,J) = B(I,J) + C(I,J)
10                CONTINUE
                  IF (D(I).GT.0.0) GOTO 20                 S3
                          E(I) = F(I-1)+G(I)               S4
                          F(I) = E(I) + H(I)               S5
20                CONTINUE                                 S6
                          P(I) = 2.0*P(I)                  S7
                  DO 40 M = 1, 8                           L2
                          Q(I,M) = R(I,M) + A(I-1,M)
                          R(I,M) = Q(I-1,M+2)*2.0
40                CONTINUE
50        CONTINUE
60        CONTINUE
```

(a) A Loop Nest



(b) DG for the Loop Nest of (a)

Figure 5. A Loop Nest and Its DG

```
l:      if $r = i - 1$ then
                $r := i$
        else if $r < i - 1$ then
                goto $l$
```

(a) Semantics of the *testset* Instruction

```
        if $i - \Delta > 0$ then
l:      if $r < i - \Delta$ then  goto $l$
```

(b) Semantics of the *test* Instruction

```
l:      if  all iterations $j$, $j < i$, have completed  then
                halt execution of all other processors executing the loop
        else
                goto  $l$
```

(c) Semantics of the *terminate* Instruction

Figure 6. Semantics of the *testset* and *testset* Instructions

The *test* instruction has the syntax $test(r)\ \Delta$. The *test* instruction in an iteration waits until a *testset* instruction has executed $\Delta$ iterations previously.

The final synchronization instruction is the *terminate* instruction. Its syntax is simply *terminate*. Its purpose is to halt execution of a *dospread* loop when an iteration branches out of the loop during execution of an exit-if. The *terminate* waits until all previous iterations of the loop have finished executing, and then halts the execution of the loop. Figure 6c summarizes its semantics.

This instruction set involves several tradeoffs. The primary advantage of the instruction set, as discussed when building the DG, is that dependence distances can safely be assumed to be any distance less than or equal to the minimum distance of a dependence. Therefore, one can always be assumed as the distance of a dependence. This is important for two reasons. First, it allows simpler dependence testing routines to be used. Secondly, in cases where the dependence distance is unknowable at compile time, as with sub-scripted subscripts, the dependence can still be safely synchronized by assuming the dependence distance is one, the minimum distance that the dependence can take.

One disadvantage is that the *testset* instruction partially serializes the loop since each *testset* must wait until a *testset* in the previous iteration has executed. Therefore no loop can execute completely in parallel. This serializing effect, however, has an important advantage. It increases the number of dependence arcs that are redundant. In doing so it allows the aforementioned assumptions to be made about the distance of a dependence.

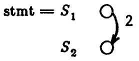
Another disadvantage is the use of a fixed number of registers. If a loop has more dependences to synchronize than there are registers available, we can be forced to resort to dependence folding [14] to reduce the number of dependence arcs present. Dependence folding usually leads to loops being more serial. Only having a fixed number of registers also causes problems with nested loops, which are discussed later in this section. The advantage of a fixed number of registers is that they can be accessed faster than global memory, and special hardware can be set up, if desired, for accessing these registers, thereby taking traffic off the interconnection network.

We now show how to synchronize LFDs and LBDs using the above synchronization instructions when no nested loops are present in the body of the loop. With any dependence it is necessary that executing the source of the dependence notify the sink, in a later iteration, that it can execute. It is also necessary that the sink of a dependence wait until notified that its source has executed. With an LFD this is simple. If a *testset* is placed after the source of the dependence, and before the sink of the dependence, the dependence will be enforced.

For example, consider the loop in Figure 7a, and its DG in Figure 7b. The dependence from $S_1$ to $S_2$ extends over two iterations. Figure 7c shows a *testset* placed between $S_1$ and $S_2$. Before the sink of a dependence executes in iteration $i + 2$ the *testset* preceding it must complete execution. The *testset* in iteration $i + 2$ cannot execute before the *testset* in iteration $i + 1$, which, in turn, cannot execute before the *testset* in iteration $i$. The *testset* in iteration $i$ cannot execute, however, until the source of the dependence in iteration $i$ has executed. Thus the dependence is forced to be honored.

```
          DO 10 I = 1, N
                  A(I) = B(I) + C(I)         S1
                  D(I) = E(I) + A(I-2)       S2
10        CONTINUE
```

(a) A Loop Nest



(b) The DDG for (a)

```
          DO 10 I = 1, N
                  A(I) = B(I) + C(I)         S1
                  testset(1)                 TS1
                  D(I) = E(I) + A(I-2)       S2
10        CONTINUE
```

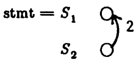
(c) The Loop Nest of (a) Synchronized

Figure 7. Synchronizing a LFD

To synchronize a LBD, such as is seen in Figure 8a, a *testset* is placed after the source of the dependence to signal that the source has executed in this iteration. A *test* is then placed before the sink of the dependence. Thus the sink of the dependence cannot execute until after the source, since the *test* instruction will not finish executing until after the *testset* following the source has completed. Figure 8c shows the synchronized loop.

```
        DO 10 I = 1, N
                A(I) = B(I) + D(I-2)    $S_1$
                D(I) = E(I) + A(I)      $S_2$
10      CONTINUE
```

(a) A Loop Nest



(b) The DDG for (a)

```
        DO 10 I = 1, N
                test(1) 2               $T_1$
                A(I) = B(I) + D(I-2)    $S_1$
                D(I) = E(I) + A(I)      $S_2$
                testset(1)              $TS_1$
10      CONTINUE
```

(c) The Loop Nest of (a) Synchronized

Figure 8. Synchronizing a LBD

If control branches are present in the *dospread* loop further actions must be taken. In particular, we must insure that a *testset* be executed for every register used in the loop, on every iteration of the loop. Consider again the loop nest of Figure 5. To synchronize the LBD from $S_5$ to $S_4$ a *testset* would be placed between $S_5$ and $S_6$. If the IF statement at $S_3$ takes the true branch, then the *testset* would not be executed on that iteration. The *testset* of the next iteration in which the false branch of the IF is taken would then deadlock waiting for its synchronization register to be updated. Therefore a *testset* should be placed along the flow of control path through the loop containing the true branch.

This problem is handled as follows. The loop is broken up into basic blocks, (BAS) [1], and a flow of control graph [1] is built that represents the control structure of the program. This graph can be represented by a matrix $reach(i, j)$, such that $reach(i, j) =$ **true** if an arc from $BAS_i$ to $BAS_j$ is in the flow of control graph. The $(i, j)$ element of the transitive closure of *reach*, $reach^*(i, j)$, is true if a path exists from $BAS_i$ to $BAS_j$.

Each arc in the DG is handled in turn. A *testset*, $TS_{so}$, is placed after the source of the dependence. At each branch, *reach* can be checked to see if any target of the branch is the block containing $TS_{so}$. If this is so, every other target BAS of the branch is checked (using $reach^*$) to determine if it can also reach the block containing $TS_{so}$. If it cannot, a *testset* is placed in that BAS. Thus every branch that can reach $TS_{so}$ must either reach the original *testset* or an added *testset*, and deadlock cannot occur.

Two other flow of control problems remain: the presence of nested DO loops within the *dospread* loop, and exit-ifs. The former case is handled in the next section. We deal with the latter case now.

Consider the exit-if of statement $S_2$ in Figure 5. The control arc from $S_2$ to $S_1$ forms a LBD that will cause *test*(1) 1 to be placed before $S_1$, and *testset*(1) to be placed after $S_2$. The *test* instruction will wait until the *testset* in the previous iteration has executed before allowing $S_1$ to execute, and the *testset* will not execute if the exit branch of the exit-if is taken. If the exit branch is taken in iteration $i$, all iterations less than $i$ must be allowed to finish, and the deadlocked iterations greater than $i$ must be terminated.

To halt the loop, a block of code is added to the loop that executes a *terminate* instruction. As will be seen, placement of this block is unimportant but in this paper it will be added to the end of the loop. For every exit branch to some label $l$, the block:

```
$l'$      terminate
          GOTO $l$
```

is added to the loop, and all branches to $l$ in the loop are changed to branches to $l'$. The *terminate* instruction checks if all iterations of the loop less than the current one have finished. If so, it halts all processors executing the loop and the GOTO makes the exit branch. To prevent other statements in the loop from falling through to this added block, a GOTO to the CONTINUE at the end of the *dospread* loop is added immediately before the *terminate* instruction. Figure 9 shows the program with these statements added.

```
        DOSPREAD 50 I = 1, N
                test(1) 1                               $T_1$
                S(I) = S(I) + 1.0                       $S_1$
                IF (S(I).GT.0.0) GOTO 61                $S_2$
                testset(1)                              $TS_1$
                DO 10 J = 1, N                          $L_1$
                        A(I,J) = B(I,J) + C(I,J)
10              CONTINUE
                IF (D(I).GT.0.0) GOTO 20                $S_3$
                        E(I) = F(I-1)+G(I)              $S_4$
                        F(I) = E(I) + H(I)              $S_5$
20              CONTINUE                                $S_6$
                        P(I) = 2.0*P(I)                 $S_7$
                DO 40 M = 1, 8                          $L_2$
                        Q(I,M) = R(I,M) + A(I-1,M)
                        R(I,M) = Q(I-1,M+2)*2.0
40              CONTINUE
                GOTO 50
61              terminate
                GOTO 60
50      CONTINUE
        ...
60      CONTINUE
```

Figure 9. Loop Nest with Exit-if Synchronized

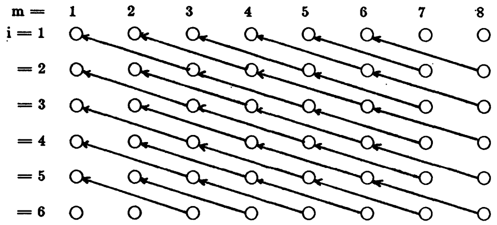
## Nested Loops

If the source and sink of a dependence are both within the same nested loop, the dependence forms a self cycle on the DG node representing the loop. $L_2$ in the DG of Figure 5b is an example of this. This loop is reproduced in Figure 10a If a *test* is placed before the loop, and a *testset* is placed after the loop, the loop forms a large chunk of serial code in the *dospread* loop. It is therefore desirable that some parallelism be extracted from this nested loop. Figure 10b gives the iteration space [17] for this loop nest. Each node in the iteration space represents an iteration of the loop, and arcs represent dependences from iteration to iteration. Dependences on the $M$ loop span two iterations of the $M$ loop, while dependences on the *dospread* loop span one iteration. Our goal is to allow some iterations of $L_2$ in iteration $i + 1$ of the *dospread* to execute before all iterations of $L_2$ execute in iteration $i$ by placing synchronization instructions so that dependences on both the *dospread* and $L_2$ loops are satisfied.

To do this a transformation called *loop splitting* is performed. Loop splitting breaks the inner loop into $n$ different sub-loops, and places synchronization instructions between the sub-loops. Each sub-loop executes a contiguous set of iterations from the original loop. Thus if the $M$ loop of our example is broken into four sub-loops, the first would run from 1 to 2, the second from 3 to 4, the third from 5 to 6 and the fourth from 7 to 8. Each sub-loop contains two iterations, therefore if the first sub-loop of iteration $i$ of the *dospread* loop waits until the second sub-loop in the previous iteration of the *dospread* loop has executed, dependences to it will be honored. That this is true can be seen by examining the sources of dependences in the iteration space diagram that have their sinks in the second sub-loop. The second sub-loop must wait until the third sub-loop of the previous iteration has finished, and the third sub-loop must wait until the fourth sub-loop of the previous iteration has finished. The fourth sub-loop contains no dependence sinks whose sources are in other sub-loops, therefore it does not need to wait on any other sub-loop. These dependences can be detected by splitting the inner loop and recomputing the dependences between the sub-loops. The synchronization of these dependences will give the execution ordering specified above.

```
      DOSPREAD 50 I = 1, N
         ...
         DO 10 J = 1, N                          L1
            A(I,J) = B(I,J) + C(I,J)
         ...
         DO 40 M = 1, 8                          L2
            Q(I,M) = R(I,M) + A(I-1,M)
            R(I,M) = Q(I-1,M+2)*2.0
40       CONTINUE
50    CONTINUE
```

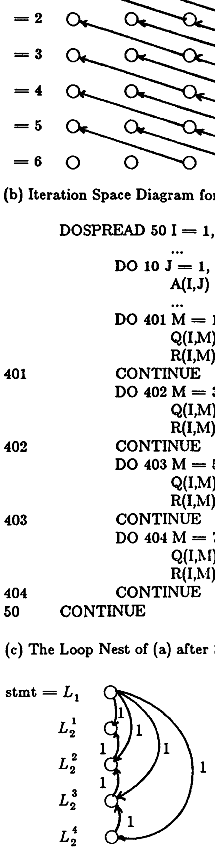
(a) A Loop Nest



(b) Iteration Space Diagram for (a)

```
      DOSPREAD 50 I = 1, N
         ...
         DO 10 J = 1, N                          L1
            A(I,J) = B(I,J) + C(I,J)
         ...
         DO 401 M = 1, 2                         L2^1
            Q(I,M) = R(I,M) + A(I-1,M)
            R(I,M) = Q(I-1,M+2)*2.0
401      CONTINUE
         DO 402 M = 3, 4                         L2^2
            Q(I,M) = R(I,M) + A(I-1,M)
            R(I,M) = Q(I-1,M+2)*2.0
402      CONTINUE
         DO 403 M = 5, 6                         L2^3
            Q(I,M) = R(I,M) + A(I-1,M)
            R(I,M) = Q(I-1,M+2)*2.0
403      CONTINUE
         DO 404 M = 7, 8                         L2^4
            Q(I,M) = R(I,M) + A(I-1,M)
            R(I,M) = Q(I-1,M+2)*2.0
404      CONTINUE
50    CONTINUE
```

(c) The Loop Nest of (a) after Splitting



(d) DG for the Split Loop

Figure 10. A Loop Splitting Example

When this synchronization has taken place, the execution of the third and first, and fourth and second sub-loops of adjacent iterations of the *dospread* loop can overlap, salvaging some of the parallelism present before loop spreading.

The only other problem that remains is what to do with dependences whose sources are outside the loop being split, but whose sink is in the loop being split, or whose sink is outside the loop being split but whose source is the CONTINUE statement of the loop being split. The dependence should be replicated over every copy of the loop created by loop splitting. The dependence routines used to build the original DG should then be queried to see if the dependences exist to or from each sub-loop of the split loop. If the dependence no longer exists to or from a particular sub-loop, it should be removed from the DG. In our example, the dependence does exist to every sub-loop.

Figure 10c shows the $M$ loop split, and Figure 10d shows the DG for the loop nest with dependences resulting from loop splitting added.

In [14] a method is given for determining analytically the placement of dependences for split loops as a function of the maximum dependence distance on the split loop and the minimum number of iterations in a sub-loop.

## Dependence Arc Elimination

After nested loops have been taken care of, as in the previous section, synchronization instructions can be inserted into the source program using the methods of the second section. Adding synchronization instructions now can result in more instructions being added than is necessary to insure the legal execution of the program. We now discuss a method of reducing the number of dependences needing synchronization in a DG. As the number of dependences needing synchronization is reduced, the amount of synchronization added to the loop is reduced, and consequently the overhead associated with synchronization is reduced. The goal of dependence arc elimination is to determine what dependences are redundant, i.e. what dependences are implicitly synchronized by a combination of the control structure of the architecture and the synchronization of other dependences. Related work in this area is discussed in [12].

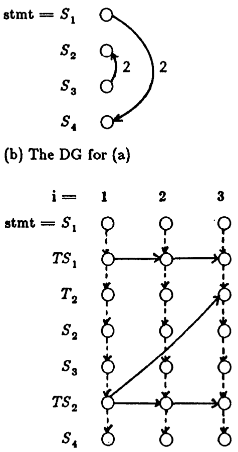
We present our dependence arc elimination method with the instruction set defined in this paper in mind. The technique is usable, however, with a wide range of architectures. Details of this are given in [14].

A dependence arc can be eliminated if a combination of the control structure of the target architecture, and synchronization added for other dependences, force the dependence to always be synchronized. To eliminate a dependence arc, a simple way of representing the total control structure of a loop nest is required. We use a *controlled path graph*, or CPG, to give this representation. A CPG contains nodes, each of which represents a *statement instance*. A statement instance is the occurrence of a statement within an iteration of the *dospread* loop. At least a source and sink of any dependence arc to be eliminated must be included in the CPG. Thus, if $\Delta_{max}$ is the longest distance of any dependence in the *dospread* loop, the CPG must contain at least $\Delta_{max} + 1$ iterations (columns). That no more columns than this are needed is proven in [14]. Arcs in the CPG represent the execution order of the statement instances: the statement instance at the head of an arc will execute after the statement instance at the tail of an arc.

Figure 11a gives a *dospread* loop without branches and with synchronization instructions added. Figure 11b gives the DG for the loop that led to this synchronization, and Figure 11c gives the CPG for this loop. The dashed lines in the CPG represent control exercised over the program execution by the architecture. As stated before, statements within a single iteration execute serially on a single processor. Each statement within an iteration must execute after the previous statement in that iteration, and therefore is at the head of an arc whose tail is the previous statement. Next, we know a *testset* on a certain register must follow the *testset* on that same register in the previous iteration. Therefore an arc is drawn from each *testset* in the first iteration to the *testset* in the second iteration using the same synchronization register. Finally, we know that a *test*($r$) $\Delta$ instruction must follow the *testset* instruction $\Delta$ iterations back. Therefore we draw arcs from each *testset* to its corresponding *test*, if one exists.

```
        DOSPREAD 10 I = 1, N
            A(I) = B(I) + C(I)          S1
            testset(1)                  TS1
            test(2) 2                   T2
            B(I) = A(I) + E(I-2)        S2
            E(I) = D(I) + F(I)          S3
            testset(2)                  TS2
            F(I) = A(I-2) + E(I)        S4
10      CONTINUE
```

(a) A DOSPREAD Loop



(b) The DG for (a)

(c) The CPG for the (a)

Figure 11. Example of a CPG

Constructing the CPG is the architecturally dependent portion of this method. If it is desirable to do dependence arc elimination with another architecture, all that is necessary is that the dashed arcs accurately reflect the control exercised by the target architecture, and that arcs are added to reflect the effects of synchronizing the loop.

Deciding if an arc should be removed from the DG is trivial once the CPG has been built. We need only to find a path from the source to the sink of the dependence in the CPG without using any arcs in the CPG that result from synchronizing that dependence. It is possible to eliminate the dependence arc since if a path exists through the CPG from the source to the sink of the eliminated dependence, then the sink of the dependence must execute after the source of the dependence without explicitly synchronizing the dependence.

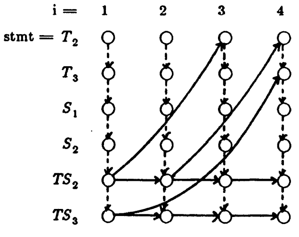
The dependence arc from $S_1$ to $S_4$ can be removed from the DG for this loop. Starting at $S_1$ in iteration one, $TS_2$ in the same iteration can be reached. Its arcs can be followed to $TS_2$ in iteration 3, and from there the dotted lines can be followed to $TS_4$ in iteration 3. Therefore we can travel from the source to the sink of the dependence without going through any arcs resulting from $TS_1$, which synchronize the dependence whose arc is being eliminated.

After deciding that an arc can be removed from the DG, we do so. As well, all nodes in the CPG used to synchronize the arc are removed, and the arcs associated with them are removed. This insures that the dependence arc just removed is not used to eliminate any other dependence arc.

We give another example that shows that only the dependence arc with the minimum distance needs to be retained when two or more dependence arcs with different distances exist between two statements. Figure 12a gives a loop nest with both dependences from $S_2$ to $S_1$ synchronized. Figure 12b gives the CPG for this loop nest. The *testset* and *set* on register 2 synchronizes the dependence of length two, while the *testset* and *test* on register 3 synchronizes the dependence of length three. Starting at $S_2$ in iteration one, the dashed arcs can be traveled to reach $TS_2$. The arc leaving $TS_2$ in iteration one, and traveling to $TS_2$ in iteration two is taken, followed by the arc to $T_2$ in iteration three. From there the dashed arcs can be taken to $S_1$ in iteration three, showing that the dependence arc with distance three can be eliminated.

```
        DO 10 I = 1, N
            test(2) 2                   T2
            test(3) 3                   T3
            A(I) = B(I-1) + B(I-3)      S1
            B(I) = A(I) * 2.0           S2
            testset(2)                  TS2
            testset(3)                  TS3
            ...
10      CONTINUE
```

(a) A Loop Nest



(b) The CPG for (a)

Figure 12. Dependence Arc Elimination Example: Redundancy of Longer Dependence

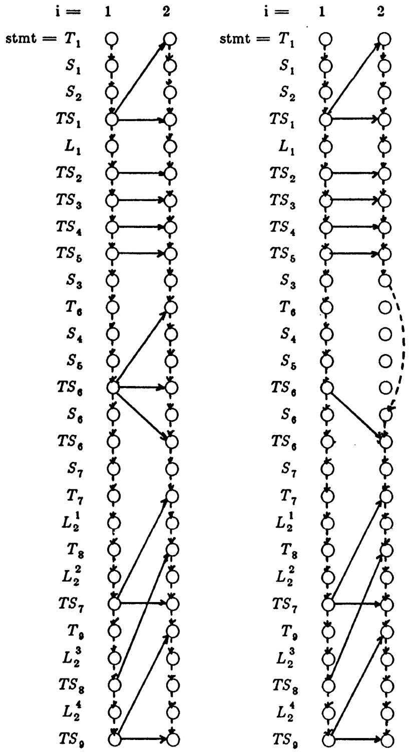
Figure 13a shows the program of Figure 5a with synchronization statements added. Figure 13b and Figure 13c show the four possible CPGs for this program. Four CPGs are needed to express the possible combinations of control paths taken on

```
        DOSPREAD 50 I = 1, N
            test(1) 1                           T1
            S(I) = S(I) + 1.0                   S1
            IF (S(I).GT0.0) GOTO 61             S2
            testset(1)                          TS1
            DO 10 J = 1, N                      L1
              A(I,J) = B(I,J) + C(I,J)
10          CONTINUE
            testset(2)                          TS2
            testset(3)                          TS3
            testset(4)                          TS4
            testset(5)                          TS5
            IF (D(I).GT.0.0) GOTO 20            S3
                test(6) 1                       T6
                E(I) = F(I-1) + G(I)            S4
                F(I) = E(I) + H(I)              S5
                testset(6)                      TS6
20          CONTINUE                            S6
                testset(6)                      TS6
                P(I) = 2.0*P(I)                 S7
            test(7) 1                           T7
            DO 401 M = 1, 2                     L2^1
            ...
            test(8) 1                           T8
            DO 402 M = 3, 4                     L2^2
            ...
            testset(7)                          TS7
            test(9) 1                           T9
            DO 403 M = 5, 6                     L2^3
            ...
            testset(8)                          TS8
            DO 404 M = 7, 8                     L2^4
            ...
            testset(9)                          TS9
            GOTO 50
61          terminate
            GOTO 60
50      CONTINUE
        ...
60      CONTINUE
```

(a) Loop of Figure 5 Synchronized

Figure 13. Dependence Arc Elimination with Branches

different iterations. The first assumes that the false branch of the IF of $S_3$ is taken in both iterations. The second assumes that the false branch is taken in the first iteration, and the true branch is taken in the next iteration. The third assumes that the true branch is taken in the first iteration, and the false branch is taken in the second. Finally, the fourth assumes that the true branch is taken in both iterations. The CPGs are built the same as when no flow of control branches are present with two exceptions. When a statement is a branch, an arc is only placed from the branch to the statement that will execute after the branch, and not to every target of the branch. Finally any synchronization instruction that is not executed in the flow of control path for an iteration does not have any synchronization arcs coming in or going out of it.

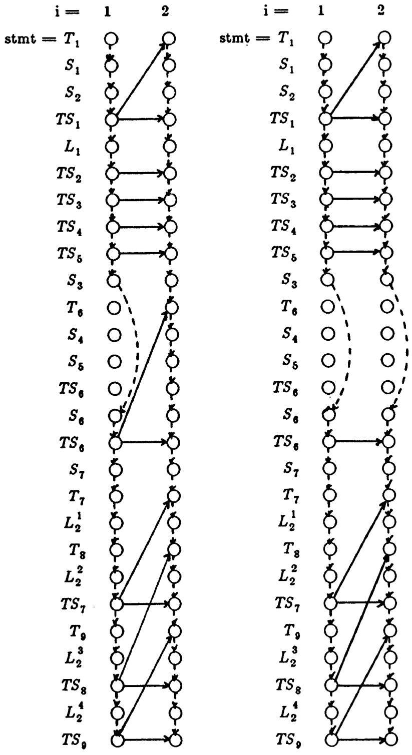


(b) CPGs for (a)

Figure 13. Continued

Eliminating a dependence arc is the same as with the CPG of Figure 11c except that it is necessary to find a path through every CPG constructed for the loop. This is necessary since the dependence arc must be redundant regardless of the flow of control path taken through the graph. It is possible to eliminate the dependence arc since if a path exists through all the CPGs, then the sink of the dependence must execute after the source of the dependence, without synchronizing the dependence, regardless of what branches are taken.

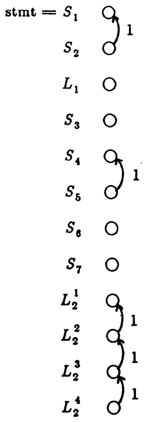
Consider, as an example, the dependence from $L_1$ to $L_2^4$, synchronized by the *testset* instruction, $TS_5$. Using the arcs in the CPGs provided by $TS_2$, and the architecturally dependent arcs, we can travel from the source to the sink of the dependence without using any arcs supplied by the $L_1$ to $L_2^4$ dependence, in this case those arcs connecting $TS_5$. Therefore we can eliminate this arc from the graph, since it is synchronized by other arcs. As each arc is eliminated in the DG, the corresponding synchronization instruction nodes in the CPG are eliminated so that they are not used when eliminating other dependence arcs. Likewise, the arcs representing dependences from $L_1$ to $L_2^3$ and $L_1$ to $L_2^2$ can be eliminated.



(c) CPGs for (a)

Figure 13. Continued

The dependence arc in the DG from $L_1$ to $L_2^1$ is eliminated by the backward dependence from $S_5$ to $S_4$, and the arcs provided by its *testset*, $TS_6$. After this arc is eliminated from the DG, no more arcs can be eliminated. Figure 14a shows the DG after dependence arc elimination, and Figure 14b shows the synchronized program after dependence arc elimination.



(a) DG for the Loop Nest of Figure 5

**Figure 14. The Loop of Figure 5 After Dependence Arc Elimination**

```
         DOSPREAD 50 I = 1, N
                  test(1) 1                              $T_1$
                  S(I) = S(I) + 1.0                      $S_1$
                  IF (S(I).GT0.0) GOTO 61                $S_2$
                  testset(1)                             $TS_1$
                  DO 10 J = 1, N                         $L_1$
                        A(I,J) = B(I,J) + C(I,J)
10                CONTINUE
                  IF (D(I).GT.0.0) GOTO 20               $S_3$
                        test(2) 1                        $T_2$
                        E(I) = F(I-1) + G(I)             $S_4$
                        F(I) = E(I) + H(I)               $S_5$
                        testset(2)                       $TS_2$
20                CONTINUE                               $S_6$
                        testset(2)                       $TS_2$
                        P(I) = 2.0*P(I)                  $S_7$
                  test(3) 1                              $T_3$
                  DO 401 M = 1, 2                        $L_2^1$
                        ...
                  test(4) 1                              $T_4$
                  DO 402 M = 3, 4                        $L_2^2$
                        ...
                  testset(3)                             $TS_3$
                  test(5) 1                              $T_5$
                  DO 403 M = 5, 6                        $L_2^3$
                        ...
                  testset(4)                             $TS_4$
                  DO 404 M = 7, 8                        $L_2^4$
                        ...
                  testset(5)                             $TS_5$
                  GOTO 50
61                terminate
                  GOTO 60
50       CONTINUE
         ...
60       CONTINUE
```

(b) Loop of Figure 5 Synchronized

Figure 14. Continued

Finding a path through all the CPGs for a loop is necessary and sufficient for eliminating a dependence arc. This is proven in [14]. It is also shown in [14] that dependence arc elimination is NP-hard, and a polynomial time algorithm for dependence arc elimination in loops without control branches is given. An exponential algorithm is given for loops with control branches. The exponential nature of the algorithm is shown to be not too bad in practice because of the small number of branches and the short dependence distances present in most loops.

If one is willing to not eliminate as many dependence arcs as possible, an easy but conservative approach is possible. By putting in arcs representing control exercised by the architecture only for non-branching statements, it will not be possible to go from the source to the sink of a dependence that is separated by a control flow branch. This means only dependences whose sources and sinks lie within a basic block will be eliminated. Therefore, only one CPG can be built and the dependence elimination can be done in polynomial time. Using this method, the dependence arcs from $L_1$ to the $L_2$ sub-loops would not have been eliminated in the example of Figure 13.

## Conclusions

We have shown that the automatic generation of synchronization instructions is not only practical, but straightforward. Many of the techniques discussed in this paper have been implemented in a pass for the Parafrase program restructurer, demonstrating their practicality. We have also shown that a set of synchronization primitives that allows fairly primitive dependence testing to take place can be used effectively to synchronize DO loops.

Several of the techniques discussed can be easily extended to other instruction sets. The algorithm used to place *testset* instructions can be used with any set instruction. The method of dependence arc elimination we have discussed can be immediately extended to most if not all architectures by modifying the placement of arcs in the CPG. Finally, loop splitting can be used effectively with any instruction set in which the number of synchronization registers is limited.

## References

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Massachusetts, 1977.

[2] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-989, October 1979.

[3] S. Chen, "Large-scale and High-speed Multiprocessor System for Scientific Applications - Cray-X-MP-2 Series," Proc. of NATO Advanced Research Workshop on High Speed Computing, Kawalik(Editor), pp. 59-67, June 1983.

[4] R.G. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois at Urbana Champaign, DCS Report No. UIUCDCS-R-84-1177, 1984.

[5] J.R. Beckman-Davies, "Parallel Loop Constructs for Multiprocessors," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-81-1070, May 1981.

[6] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared-Memory Parallel Machine," IEEE Trans. on Computers, Vol. C-32, No. 2, pp. 175-189, February 1983.

[7] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "CEDAR -- A Large Scale Multiprocessor," Proc. of International Conference on Parallel Processing, pp. 524-529, August 1983.

[8] D.T. Jackson, "Data Movement in Doall Loops," M.S. Thesis, University of Illinois at Urbana Champaign, CSRD Report No. 524, 1985.

[9] D.J. Kuck, The Structure of Computers and Computations, Volume 1, John Wiley and Sons, New York, 1978.

[10] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," Fourth International Computer Software and Applications Conference, October, 1980.

[11] D.J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proceedings of the 8-th ACM Symposium on Principles of Programming Languages, pp. 207-218, January 1981.

[12] Z. Li and W. Abu-Sufah, "A Technique for Reducing Synchronization Overhead in Large Scale Multiprocessors", Proceedings of the 12th International Symposium on Computer Architecture, pp. 406-413, June, 1985.

[13] S.F. Lundstrom and G.H. Barnes, "Controllable MIMD Architecture," Proceedings of the 1980 International Conference on Parallel Processing, pp. 19-27, 1980

[14] S.P. Midkiff, "Compiler Generated Synchronization for High Speed Multiprocessors", M.S. Thesis, University of Illinois at Urbana-Champaign, May 1986

[15] D.A. Padua, "Multiprocessors: Discussions of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-990, November 1979.

[16] C.D. Polychronopolous, Ph.D. Thesis, Work in Progress, University of Illinois at Urbana-Champaign, 1986.

[17] M.J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-82-1105, 1982.

# EFFICIENT EXECUTION OF PROGRAMS WITH PIPELINE CONFIGURATION OF A RECONFIGURABLE MULTIPROCESSOR*

Karam Mossad and Chuan-lin Wu

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract -- In this paper we introduce a program graph model for sequential programs. We then present two algorithms which can assign any program graph to be executed on a pipeline configuration having the minimum number of processors of a reconfigurable multiprocessor. We show that the time complexity of these algorithms is polynomial for large and practical classes of program graphs that include r-ary trees and serial/parallel graphs. We conject that the problem of assignment is NP-complete. Consequently, the complexity of the algorithms is the best that one can hope to achieve.

## 1. INTRODUCTION:

One of the main reasons of not being able to achieve the potential gain of concurrent and parallel systems is that the architecture is not well matched to the application of interest or vise versa [1]. In this paper an effort is made to match the sequential programs to a pipeline configuration of a reconfigurable multiprocessor [2]. The reconfigurable multiprocessor can be reconfigured to connect processors into commonly used topologies such as pipeline, tree, mesh, loop, and their composite [3]

There will be a transition from the normally known pipeline architecture [4,5,6] to a new one which will be called program pipelines, in which, each stage will be a processor, or many processors. To execute a sequential program concurrently on a program pipeline, the program is divided into a sequence of blocks. Each block is executed by one stage of the pipeline. It is assumed that the program will be executed frequently, and hence a reduction of total execution time will result if compared to the total execution time of the same program on a uniprocessor. A compiler program is a practical example which can be divided and which can repeatedly receive compilation tasks [7,8].

## 2. PROGRAM GRAPH MODEL

Definition 1:

An acyclic directed graph G is called Program Graph iff the following three conditions are satisfied:

1) Exactly on node in G has no in going edges; this node is called the entry node of G
2) Every node in G is reachable by a directed path from the entry node
3) Every node V in G is assigned a positive integer d(V) called the delay of node V

Definition 2:

Let G be a program graph. The delay of G, denoted d(G), is as follows:

$d(G) = \max [d(V_1), d(V_2), \ldots.., d(V_n)]$, such that V is in G.

Definition.3:

A program graph is called linear iff it consists of only one directed path.

Definition 4:

A linear program graph is called a pipeline with delay K iff for every node V in the graph, d(V) = K.

Definition 5:

Let G be a program graph and P be a pipeline with delay K. An assignment f of G into P is a function that assigns each node V in G to a node f(V) in P such that the following two conditions are satisfied:

1) For every two distinct nodes V and V' in G, if there is a directed path from V to V' then either f(V) and f(V') are the same node in P or there is a directed path from node f(V) to node f(V') in P.
2) The max value in the set

$\{ \sum_1^r d(V_1)\ V_1, \ldots\ldots V_1$ constitute a directed path in G,

and are assigned to the same node in P} is at most K, the delay of P.

The next lemma follows immediately from the above definitions.

Lemma 1:

If there is an assignment from a program graph G into a pipeline P, with delay K, then $d(G) \leq d(P)$, i.e., $d(G) \leq K$..

**Problem:** It is required to design an algorithim that takes any program graph G and any positive integer K, where K $\geq$ d(G), then defines a pipeline P with delay K and an assignment f of G into P such that the number of nodes in P is minimum.

*This work is supported in part by a grant from IBM Corporation

0190-3918/86/0000/0552 $01.00 © 1986 IEEE

Let us comment on the fact that our model program graph does not allow cycles, while sequential programs do have cycles. If a sequential program has a cycle, as shown in Fig. 1 - a as an example, then its execution can be pipelined only if there is an upperbound on the number of times the cycle is to be executed. In this case, we suggest two solutions on how to represent a cycle in our program graph model as follows:

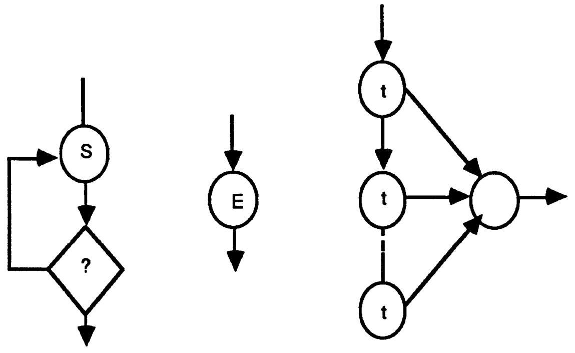
1 Collapse the cycle into one node in the program graph model. The execution time of the node equals t x s, where t is the time to execute the cycle once, and s is the upper bound on the number of times the cycle is executed. Collapsing is as shown in Fig. 1 - b

2 Unfold the cycle into a sequence of s nodes, each with execution time t, in the program graph model. Where t and s are as defined above. Unfolding is as shown in Fig. 1 - c.

Notice that the first solution will produce a program graph with a smaller number of nodes but the execution time of such nodes is bigger than those in the second solution.

An efficient assignment of linear program graphs into pipelines is given in the next section.

Fig. 1. Cycle Problem Solution



a) Cycle b) Collapsing method c) Unfolding method

## 3. EFFICIENT ASSIGNMENT OF LINEAR PROGRAM GRAPHS INTO PIPELINES:

In this section, we present an algorithm that takes any linear program graph G, and a positive integer K, where d(G) ≤ κ, then computes:

a) The minimum number p of nodes in a pipeline P with delay K such that there is an assignment of G into P, and
b) The assignment function F: G ------>P

Algorithm 1:

```
Input: G and K (as defined above)
Output: An integer p, and an array f (as defined above)
Variables:
    sum: integer;
    Node: (1, . . . . . ,g, g + 1)
Steps: node: = 1;
       p: = 0;
       While node ≤g
       do  p = p + 1;
           sum: = 0;
           While  node ≤ g and sum + d(node) ≤ K
              do  f [node] : = P;
              sum: = sum + d (node);
              node: = node + 1
              endwhile
          endwhile
End of Algorithm 1
```

The following theorem estabishes the correctness of Algorithm 1

Theorem 1:

Let G be any linear program graph, and K be any positive integer such that K ≥ d(G). And let p and f be the output of applying Algorithm 1 to G and K:.

a) f is an assignment of G into a pipeline with delay K that has p nodes.
b) If there is an assignment of G into a pipeline with delay K, then this pipeline must have at least p nodes.

## 4. EFFICIENT ASSIGNMENT OF GENERAL PROGRAM GRAPHS INTO PIPELINES:

We present in this section an algorithm that takes any (general) program graph G, and a positive integer K, where d(G) ≤ K, then computes:

a) The minimum number p, of nodes in a pipeline P with delay K such that there is an assignment of G into P, and
b) An assignment f of G into P.

Algorithm 2:

```
Input:  G and K (as defined above)
Output: An integer p, and an array f (as defined above)
Variables:
        node: (1, . . . . . ,g)
Steps:
    1.  p: = O
    2.  for node = 1 (1, . . . . . ,g) do f [node]: = O endfor
        3.  for each complete path T in G do
            *  Recall that T is in fact a linear
            *  Program graph
               a) Apply algorithm 1 to T and K, and let the
                  output be q and h respectively.
            *  By theorem 1, q is the minimum number
            *  Of nodes in a pipeline Q with delay K
            *  Such that there is assignment h of
            *  T into Q.
               b) p: = max (p,q)
               c) for every node in T do
                     f [node]: = max (f [node], h [node])
        endfor
    endfor
```

The following theorem estabishes the correctness of Algorithm 2

Theorem 2:

Let G be any linear program graph, and K be any positive integer such that K ≥ d(G). And let p and f be the output of applying Algorithm 2 to G and K.

a) f is an assignment of G into a pipeline with delay K that has p nodes.
b) If there is an assignment of G into a pipeline with delay K, then this pipeline must have at least p nodes.

## 5. TIME COMPLEXITY ANALYSIS:

In this section we present some results concerning the time complexity of algorithms one and two. In particular we show that:

1) Algorithm 1 has a linear time complexity.
2) Algorithm 2 has a polynomial time complexity for large (and practical) classes of program graphs.
3) Algorithm 2 has an exponential time complexity in general.

Theorem 3:

The time complexity of Algorithm 1 is 0(n), where n is the # of nodes in the linear program graph G.

Next we discuss the complexity of Algorithm 2 for some practical classes of program graphs, namely, and serial parallel graphs.

Definition 6: An r-ary tree is a directed rooted tree where each nonleaf node has at most r children.

Definition 7: A serial/ parallel graph is a program graph that has one entry node, one exit node, and a block in between as shown in Fig. 2, where a block is defined recursively as follows:

1) One node as shown in Fig. 3 a
2) One node followed by a block as shown in Fig. 3 b, or
3) One node that branches to two blocks as shown in Fig. 3 c

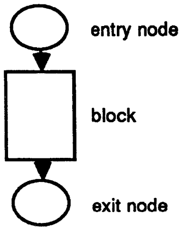


Fig. 2 serial/parallel graph

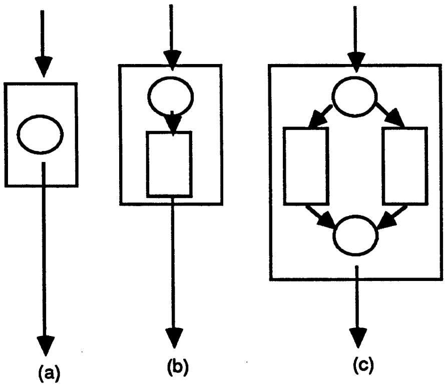


Fig. 3 The recursive definition of a block

Corollary 1:

The time complexity of Algorithm 2 is

$0 ( 1/r [n (r - 1) + 1] \log_r [n (r - 1) + 1])$ i.e.

$0 (n \log_r n )$ when applied to

program graphs in the shape of an r - ary tree with n nodes.

Corollary 2:

The time complexity of Algorithm 2 is $0 (n^2)$ when applied to a program graph in the shape of a serial/parallel program graph with n nodes.

Unfortunately, the time complexity of Algorithm 2 is not always polynomialas demonstrated by the following theorem.

Theorem 4:

There is a class of program graphs, namely, complete acyclic, for which the time complexity of Algorithm 2 is exponential.

## 6. CONCLUSION:

The execution of sequential programs can be speeded up by executing them on reconfigurable architecture. Specifically, we have introduced a program graph to model sequential programs and presented two algorithms which can be used to assign any program graph to a pipeline having the minimum number of processors. We proved the correctness of both algorithms and discussed their time complexity. The complexity analysis showed that the complexity of Algorithm 1 is linear, and the complexity of Algorithm 2 is polynomial when applied to program graph on the shape of r-ary trees or serial/parallel graphs. It is, also, shown that there exists a class of program graphs for which the complexity of Algorithm 2 is exponential. We suspect that the problem is NP - complete in general, and if it is so, then the time complexity of Algorithm 1 and 2 is the best that one can hope to achieve.

## References

[1] A.K. Jones and P. Schwartz, "Experience Using Multiprocessor Systems - A Status Report", Computing Survey, Vol. 12, No. 2, June 1980.

[2] C. Wu, K. Mossaad, W. Lin, "Architecture of a Distributed Reconfigurable Multimicro-Computer Network", Proceedings of Internatinal Computer Symposium, 1982.

[3] W. Lin and C. Wu, "Design of Configuration Algorithms for a Multiprocessor-STAR", Proceedings of the International Conference on Parallel Processing, 1985.

[4] P. M. Kogge, "The Architecture of Pipelined Computers", Hemisphere Publishing Corp. and Mcgraw-Hill Book Company, 1981.

[5] C.V. Ramamoorthy and H. F. Li, "Pipeline Architecture", Computing Survey, Vol. 9. No. 1, March 1977.

[6] K. Hwang, and F.A. Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill, 1984.

[7] P. El-Dessouki, W. Huen, and M. Evens, "Towards a Partitioning Compiler for a Distributed Computing System", Proceedings of the International Conference of Parallel Processing, 1979.

[8] J. K. Ousterhout, "Medusa, a Distributed Operating System", UMI Research Press, 1981.

# SIMULTANEOUS BROADCASTING IN MULTIPROCESSOR NETWORKS

*K. N. Venkataraman*
*George Cybenko*
*David W. Krumme*

Department of Computer Science
Tufts University
Medford MA 02155

## Abstract

The problem of *simultaneous broadcasting* in multiprocessor networks is studied in this paper. This problem, in which every processor has a token that must be broadcast to every other processor in the network, arises in the development of applications programs for multiprocessor networks. The main result of this paper is that the optimal algorithm for the token broadcast problem (under a restricted model of communication) for a complete graph network with N processors runs in $\lceil 1.44 \log_2 N + 4 \rceil$ steps. It is also shown that the same problem can be solved on a hypercube in $\lceil 2 \log_2 N \rceil$ steps.

## 1 Introduction

In writing applications programs to run on the INTEL iPSC/d5 hypercube multiprocessor [2], we have found a number of uses for the following construct. Every processor has a token that must be broadcast to every other processor and these tokens can be combined so that the effective size of combined tokens is the same as the size of the original tokens. This assumption that tokens can be combined applies where communication or packet-size overhead is the dominating factor; or where tokens can be combined arithmetically or otherwise. This construction is useful for synchronizing the starting times of the actual applications code on all processors for accurate timing measurements. Another equally important use for such a construct is in the computation of a global sum where each processor is storing one of the summands and the sum must be computed and communicated to all processors. This situation has arisen, for example, in sparse eigenvalue and linear equations solving techniques with which we have been experimenting [1]. In general, the communications requirements demanded by this problem are necessary and sufficient to evaluate any function of all the tokens at all of the nodes in the network, and the assumption that tokens can be combined yields a simple yet reasonable abstraction of the problem.

Using realistic models for interprocessor communication in a local memory, message passing multiprocessor network, we have studied good and optimal algorithms for performing this task on hypercubes and other multiprocessor networks. It appears that the minimal time parallel algorithm for solving this global token passing problem provides a useful criterion for evaluating a multiprocessor network topology. It is quite distinct, as our examples show, from previously used characteristics such as diameter, girth, density and connectivity [7].

## 2 Token Broadcast Problem

In this section we shall state our version of the *token broadcast problem* and the two different models of communication that we shall focus on in this paper.

**Definition:** Given a network N, specified by the interconnection graph G=(V,E), and a token $t_i$ associated with each of the nodes $p_i$, $1 \le i \le n = |V|$, the *token broadcast* problem is that each node should broadcast its token to all the other nodes in G and as a result at the end of the broadcast each node should contain the tokens $t_1, \cdots, t_n$.

This version of the broadcast problem is different from the *routing problems* that have been studied [3,4,8,10]. We require that all tokens be available at all of the nodes at the end of the broadcast.

We are interested in efficient algorithms for solving the broadcast problem for different interconnection graphs that have been proposed [7,8,9]. In the current paper we shall focus our attention on the *hypercube* and the *complete graph.* We shall investigate the algorithms for the broadcast problem under the two models of communication described below. In both the models we assume that it takes the same amount of time to transmit one token or a collection of tokens across the communication link from a node to its neighbour and we shall use that measure as our unit of time. This assumption is realistic to a large extent on the INTEL iPSC multiprocessor system due to packet-size effects and further, makes it easier to analyze the properties of optimal algorithms for the broadcast problem.

**Model A:** In each step of the computation, messages can be transmitted on any links but only in one direction along each link in the network. In other words, any number of edges can be active but each edge can be used only in one direction at a given step. This is a good model of

This work was partially supported by NSF Grant DCR-8505634 and US-Spain Joint Committee for Scientific and Technological Cooperation Grant CCA8411019.

certain architectures, including the iPSC, if one assumes that actual communication time dominates.

**Model B:** In each step of the computation, a processor can either receive a message from one of its neighbours or send a message to one of its neighbours. In effect, in a given step at most half the processors communicate with the other half. This model describes virtually any architecture under the assumption that the overhead or setup time at each node is great compared with the actual communication time. On the iPSC, this is often the best model.

For an architecture like that of the Ncube [5] with bidirectional channels, a bidirectional version of model A could apply, and in that case the graph-theoretic questions that we ask have easy answers, while model B if appropriate would still most naturally use unidirectional communication.

## 3 Optimal Algorithms for Complete Graphs

In this section we shall study the token broadcast problem for the complete graph interconnection pattern. We will present an algorithm for each of the communication models and we will also show that the algorithms are optimal in terms of number of steps.

It is easy to see that we can solve the problem in two steps under model A. We pick a distinguished node $p$ and in the first step, every node (except $p$) sends its token to $p$ and in the second step $p$ sends all the accumulated tokens to each of the other processors. It is also clear that this algorithm is optimal in terms of number of steps.

For model B, we shall first establish a lower bound on the number of steps and then present an algorithm which runs at that speed. Let us assume that there are $N = 2^d$ nodes in the network. We shall introduce a measure on the amount of information at a given node and study the maximum rate of growth of the total information over all the nodes in the network. At first, one may try the number of tokens $\nu_i$ at processor $p_i$ as a measure of information content at that processor with $\Sigma_{i=1}^{n}\nu_i$ being the total information over all the nodes in that network. A simple analysis shows that the maximum rate of growth of information is 2 and the resulting lower bound on the number of steps is $d$. Instead, let us use $\nu_i^k$ as the measure and find that $k$ which will give us the best lower bound. A careful analysis shows that $k = 2$ is the best choice; given $k = 2$ a straightforward calculation establishes the following theorem.

**Theorem 1:** For a network of N processors whose interconnection pattern is a complete graph, any algorithm that solves the token broadcast problem under model B takes at least

$$\left( \frac{\log_2 N}{\log_2(\frac{1+\sqrt{5}}{2})} \right)$$

steps. (This translates to roughly 1.44 $\log_2 N$ steps.)

The following theorem shows that the lower bound is actually matched by the optimal algorithm to the broadcast problem.

**Theorem 2:** There exists an algorithm that solves the token broadcast problem under model B for a network of N processors whose interconnection pattern is a complete graph in

$$\left\lceil \frac{\log_2 N}{\log_2(\frac{1+\sqrt{5}}{2})} \right\rceil + 4$$

steps.

*Proof:* The general description of the actual algorithm which achieves this bound is as follows. After $k \geq 2$ steps of the algorithm half the number of nodes have accumulated $2 \cdot fib(k)$ tokens and the other half have accumulated $2 \cdot fib(k-1)$ tokens where $fib(k)$ is the $k^{th}$ element in the *Fibonacci* sequence with $fib(0) = 0$ and $fib(1) = 1$ . At each step, the nodes with larger number of tokens send their tokens to the other half of the nodes; care should be taken so that the tokens that a node receives is disjoint from the tokens that it already has. The actual protocol is given below.

We assume $N$ is even; if not, choose any one node and send its token to another site, solve the problem on all nodes but the one, and lastly send from some site to the chosen node. Let $n = N/2$ and label the nodes $p_0, p_1, \cdots, p_{n-1}, q_0, q_1, \cdots, q_{n-1}$.

*Step 0*: Transmit from $p_i$ to $q_i$ for all $i$, $0 \leq i \leq (n-1)$.

*Step 1*: Transmit from $q_i$ to $p_i$ for all $i$, $0 \leq i \leq (n-1)$.

*Step k, $k \geq 2$*: If $k$ is even transmit from $p_i$ to $q_{i+fib(k-1)}$ for all $i$, and if $k$ is odd, transmit from $q_i$ to $p_{i+fib(k-1)}$ for all $i$, where + denotes addition modulo $n$.

It is easy to verify that this strategy has the following properties: after step $k$, for each $i$ either $p_i$ or $q_i$ has all tokens from all nodes $p_j$ and $q_j$ for $j = i, i-1, \cdots, i - fib(k+1)+1$ where − denotes subtraction modulo $n$; the other one ($p_i$ or $q_i$) has all tokens from nodes $p_j$ and $q_j$ for $j = i, i-1, \cdots, i - fib(k) + 1$; after step $k$, nodes $p_i$ and $q_i$ ($k$ even) or nodes $q_i$ and $p_i$ ($k$ odd) have $2 \cdot fib(k)$ and $2 \cdot fib(k+1)$ tokens respectively; each step involves transmitting from the sites with the larger number of tokens to the sites with the smaller number. Then it can be seen that when $fib(k) \geq n$, every node has all tokens and the broadcast is finished. It is well known that $fib(k) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-2}$, and from this inequality it is straightforward to count steps and arrive at the above bound.

It is not obvious that this algorithm is optimal, and our only proof of its optimality is that it asymptotically takes exactly the same number of steps given by the lower bound. In fact this algorithm achieves at each step very close to the maximum rate of growth of information as we defined it in establishing the lower bound, and it solves the broadcast problem efficiently (usually within 1 step of the lower bound) even for small values of $N$.

The optimal algorithm does not make use of all the links in the complete graph. The communications required by this algorithm define a network whose interconnection pattern is a regular graph of valence O(log N). Further this algorithm would be optimal even if one were to charge for the transmission as a linear function of the size of the message.

## 4 Broadcast Problem on a Hypercube

The hypercube $H_d$ of dimension $d$ has (0,1)-vectors of length $d$ as nodes, with an edge between two nodes if they differ in exactly one coordinate [7]. It is easy to observe that any algorithm under either model of communication will take at least $d$ steps to solve the broadcast problem since the *diameter* of the hypercube $H_d$ is $d$.

It is not too difficult to find algorithms that under model A solve the broadcast problem in $d+1$ steps. The simplest was first presented in [6] and can be described in this way. In the even numbered steps, nodes labelled with even parity send their collection of tokens to their respective neighbours labelled with odd parity and during odd numbered steps, the nodes with odd parity send their collection of tokens to their respective neighbours with even parity. But we have recently discovered for $d \geq 3$ a way to merge two different strategies of this sort into a combined strategy that solves the problem in $d$ steps which is optimal.

Let us look at the broadcast problem for model B. Let $T_i^+$ ($T_i^-$) denote the computation step where each node whose $i^{th}$ bit is 0 (1) transmits all its accumulated tokens to its neighbour whose corresponding bit is 1 (0). Now consider the sequence $\sigma = T_1^+, T_1^-, \cdots, T_d^+, T_d^-$. It can be shown that any permutation of $\sigma$ is a solution to the broadcast problem, and furthermore if any step is omitted from such a permutation then it is no longer a solution. Thus this algorithm runs in $2d$ steps but no faster. We have tried a number of different direct approaches, and they all resulted in algorithms that run in $2d$ steps. However, we have recently discovered an extremely complicated algorithm that solves the problem in 17 steps on the 9-cube, so we know that $2d$ is not optimal. But we believe there will be no simple algorithm that is better than $2d$, and we also suspect that the optimum is closer to $2d$ than to $1.44d$, so for practical purposes we consider the permutations of $\sigma$ to be the recommended algorithms for the broadcast problem under model B.

## 5 Other Multiprocessor Networks

Given an arbitrary interconnect network, it is interesting to consider the minimal number of steps needed by an algorithm to solve this token broadcast problem for either of the models we consider here. Especially in the case of Model B, the minimal number of steps appears to effectively capture the speed at which a network can perform a global synchronization. This measure is quite distinct from diameter or other common measures.

Restricting our attention to Model B (which is arguably most realistic and in fact models the performance of the INTEL iPSC family), our results demonstrate that a complete graph (diameter of 1) is less than 50% more efficient than a hypercube with the same number of nodes. Another way of viewing our result is that there exist interconnection networks with about 50% more communications channels than a hypercube that are optimal among all networks for this token passing problem. Our understanding of these optimal networks is extremely rudimentary at present.

A good example to illustrate this new measure is a star network with one central node and $n$ spokes. Each of the $n$ spokes is connected to the central node only. The diameter of this graph is 2 but the optimal algorithm under Model B requires $2n$ steps. This observation captures precisely the common wisdom that a star network encounters too much contention at its central node.

A linear array of nodes requires either $n$ or $n+1$ steps for an optimal algorithm (depending on the parity of $n$) and this does in fact correspond closely to the diameter in this case. We have shown that for an $n_1 \times n_2 \times \cdots \times n_k$ multidimensional grid, the problem can be solved in $\Sigma_{i=1}^{k} p(n_i)$ steps, where $p(2m) = 2m$, and $p(2m+1) = 2m+2$. However, the diameter of such a grid is $\Sigma n_i - k$ and so this solution is always within 2k of the diameter which is a lower bound. We have also shown that for a ring of $n$ nodes it can be solved in $n/2 + \sqrt{2n}$ steps. In both cases, the solution is optimal and the number of steps is close to the diameter. We do not know what the optimum is for the hypercube.

## 6 Conclusions

The main result of this paper is that the optimal algorithm for the token broadcast problem (under a restricted model of communication) for a complete graph network with N processors runs in $\lceil 1.44 \log_2 N + 4 \rceil$ steps. We observe that the hypercube network is not very much slower as the same problem can be solved on a hypercube in $\lceil 2 \log_2 N \rceil$ steps. There are two immediate questions that one would like to solve. What is the optimal algorithm for the hypercube network? Is there a structural property of the interconnection graph that totally captures the difficulty of the token broadcast problem?

## References

[1] G. Cybenko, Alva Couch, David Krumme, and K. N. Venkataraman. Heterogeneous processors on homogeneous multiprocessors. To Appear in *Proceedings of Second ARO Workshop on Scientific Computing and Medium-Scale Multiprocessors*, SIAM, Philadelphia, 1986.

[2] INTEL Corporation. *iPSC Data Sheet.* 1985.

[3] G. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. on Computers*, C-30(2):93–100, 1981.

[4] D. Nassimi and Sartaj Sahni. Data broadcasting in SIMD computers. *IEEE Trans. on Computers*, C30(2), February 1981.

[5] NCUBE Corporation. *Product Announcement.* Tempe Arizona, 1985.

[6] Y. Saad and Martin Schultz. *Data Communication in Hypercubes.* Technical Report, Research Report YALEU/DCS /RR-428, October 85.

[7] Leonard Uhr. *Algorithm-Structured Computer Arrays and Networks.* Academic Press, 1984.

[8] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[9] J.D. Ullman. *Some Thoughts about Supercomputer Organization.* Technical Report STAN-CS-83-987, Department of Computer Science, Stanford University, October 1983.

[10] L. G. Valiant. A scheme for fast parallel communication. *SIAM J. Computing*, 11(2), 1982.

# VECTOR PROCESSING ON THE ALLIANT FX/8 MULTIPROCESSOR

Walid Abu-Sufah and Allen D. Malony
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
104 S. Wright St.
Urbana, Ill. 61801-2987

## Abstract

The Alliant FX/8 multiprocessor implements several high-speed computation ideas in software and hardware. Each of the 8 computational elements (CEs) has vector capabilities and multiprocessor support. Generally, the FX/8 delivers its highest processing rates when executing vector loops concurrently [5]. In this paper, we present extensive empirical performance results for vector processing on the FX/8. The vector kernels of the LANL BMK8a1 benchmark are used in the experiments. We execute each kernel on 1 and 8 CEs and show the measured execution rate (in MFLOPS) as a function of vector length. We assess the performance of 1 CE as a vector processor by finding the vector lengths where vector processing exceeds that of scalar processing and calculating Hockney's $n_{1/2}$. For 8 CEs, we give upper/lower bounds on the achieved speedups and on the multiprocessing overhead. We also show the speedup variation as the number of CEs increases from 2 to 8. Our results reveal some interesting phenomena. Vector processing performance in a machine with a multi-level memory hierarchy, such as the FX/8, depends significantly on where the referenced vectors reside. Execution from memory, rather than from cache, degrades performance by a factor up to 3.7. Although speedups around 7 can be achieved for most stride-1 kernels when executed on 8 CEs, the maximum execution rates occur only for a narrow range of vector lengths ($O(1000)$). Performance drops rapidly when the vector lengths deviate slightly from the optimal values. This phenomena is not observed when executing on a single CE; the peak performance is obtained when the vectors are 32 elements long and remains close to the maximum for longer vector lengths ($O(1000)$). The kernels do not gain any appreciable speedup when the number of CEs is increased beyond 4 for short ($O(100)$) or long ($O(10{,}000)$) vectors. Multiprocessing of some indexed vector kernels results in almost no speedup due to the synchronization necessary to enforce output dependencies.

## 1. Introduction

The Alliant FX/8 is a shared memory multiprocessor system with a maximum advertised performance of 94.4 millions of floating point operations per second (MFLOPS) for single precision computations [5]. Each of its 8 computational elements (CEs) has vector processing capability with a peak advertised execution rate of 11.8 MFLOPS. The FX/8 is one of the several machines which have been announced in the last few years that use different forms of parallelism to exceed the performance attainable from the technology used in the implementation[a]. The FX/8 combines several interesting high-speed computation ideas in both software and hardware [12], [20], [21], [19], [13], [17]. It has an interactive optimizing Fortran compiler which transforms loops in subroutines to execute in vector mode on a single CE, vector-concurrent mode on multiple CEs, or scalar-concurrent mode on multiple CEs [5]. The operating system, Concentrix, is a multiprocessor Unix based on Berkeley 4.2 BSD. Multiprocessing is realized by concurrency control hardware in each CE which is accessed using special concurrency instructions. The 8 CEs of the system are crossbar connected to a shared, direct mapped, cache. The cache is connected to the shared memory via a bus. A more detailed description of the FX/8 is presented in the Section Two.

The performance assessment of a vector multiprocessor machine, like the FX/8, is important because of the great amount of effort that was spent in the last decade to develop vector algorithms for different applications and to enhance the capabilities of vectorizing preprocessors to detect vector loops in dusty deck codes [19]. In addition, there is little empirical data in the literature modeling the behavior of vector multiprocessors [7], [11]. This paper presents empirical results on the vector performance of the Alliant FX/8 multiprocessor. The thirteen vector kernels of the Los Alamos National Laboratory benchmark **BMK8a1** (for double precision computations) were used in our experiments [14].

In Section Three we report and discuss the experimental results. We show the delivered performance for each kernel when executed on one CE. A single CE demonstrates the classical performance behavior of a vector processor where the maximum performance is sustained over a wide range of vector lengths. However, as cache misses increase for longer vector lengths, the performance drops sharply to a rate where the cache hit ratio is at a minimum. To characterize the vector performance of 1 CE, we determine the vector length where each kernel starts executing faster in vector mode than in scalar mode and calculate $n_{1/2}$, the vector length at which the CE is supposed to deliver half of its peak performance ($r_{\infty}$), as described by Hockney in [16]. These results indicate that a single CE processes short vectors efficiently.

For 8 CEs, we present the delivered performance, speedup, and multiprocessing overhead for each kernel. We observe that the

* This work was supported in part by the **National Science Foundation** under Grants No. NSF DCR84-10110 and NSF DCR84-06916, the **US Department of Energy** under Contract No. DOE DE FG02-85ER25001 and the **Center for Supercomputing Research and Development** at the University of Illinois at Urbana-Champaign.

[a] For a rather comprehensive survey of such machines and brief descriptions see [9], [10].

0190-3918/86/0000/0559 $01.00 © 1986 IEEE

execution rate increases as the vector length increases and then drops significantly to a minimum rate. The results show that the maximum performance for each kernel on 8 CEs is sustained over a significantly smaller range of vector lengths than for 1 CE. This is reflected in the speedup and multiprocessing overhead calculations. Speedup, $S_p(n)$, is defined as $t_1/t_p$ where $t_1$ and $t_p$ are the execution times of a kernel for vector length $n$ executed on 1 and $p$ CEs, respectively. We define the **machine efficiency**, $E_m(n)$, as the maximum delivered execution rate divided by the peak advertised execution rate of the machine and **multiprocessor efficiency**, $E_p(n)$, as $S_p(n)/p$. Multiprocessing overhead, $OV_p(n)$, is equal to $1\text{-}E_p(n)$. Our results indicate that only modest improvements in speedup are achieved when processing short ( $<=100$) and long ($\geq 10{,}000$) vectors on more than 4 CEs.

To determine the effect that the cache has on performance, we repeat the experiments for each kernel such that cache misses will be encountered whenever possible. Our measurements show that the performance decreases by a factor up to 3.7 when the vectors are referenced from memory instead of from the cache.

The BMK8a1 benchmark contains kernels with subscripted vectors. These vector kernels run slower than the stride-1 kernels. In Section Three, using one of the indexed kernels, we briefly discuss issues which affect the performance of such kernels. In Section Four we make some concluding remarks.

## 2. The Experimental Environment

We performed our experiments at the Center for Supercomputing Research and Development (CSRD) of the University of Illinois[b]. The configuration of the FX/8 used for these experiments is shown in Figure 1. The computational complex of the FX/8 contains 8 CEs. When executing concurrency instructions, the CEs communicate via a concurrency control bus. Each CE has a computational clock period of 170 nsec with a peak execution rate of 11.8 MFLOPS and 5.9 MFLOPS for single and double precision computation, respectively [5], [9]. With the 8 CEs working concurrently, the FX/8 advertised peak performance is 47.2 MFLOPS for double precision computations. The CEs are connected by a crossbar switch to a direct-mapped, write-back, shared cache of 16K double precision words[c]. The cache is implemented in 4 quadrants with a peak interleaved bandwidth to the CEs of 47.125 MW/sec. It is connected to a 4 MW shared memory via a bus with a peak bandwidth of 23.5 MW/sec. The system also contains 6 interactive processors (IPs) connected to their own caches as shown in Figure 1. The IPs primarily perform operating system related functions and I/O operations.

A computational element has vector processing capabilities as well as multiprocessing support. It has a rich set of arithmetic, logical and comparison vector instructions plus vector move instructions including scatter, gather and merge. There are 8 32-bit data registers, 8 address registers, 8 double precision floating point registers, and 8 32-element, double precision vector registers in each CE. Operands of vector instructions can come from vector registers, vector and floating point registers, or vector registers and the cache. Chaining is also supported for vector add-multiply and vector multiply-add instructions. Multiprocessing is supported by concurrency instructions which permit iterations of a loop to be executed concurrently across multiple processors in the CE complex.

The Alliant FX/8 Fortran compiler provides automatic detection of vector and/or multiprocessed loops. It optimizes code for scalar, vector and concurrent execution. Based on data dependency analysis, loops are optimized to execute in one of four modes: vector, scalar-concurrent, vector-concurrent, or concurrent-outer/vector-inner [5]. The FX/8 operating system, Concentrix, extends Berkeley Unix 4.2 to provide support for multiple processors and a large virtual space.

The FX/8 system maintains timing information for each program which is accessible through Fortran library routines and can be used for measurement purposes. Our experimentation procedure attempted to remove any inconsistencies that might result in the performance measurements due to the resolution of these timing tools by assuring a long running time relative to the granularity of the timed event. This was achieved by enclosing each kernel in a serial timing loop which repeats the execution of the kernel as many times as needed to obtain reliable timing data. All measurements were performed in standalone mode. Each vector kernel was executed five times for vector lengths varying between 1 and 100,000. The repetition of the experiments was necessary due to significant variations in the execution rates from one run to another for certain regions of vector lengths.

The Los Alamos National Laboratory benchmark BMK8a1 contains thirteen vector kernels designed to reflect the vector statements which are widely encountered in scientific codes [14]. Each kernel is a different combination of add and multiply operations of vectors and scalars that stores the outcome into a result vector. Some kernels use an additional vector to index an operand or result vector. In our notation used to identify the different kernels, $v$ is a vector, $s$ is a scalar, $p$ denotes addition, $t$ denotes multiplication, and $i$ denotes an indexing vector[d]. For instance, ***vts*** is the kernel $v1 = v2 * s$ and ***vi=vtv*** is the kernel $v1(i+k) = v2 * v3$ where $i$ is the indexing vector and $k$ is a constant. The complete list of vector kernels is:

***vps***
***vts***
***vpv***
***vtv***
***vtvps***
***vpvts***
***vtspvts***
***vtvpv***
***vtvpvtv***
***vips***
***vi=vtv***
***vpvtvi***
***vi=vipvtv***

## 3. Experimental Results

The performance of a one CE system is measured in order to calculate speedup and other performance metrics for multiple CEs. Examining the one CE results reveals interesting characteristics of the behavior of a vector processor when accessing data in a multi-level memory. The 8 CEs results show the performance improvement obtainable from vector-concurrent operation. The vector length region where maximum execution rate is achieved using 8 CEs is narrower than for one CE. However, the speedup in this region is around 7 for most stride-1 kernels. Comparing the performance for one and multiple CEs reveals important observations on the number of CEs which can be

[b] At the time the work reported in this paper was performed, the machine did not run production releases of the OS and the compiler. However, we believe that our conclusions will not change significantly when these releases are available.

[c] A double precision word is 64 bits wide.

[d] ***vi*** denotes a vector, $v$, indexed by another vector, $i$.

efficiently employed in a vector multiprocessor system. The performance results for the indexed kernels provides qualitative measure of the difficulties encountered when attempting to improve the performance of some codes using multiple vector processors.

### 3.1. One CE Performance

Figure 2 shows the maximum measured execution rate as a function of vector length for each kernel running on 1 CE. We observe that the behavior of all the kernels is similar; the computational rate increases as the vector lengths increase and reaches a maximum at vector length $n_{peak}$. For each kernel, the execution rate stays within a small percentage of the maximum until the vector length reaches a value denoted by $n_{drop}$. The computational rate then starts to fall until a vector length denoted by $n_{min}$ is reached. For vectors longer than $n_{min}$ the execution rate remains rather constant. These three vector length points are shown for the ***vtspvts*** kernel in Figure 2. We identify four regions for each performance curve: the **cache rate region** $(1 \leq n \leq n_{drop})$, **maximum rate region** $(n_{peak} \leq n \leq n_{drop})$, the **falloff region** $(n_{drop} < n \leq n_{min})$, and the **minimum rate region** $(n > n_{min})$. In the cache rate region, the size of the data referenced by each kernel is small enough such that the cache hit ratio is maximized. The performance in this region is characteristic of vector processors where the execution rate rapidly increases to a maximum point which is sustained as the vector length increases. The wide range of vector lengths where the execution rate stays within a small percentage of the maximum identifies the maximum rate region. The falloff region begins when the cache hit ratio starts decreasing. As the cache hit ratio continues to decrease for longer vector lengths, the number of the references to the shared memory increases and the performance drops until the cache hit ratio reaches its minimum at $n_{min}$. In the minimum rate region, the size of the referenced data is so large that a cache miss occurs whenever the kernel accesses the first word of a cache block[e].

Several factors affect the delivered performance for a given kernel at a particular vector length. These factors include the number of memory references, the number of floating point operations performed, the types of floating point operations, and the degree of chaining in the kernel. The ***vps*** kernel runs faster than the ***vts*** kernel because of operation type; ***vtvps*** runs faster than ***vtv*** due to the number of floating point operations performed and chaining; ***vtspvts*** runs faster than ***vtvpvtv*** due to the difference in the number of memory references. Table 1 shows the maximum MFLOPS measured for each kernel on 1 CE. The maximum execution rate occurs at vector length 32 for all kernels. This is expected since the vector registers are full at this vector length. Table 2 shows the execution rate for each kernel in the minimum rate region. Performance in the maximum rate region can be two times greater than the performance in the minimum rate region.

In order to determine how efficiently one CE processes short vectors, we found the vector length where execution in vector mode starts to be faster than in scalar mode. More performance can be achieved in vector mode for vector lengths > 2 for the stride-1 kernels and >6 for the indexed kernels. Hockney's $(n_{1/2}, r_{\infty})$ model can also be used to characterize the vector performance for one CE [16]. Table 3 shows that all kernels have an $n_{1/2} \leq 4$. This indicates that short vectors will be processed efficiently on the FX/8. However, most kernels deliver only around 1/3 of the measured peak execution rate at $n_{1/2}$ when executed on 1 CE instead of half the peak rate as expected by Hockney's model. It can be shown that Hockney's two parameter model $(n_{1/2}, r_{\infty})$ is simplistic when used to model real vector processors [4], [22].

### 3.2. Eight CEs Performance

Delivered Execution Rate

Figure 3 shows the maximum performance results when executing in vector-concurrent mode on 8 CEs. We observe that the execution rate rises more slowly and reaches the maximum rate region for much longer vectors than in the 1 CE case ($O(1000)$ compared to 32). This is partially due to the fact that vector-concurrent execution partitions the vector operation equally among the 8 CEs and longer vectors are required before the vector registers of each CE are maximally utilized[f]. Multiprocessing overhead associated with starting and sustaining vector-concurrent operations accounts for the further increase needed in vector length before the maximum rate is achieved. The performance of a kernel on 8 CEs is affected by the multi-level memory hierarchy for the same reasons as in the 1 CE case. In fact, we observed that the falloff region in the 8 CE performance curves coincides with the falloff region in the 1 CE performance curves for each kernel. However, the percentage drop in MFLOPS in the falloff region is greater with 8 CEs. Due to the initial slow performance increase to the maximum rate and the fixed falloff region, the maximum rate region spans a much smaller vector length interval than in the 1 CE case. This result has ramifications on how codes should be structured, with respect to vector length, so as to maximize the vector-concurrent performance when running on 8 CEs. In particular, we notice that if vector lengths deviate slightly from the maximum rate region, performance degrades rapidly.

Table 1 shows the maximum peak MFLOPS measured for each kernel when executing on 8 CEs, the vector length where the peak performance is delivered, and the machine efficiency $(E_m)$ at this vector length[g]. The machine efficiency is less for 8 CEs than for 1 CE due to multiprocessing overhead. All the kernels achieve < 50% machine efficiency and 10 kernels are < 30% efficient for 8 CEs. Table 2 is analogous Table 1 except the data for the MFLOPS in the minimum rate region is presented. It can be seen from the MFLOPS and the machine efficiency that a low cache hit ratio significantly reduces the performance. This subject is discussed in more detail in the section 3.3.

Speedup Results

Speedup is defined as $S_p(n) = t_1/t_p$ where $t_1$ and $t_p$ are the execution times of a kernel for vector length $n$ executed on 1 and $p$ CEs, respectively[h]. Since there were variations in measuring $t_1$ and $t_8$ over the five runs, we define the lower bound on the speedup of a kernel with vector lengths $n$, $L_{S(n)}$, to be the ratio of the smallest of the five $t_1$'s and the largest $t_8$. The upper bound on the speedup, $U_{S(n)}$ is calculated as the ratio of the largest measured $t_1$ to the smallest $t_8$. Figure 4 shows the upper and lower bound speedup curve for the ***vtv***

[e] A cache block contains four double precision words.

[f] This analysis is supported by the observation that the execution rate has a local maximum at vector length 256 for almost all of the kernels. At vector length 256, the vector registers for each CE are full.

[g] By definition, the maximum $E_m$ occurs at this vector length.

[h] In the remainder of the paper, $S(n)$ will be used to denoted $S_8(n)$.

kernel[i]. We observe that for all kernels the speedup upper bound is less than 4 for vector lengths smaller than 500 and less than 5 for vector lengths greater than 10,000. The maximum upper bound speedups for all the kernels are shown in Table 4. Eight of the 13 kernels have a maximum upper bound speedup greater than 7. The speedup of 4 of the remaining kernels is between 5 and 7. The indexed kernels have the smallest speedups; ***vi=vtv*** has a maximum upper bound speedup of only 1.32.

By comparing the vector lengths in Tables 1 and 4 we observe that the vector lengths where the peak MFLOPS and maximum speedup occur are not necessarily the same for a given kernel. Table 5 shows the speedup upper bound at the vector lengths where each kernel runs at its peak execution rate. We notice from Figure 4 and Tables 4 and 5 that the lower and upper bounds on speedup can differ significantly (up to 57%). This variation occurs in the falloff region and is primarily a result of the nondeterministic behavior of the cache in this region from one run to another. However, this variation is insignificant for short and long vector lengths ($n<500$ and $n>10{,}000$).

Figure 5 shows the speedup as a function of the number of CEs for stride-1 kernels. The envelopes in the figure enclose the speedup curves for all kernels at vector lengths 100, 1K and 100K. The speedup curve for ***vtvps*** is shown as a dashed line and roughly represents the median speedup within each speedup envelope. We observe that for both short and long vectors the speedup gained by increasing the number of processors beyond 4 is modest for most kernels. As the number of CEs increase from 4 to 8, the speedups approach 4.5 and 4 for vector lengths of 100 and 100K, respectively. For vector lengths of 1K, speedups are close to being linear in the number of CEs. These speedup results could be very useful to designers of multi-million dollar multiprocessor vector machines (e.g., the new Cray multiprocessors) in light of the mean vector lengths encountered in application codes ($\leq$468 in the Lawrence Livermore National Lab workload [24]).

Multiprocessing Overhead

The percentage multiprocessing overhead of a machine with $p$ processors when executing a kernel with vector length $n$, $OV_p(n)$, is given by $(1\text{-}S_p(n)/p)$*100%; $OV(n)$ denotes the overhead when $p$=8. We let $U_{OV(n)}$ denote the upper bound on the overhead and $L_{OV(n)}$ denote the lower bound. Figure 6 shows the upper and lower bound overhead curve for the ***vpvts*** kernel[j]. For all kernels, $U_{OV(n)}$ is greater than 50% for short ($n\leq$ 100) and long ($n\geq$ 10,000) vectors. For six of the nine stride-1 kernels, $U_{OV(n)}$ is less than 25% for vector lengths in the region $1000\leq n\leq 2000$. The overheads of the indexed vector kernels are greater than 30% for all vector lengths.

Table 6 identifies the vector lengths where the minimum $U_{OV(n)}$ occurs for all the kernels. We note that for the 9 stride-1 kernels the minimum $U_{OV(n)}$ is between 10-20%. For the indexed kernels, the minimum $U_{OV(n)}$ is close to 30% for 1 kernel, 40% for 2, and 85% for the fourth kernel.

### 3.3. Execution from Memory

In order to measure the vector-concurrent execution rate for a kernel with vector length $n$ ($1\leq n\leq 100{,}000$) such that the the maximum number of cache misses occurs, we reference the vectors as columns of two-dimensional arrays. The kernel is executed for the first column (of length $n$), then the second column, and so on. Since every column is distinct, new vectors are always being referenced. Also, the two-dimensional array sizes are declared such that when a column is referenced again by the timing loop, none of the data from the previous reference of that column will be present in the cache. We refer to the running of a kernel in this fashion as **execution from memory**. When a kernel is not restricted to execute in this manner and is able to take full advantage of the cache, we say that the kernel is **executing from cache**.

Figure 7 shows the execution rate from memory and from cache for the ***vtspvts*** kernel using 8 CEs. When the kernel executes from memory, the execution rate rises slowly as the vector length increases and reaches a maximum that coincides with the rate obtained in the minimum rate region when the kernel executes from cache. The same behavior is observed for the other kernels [3]. Table 7 shows the maximum performance degradation factors when kernels execute from memory instead of from the cache. This factor ranges between 1.35 and 2.26 when running on 1 CE. When executing on 8 CEs and using stride-1 kernels, the degradation factor is larger and ranges between 3.03 and 3.67. For indexed kernels, the degradation factor is between 1.49 and 2.08.

It is obvious that if the memory speed were increased, the execution performance from memory would increase. It is also expected that by increasing the size of the cache, the maximum rate region of the execution from cache curve would be extended. A current limitation to increasing the performance in the minimum rate region of the two curves is the bandwidth available on the Data Memory Bus [3]. Improving the data memory bus bandwidth would have the effect of shifting the minimum rate region of the two curves upward.

### 3.4. The Behavior of Indexed Kernels

When running on a single CE, the execution rate of a kernel will drop if one or more of the referenced vectors are addressed indirectly. This is mainly due to an increase in the number of vector instructions generated by the compiler for an indexed kernel (scatter/gather instructions, etc.). Moreover, the memory access pattern of the indexed vector elements might result in an appreciable decrease in the utilization of the bandwidth of the interleaved memory modules.

The execution rate of a multiprocessed vector kernel could degrade significantly if the result vector is addressed indirectly. This can be seen clearly in Figure 3 when comparing the execution rate curves for kernels ***vtv*** and ***vi=vtv***. Examing the assembly code generated for the two kernels reveals that while the ***vtv*** kernel is executed as a single concurrent vector loop, the concurrent loop of the ***vi=vtv*** kernel encloses two vector loops. Each CE first executes the vector statement ***temp←vtv***, where ***temp*** is a temporary vector. While $\mathrm{CE}_0$ continues by executing a second vector loop to scatter the contents of ***temp*** to the specified elements of the result vector, each $\mathrm{CE}_i$ (i=1,...,7) waits for a synchronization signal from $\mathrm{CE}_{i-1}$ indicating that it has finished scattering its results. In this fashion, any output dependence relationships between different instances of the original statement ***vi=vtv*** will be preserved. Figure 8 shows the execution scheme of this kernel using 8 CEs. This explains the lack of any speedup when this kernel is executed concurrently. Synchronization is not required in kernels with no potential output

[i] The speedup curves for the other kernels are found in [3].

[j] The overhead curves for the other kernels are found in [3].

dependencies. Kernels with output dependencies will gain speedup if the time spent evaluating the right hand side expression of the kernel is significantly larger than the time spent in the scattering loop by each CE. This is the case for the kernel ***vi=vipvtv*** where it is possible to attain a maximum speedup of 5.86 compared to 1.32 for the kernel ***vi=vtv***.

### Conclusion

Using the vector kernels of the LANL BMK8a1, this paper assesses the performance of the Alliant FX/8 multiprocessor for vector processing. One CE of the FX/8 shows the classical vector processor behavior where the performance increases as the vector length increases to a maximum which is maintained for larger vector lengths. However, because of the memory hierarchy, the vector performance of 1 CE falls to a rate dependent on the shared memory access speed when the cache size is not large enough for the referenced vectors to be cache resident. Although vector processing performance on 8 CEs shows the same rise and fall as vector length increases, parallel execution accentuates this behavior. First, the increase in execution rate is slower for short vectors due to the partitioning of the vectors across multiple processors and the multiprocessing overhead. Second, although speedups greater than 7 are achieved for most stride-1 kernels for vector lengths of $O(1000)$, the region of maximum performance is narrower than for 1 CE. Third, the speedup gain by increasing the number of computational elements beyond four is small for short ($O(100)$) and long ($O(10{,}000)$) vectors. The multiprocessing overhead exceeds 50% for short and long vectors for all kernels. It is less than 25% for most stride-1 kernels for vector lengths of $O(1000)$. Lastly, the performance of the FX/8 on stride-1 kernels can drop by a factor greater than 3 if the vectors are referenced from the shared memory.

The lower performance of the machine for kernels with indexed vectors is partially due to the increase in the number of vector instructions needed to execute the kernel. However, interprocessor synchronization to satisfy potential output dependencies is the major reason for performance degradation of indexed kernels.

The successful use of an 8 CEs FX/8 for vector processing depends on observing the principle of locality of reference [8]. This implies that the programmer (or optimizing compiler) should structure the code such that once certain sections of the program's data are resident in the cache, as much computation as possible is performed using this data before processing other sets of data [1], [2], [6], [18]. Some of the other factors that enhance the delivered performance are reducing the number of memory references in the vector statement, increasing the number of floating point operations performed using the same operands, and taking advantage of the chaining capabilities of the processors.

### References

[1] W. Abu-Sufah, D.J. Kuck, and D.H. Lawrie. *Automatic Program Transformations for Virtual Memory Computers.* **Proc. of the 1979 National Computer Conf.**, pp. 969-974, June 1979.

[2] W. Abu-Sufah, D.J. Kuck, and D.H. Lawrie. *On the Performance Enhancement of Paging Systems through Program Analysis and Transformations.* **IEEE Trans. on Computers**, Vol. C-30, No. 5, pp. 341-356, May 1981.

[3] W. Abu-Sufah and A.D. Malony. *Experimental Results for Vector Processing on the Alliant FX/8.* CSRD Report #539, Center for Supercomputing Research and Development, University of Illinois, Jan., 1986.

[4] W. Abu-Sufah. *On Modeling Vector Parallel Machines.* In preparation, 1986.

[5] Alliant Computer Systems Corporation. **FX/Series Product Summary**. June 1985

[6] W. Abu-Sufah, R. Lee, M. Malkawi, and P. Yew. *Experimental Results on the Paging Behavior of Numerical Programs.* **6th Intl. Conf. on Software Eng.**, pp.110-117, Sept. 1982.

[7] I.Y. Bucher and M.L. Simmons. *Performance Assessment of Supercomputers.* in **Vector and Parallel Processors: Architecture, Applications, and Performance Evaluation**, edited by Myron Ginsberg, North Holland, also Los Alamos National Laboratory Report **LA-UR-85-1505**, 1985.

[8] P.J. Denning. *Virtual Memory.* **Computing Surveys**, Vol. 2, No. 3, pp. 153-180, Sept. 1970.

[9] J.J. Dongarra and I.S. Duff. *Advanced Architecture Computers.* Mathematics and Computer Science Division, Tech. Memo #57,Argonne National Laboratory, Oct. 1985.

[10] J.J. Dongarra. *A Survey of High Performance Computers.* Proc. of Spring **COMPCON '86**, pp. 8-11, March 1986.

[11] H.P. Flatt. *Some Limitations of Parallel Processing.* invited presentation, Center for Supercomputing Research and Development, Univ. of Illinois, May 1986.

[12] M. Flynn. *Some Computer Organizations and their Effectiveness.* **IEEE Trans. on Computers**, Vol. C-21, No. 9, pp.948-960, Sept. 1972.

[13] D. Gajski, D. Gannon, J. Schwartz, and J. Browne. *Classification of Parallel Processor Architectures - Invited Tutorial Session.* **12th Int. Symp. on Computer Arch.**, June 1985.

[14] J.H. Griffin and M.L. Simmons. *Los Alamos National Laboratory Computer Benchmarking 1983.* Tech. Report LA-10151-MS, Los Alamos National Laboratory, June 1984.

[15] R.W. Hockney. *Performance of Parallel Computers.* in **High-Speed Computation**, edited by J.S. Kowalik, pp. 159-175, Springer-Verlag, Berlin, Germany, 1984.

[16] R.W. Hockney and C. Jesshope. **Parallel Computers**. Adam Hilger Ltd., Bristol, England, 1981.

[17] K. Hwang and F. Briggs. **Computer Architecture and Parallel Processing**. McGraw-Hill, New York, New York, 1984.

[18] W. Jalby and U. Meier. *Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy.* **Inter. Conf. on Parallel Proc.**, 1986.

[19] D. Kuck, A. Sameh, R. Cytron, A. Veidenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, M. Wolfe, C. Beckman, J. Davis, and C. Kruskal. *The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance.* **1984 Intl. Conf.on Parallel Processing**, pp. 129-138, Aug. 1984.

[20] D. Kuck. **The Structure of Computers and Computation**. John Wiley and Son, 1978.

[21] D. Kuck. *Automatic Program Restructuring for High-Speed Computation.* **Proc. of COMPAR81, Conf. on Analysis Problem-Classes and Programming for Parallel Computing**, edited by W. Handler, Nurnberg, Germany, 1981.

[22] K.Y. Lee, W. Abu-Sufah, and D. Kuck. *On Modeling Performance Degradation Due to Data Movement in Vector Machines.* **Proc. of the 1984 Conf. on Parallel Processing**, pp 269-277, August 21-24, 1984.

[23] O. Lubeck, J. Moore, and R. Mendez. *A Benchmark Comparison of Three Supercomputers: Fijitsu VP-200, Hitachi S810/20, and Cray X-MP/2.* **IEEE Computer**, pp. 10-24, Dec. 1985.

[24] F.H. McMahon. *L.L.N.L. Fortran Kernels: MFLOPS Program.* Lawrence Livermore National Laboratory, Dec. 1985.

[25] A. Sameh. *Numerical Algorithmss on the Cedar System.* **Second SIAM Conference on Parallel Processing**, invited presentation, Norfolk, Virginia, Nov. 1985.

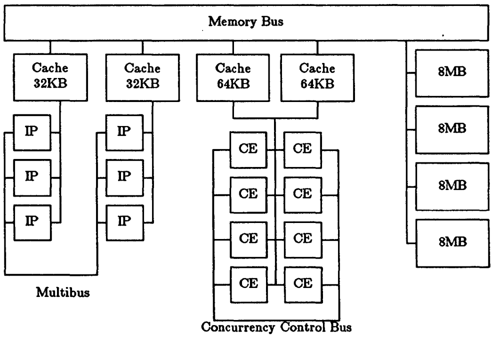


**Figure 1.**
**The Configuration of the Alliant FX/8 used in the Experiments**

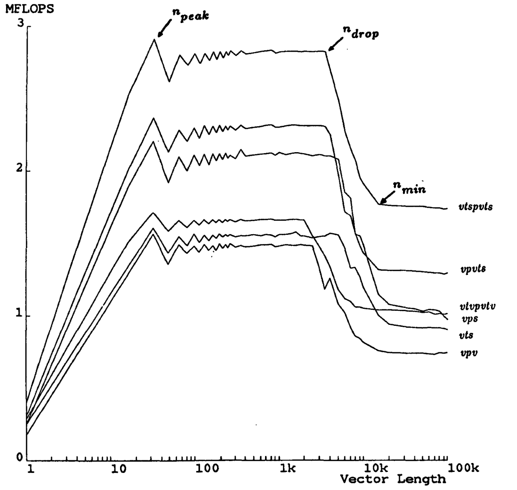


**Figure 2-a**
**The Maximum Delivered Performance for a 1 CE System**

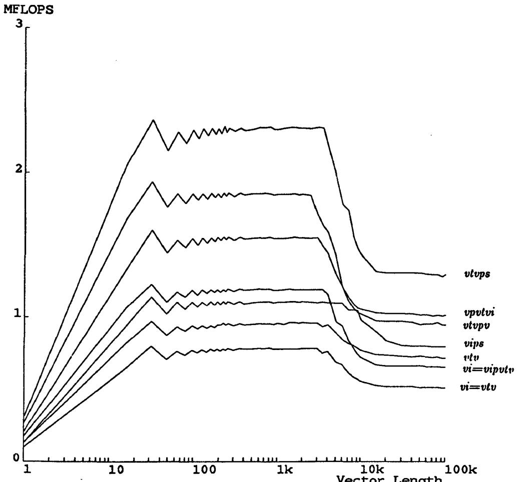


**Figure 2-b**
**The Maximum Delivered Performance for a 1 CE System (continued)**

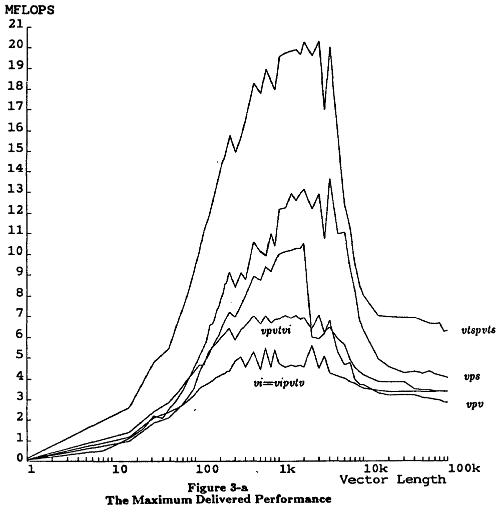


**Figure 3-a**
**The Maximum Delivered Performance for an 8 CE System**

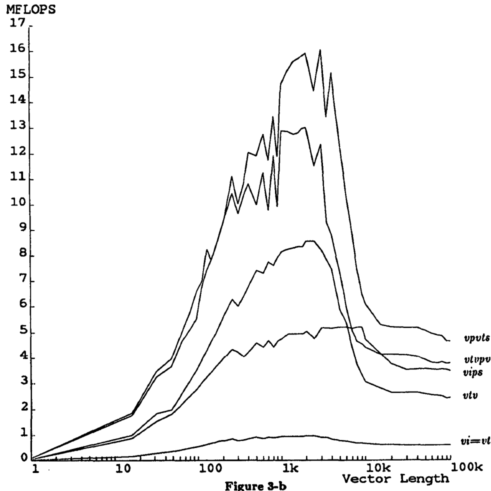


**Figure 3-b**
**The Maximum Delivered Performance for an 8 CE System (continued)**

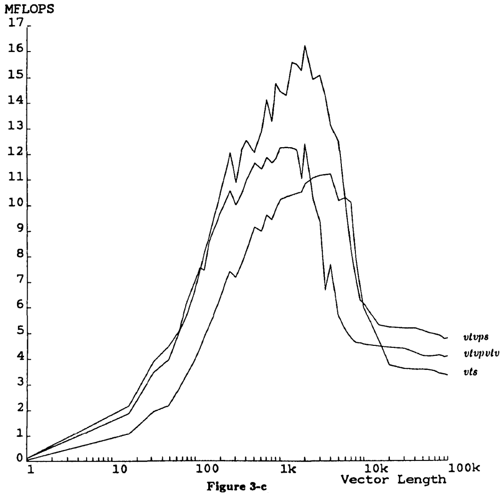


**Figure 3-c**
**The Maximum Delivered Performance**
**for an 8 CE System (continued)**

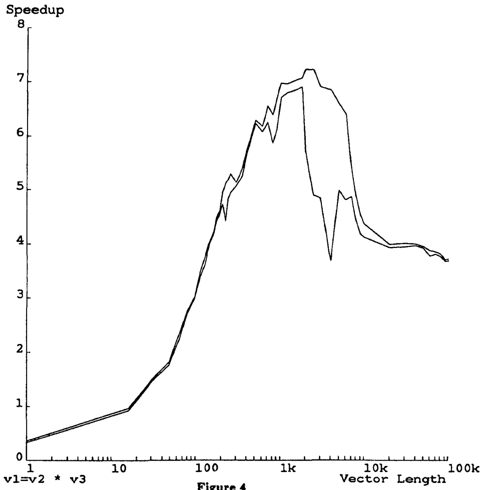


**Figure 4**
**The Speedup Upper/Lower Bounds**
**for Kernel *vtv***

Figure 5-a.
Speedup vs. #CEs
or Vector Lengths 100 and 1K

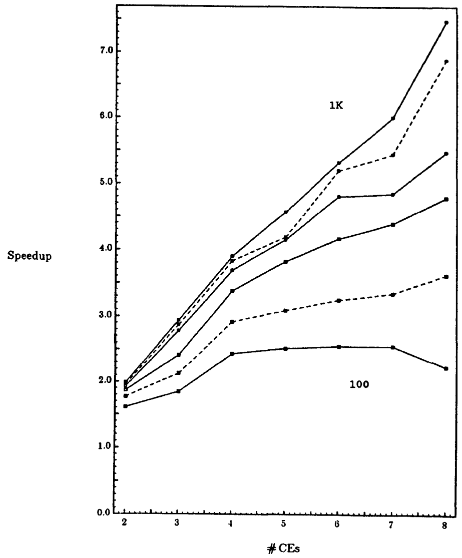
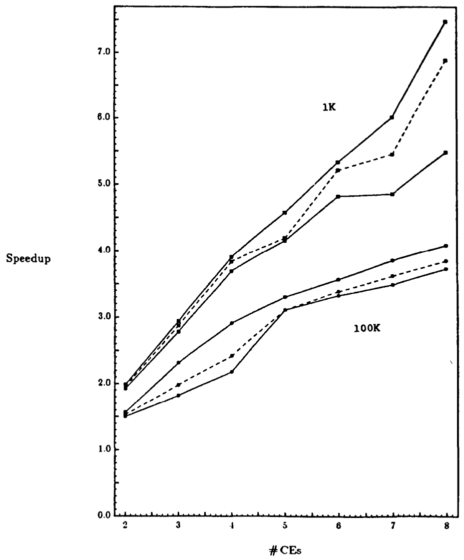


**Table 5-b.**
Speedup vs. #CEs
for Vector Lengths 1K and 100K



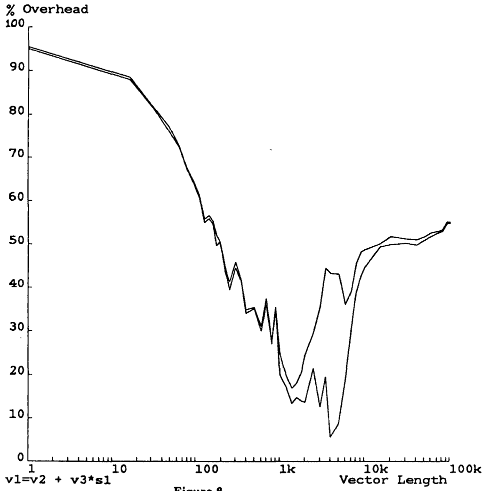


**Figure 6**
**The Percentage Multiprocessing Overhead**
**Upper/Lower Bounds for Kernel *vpvts***

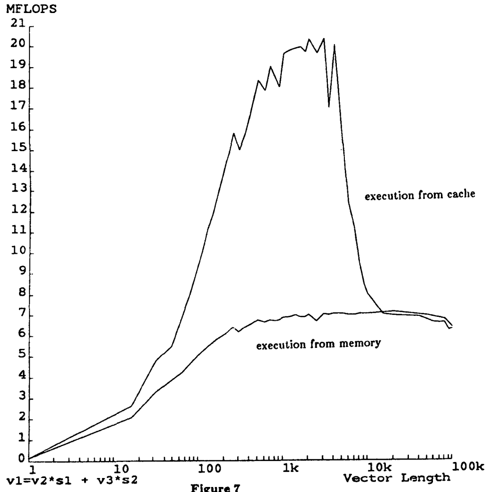


**Figure 7**
**The Execution Rates from Memory and Cache**
**for the Kernel *vtspvts***

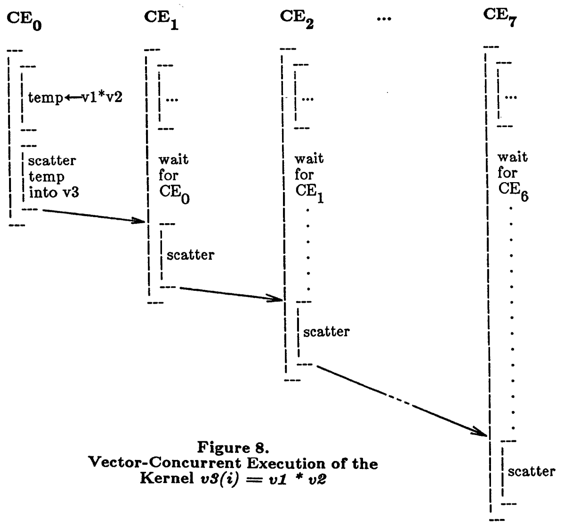


**Figure 8.**
**Vector-Concurrent Execution of the Kernel $v3(i) = v1 * v2$**

**Table 4**
**Maximum Upper Bound Speedups**

| Benchmark | Vector Length | Max. Upper Bound Speedup | Lower Bound |
|---|---|---|---|
| vps | 4000 | 6.67 | 5.13 |
| vts | 7000 | 8.15 | 5.27 |
| vpv | 2000 | 9.68 | 4.17 |
| vtv | 2000 | 7.22 | 5.73 |
| vtvps | 1400 | 8.05 | 6.04 |
| vpvts | 4000 | 7.56 | 4.54 |
| vtspvts | 2000 | 7.86 | 7.05 |
| vtvpv | 3000 | 7.84 | 4.90 |
| vtvpvtv | 2000 | 8.20 | 5.42 |
| vips | 7000 | 5.45 | 4.44 |
| vi=vtv | 3500 | 1.32 | 1.10 |
| vpvtvi | 500 | 6.08 | 4.53 |
| vi=vipvtv | 2500 | 5.86 | 5.20 |

**Table 5**
**Lower/Upper Bound on Speedups at Vector Lengths where Peak Execution Rates Occur**

| Benchmark | Vector Length | Speedup Bounds |
|---|---|---|
| vps | 4000 | 5.13 / 6.67 |
| vts | 4000 | 5.88 / 7.36 |
| vpv | 2000 | 4.17 / 9.68 |
| vtv | 2000 | 5.73 / 7.22 |
| vtvps | 2000 | 5.60 / 7.08 |
| vpvts | 3000 | 5.15 / 6.99 |
| vtspvts | 3000 | 6.04 / 7.21 |
| vtvpv | 2000 | 6.14 / 7.07 |
| vtvpvtv | 2000 | 5.42 / 8.20 |
| vips | 6000 | 4.15 / 4.83 |
| vi=vtv | 2500 | 1.08 / 1.24 |
| vpvtvi | 700 | 4.32 / 4.58 |
| vi=vipvtv | 2500 | 5.20 / 5.86 |

**Table 6**
**Minimum Upper Bound Multiprocessing Overheads**

| Benchmark | Vector Length | Min. Upper Bound Overhead | Lower Bound |
|---|---|---|---|
| vps | 1400 | 25.0% | 22.5% |
| vts | 3000 | 10.3% | 6.0% |
| vpv | 1400 | 16.0% | 13.3% |
| vtv | 1800 | 13.7% | 11.6% |
| vtvps | 1000 | 21.8% | 20.9% |
| vpvts | 1400 | 16.9% | 13.4% |
| vtspvts | 2000 | 11.9% | 1.7% |
| vtvpv | 1000 | 14.4% | 12.6% |
| vtvpvtv | 1200 | 8.6% | 7.2% |
| vips | 3000 | 41.4% | 40.9% |
| vi=vtv | 1200 | 84.8% | 84.5% |
| vpvtvi | 500 | 43.4% | 24.0% |
| vi=vipvtv | 700 | 29.2% | 27.3% |

**Table 1**
**Peak Execution Rates of Elementary Vector Operations**

| Benchmark | 1 CE | | | 8 CEs | | |
|---|---|---|---|---|---|---|
| | MFLOPS | Length | $E_m$ (%) | MFLOPS | Length | $E_m$ (%) |
| vps | 2.21 | 32 | 37.5 | 13.65 | 4000 | 28.9 |
| vts | 1.60 | 32 | 27.1 | 11.21 | 4000 | 23.8 |
| vpv | 1.56 | 32 | 26.4 | 10.52 | 2000 | 22.3 |
| vtv | 1.23 | 32 | 20.8 | 8.58 | 2000 | 18.2 |
| vtvps | 2.37 | 32 | 40.2 | 16.23 | 2000 | 34.4 |
| vpvts | 2.37 | 32 | 40.2 | 16.08 | 3000 | 34.1 |
| vtspvts | 2.91 | 32 | 49.3 | 20.33 | 3000 | 43.1 |
| vtvpv | 1.94 | 32 | 32.9 | 13.03 | 2000 | 27.6 |
| vtvpvtv | 1.71 | 32 | 29.0 | 12.40 | 2000 | 26.3 |
| vips | 1.14 | 32 | 19.3 | 5.22 | 6000 | 11.1 |
| vi=vtv | .80 | 32 | 13.6 | .97 | 2500 | 2.1 |
| vpvtvi | 1.60 | 32 | 27.1 | 7.07 | 700 | 15.0 |
| vi=vipvtv | .97 | 32 | 16.4 | 5.61 | 2500 | 11.9 |

**Table 7**
**Degradation of the Peak Execution Rates when Executing from Memory**

| Benchmark | 1 CE | 8 CEs |
|---|---|---|
| | Degradation Factor * | Degradation Factor * |
| vps | 2.26 | 3.36 |
| vts | 1.78 | 3.34 |
| vpv | 2.13 | 3.67 |
| vtv | 1.86 | 3.52 |
| vtvps | 1.85 | 3.33 |
| vpvts | 1.85 | 3.45 |
| vtspvts | 1.67 | 3.23 |
| vtvpv | 1.89 | 3.45 |
| vtvpvtv | 1.69 | 3.03 |
| vips | 1.43 | 1.49 |
| vi=vtv | 1.54 | 1.61 |
| vpvtvi | 1.69 | 2.08 |
| vi=vipvtv | 1.35 | 1.67 |

* degradation factor is calculated as the peak execution rate divided by the execution rate at 100K

**Table 2**
**Execution Rates at Vector Length 100K of Elementary Vector Operations**

| Benchmark | 1 Computational Processor | | 8 Computational Processors | |
|---|---|---|---|---|
| | MFLOPS | $E_m$ (%) | MFLOPS | $E_m$ (%) |
| vps | 0.98 | 16.6 | 4.06 | 8.6 |
| vts | 0.90 | 15.3 | 3.36 | 7.1 |
| vpv | 0.74 | 12.3 | 2.87 | 6.1 |
| vtv | 0.66 | 11.2 | 2.44 | 5.2 |
| vtvps | 1.29 | 21.9 | 4.80 | 10.2 |
| vpvts | 1.29 | 21.9 | 4.65 | 9.9 |
| vtspvts | 1.74 | 29.5 | 6.30 | 13.3 |
| vtvpv | 1.02 | 17.3 | 3.79 | 8.0 |
| vtvpvtv | 1.01 | 17.1 | 4.09 | 8.7 |
| vips | 0.80 | 13.5 | 3.49 | 7.4 |
| vi=vtv | 0.52 | 8.8 | 0.60 | 1.3 |
| vpvtvi | 0.95 | 16.1 | 3.37 | 7.1 |
| vi=vipvtv | 0.72 | 12.2 | 3.38 | 7.2 |

**Table 3**
**$n_{1/2}$ and the Percentage of Peak Execution Rate at $n_{1/2}$**

| Benchmark | $n_{1/2}$* | % of Peak |
|---|---|---|
| vps | 3 | 30% |
| vts | 3 | 35% |
| vpv | 3 | 33% |
| vtv | 3 | 36% |
| vtvps | 3 | 34% |
| vpvts | 3 | 33% |
| vtspvts | 3 | 35% |
| vtvpv | 2 | 25% |
| vtvpvtv | 2 | 31% |
| vips | 3 | 31% |
| vi=vtv | 4 | 36% |
| vpvtvi | 3 | 33% |
| vi=vipvtv | 4 | 40% |

* $n_{1/2}$ is calculated using a linear least-squares approximation of the vector execution times for vector lengths between 1 and 256.

# SUPRENUM: THE GERMAN SUPERCOMPUTER ARCHITECTURE - RATIONALE AND CONCEPTS

P.M. Behr, W.K. Giloi, H. Mühlenbein

Gesellschaft für Mathematik und Datenverarbeitung (GMD)
Birlinghoven and Berlin
W. Germany

Abstract -- Supercomputer architectures for numerical applications can be based on either of two major principles, SIMD vector machines or MIMD multicomputer systems. In the paper both solutions are compared in terms of performance, cost-effectiveness, and software problems involved, thus providing the rationale for the decision to develop a MIMD/SIMD multiprocessor supercomputer, configurable to up to 1024 nodes, where each node contains a floating-point vector processor. The crucial issues of such an architecture are the need for a bottleneck-free interconnection structure on the hardware side and for the appropriate program development environment on the software side. The solutions envisioned for the SUPRENUM MIMD/SIMD supercomputer presently under development are presented.

## The Choice of Supercomputer Architecture

In this paper the term "supercomputer" refers to very high performance machines for numerical applications; though our considerations to some extent hold true as well for non-numerical supercomputers. Supercomputers obtain their performance from two contributing factors. Firstly, they operate at the highest possible speed technology can provide. Secondly, additional performance is gained through a large amount of parallel processing.

An algorithm may be generally defined as a partial order of operations, the partial order being determined by the data dependencies between the operations. We call the parallelism of an algorithm explicit if the data dependencies are well-defined by the nature of the data types to be processed and, consequently, are known a priori. We call the parallelism implicit if it is not a priori known but must be determined through data dependence analysis.

What makes dataflow principle so attractive is its property to enable the machine to perform the data dependence analysis at run time and, thus, to provide a convenient way of exploiting implicit parallelism. Since implicit parallelism includes explicit parallelism, dataflow is a most general operational principle for parallel processing. The forte of the dataflow solution however, comes at the price of considerable overhead and, therefore, is not cost-effective in parallel processing applications where the simpler and more efficient SIMD or MIMD control scheme of handling explicit parallelism suffices.

Figure 1 presents a taxonomy relating the nature of parallelism to the appropriate control structures, processor structures, and communication structures.

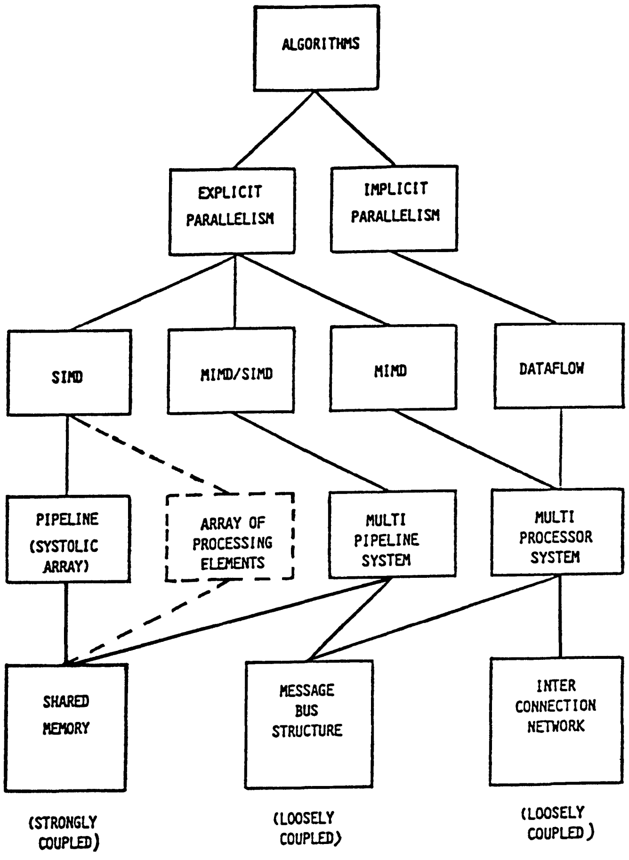


Figure 1 Taxonomy of parallel computer architectures

Figure 2 lists some of the major trade-offs between SIMD pipeline architectures and MIMD multiprocessor architectures. In the supercomputer domain, only the SIMD vector machine has found a larger market penetration. Recently, the first MIMD multiprocessor systems have become available in product form; their use at present still being more that of experimental systems rather than 'production machines'. However, two decades of research in innovative computer architecture have produced enough insight into all four forms of

0190-3918/86/0000/0567 $01.00 © 1986 IEEE

parallel architectures listed in Figure 1 to make it safe venturing the following statements.

1. SIMD machines consisting of an array of processing elements cannot compete in cost-effectiveness with SIMD pipeline machines, since they do not provide the parallel processing gain of a multi-stage pipeline processor.

2. In MIMD machines for numerical applications, the cost-effectiveness can be increased by as much as an order of magnitude through performing the floating point operations in each node in the vector mode rather than the scalar mode.

3. In the realm of numerical supercomputers, pure dataflow architectures have no chance to become a competitive solution; the reason being twofold. Firstly, since the parallelism in large numerical (array processing) problems is predominantly explicit and, thus, can be handled by the extremely efficient SIMD control scheme, the ability of the dataflow machine to handle implicit parallelism, which leads to its high control overhead, does not pay. Secondly, the pure dataflow scheme is based upon the notion that the firing of an operation consumes the tokens that are the "carriers" of the operands. Consequently, a value can be used as operand only once. This is counter-productive in all numerical algorithms in which values occur as operands of a number of operations. Both, the high control overhead and the inability of the pure dataflow scheme to handle data structure objects, result in an unfavorable MIPS/MFLOPS rate. By special measures (e.g., the introduction of the 'structure memory' in the manchester machine (Gürd, 1985), this rate can be much improved. However, it should not be overlooked that such measures in effect constitute a deviation from the pure dataflow scheme by introducing beneath the dataflow control level an SIMD control level for handling data structure objects. The performance gain obtained by such measure therefore is to be attributed to the SIMD execution of complex operations on data structure objects, while the dataflow control now synchronizes only the complex procedures and not the operations inside the procedures and, thus, becomes tolerable.

As conclusion of the discussion above we dare stating the following CONVERGENCE THEOREM: Numerical supercomputers of the future will be predominantly MIMD/SIMD machines.

The convergence will happen from the side of the vector machines by adding more and more pipelines; from the side of the MIMD architectures by pipelining the floating point coprocessor in the node. The SUPRENUM architecture is of the latter type.

SIMD PIPELINE ARCHITECTURE

PROCESSOR PERFORMANCE:

- SIMD GAIN
- PARALLEL PROCESSING GAIN OF MULTISTAGE PIPELINE FOR LARGE, ORDERED SETS OF DATA
- "SCALAR GAP"

PERFORMANCE LIMITATION DETERMINED BY:

- EFFECTIVE DATA MEMORY BANDWIDTH

POTENTIAL FOR PARALLEL PROCESSING:

- LOW TO MEDIUM

COST-EFFECTIVENESS:

- HIGHER

PROGRAMMING STYLE:

- PREDOMINANTLY FUNCTIONAL

CONFIGURABILITY AND FLEXIBILITY OF APPLICATION:

- LOW

MIMD MULTIPROCESSOR ARCHITECTURE

PROCESSOR PERFORMANCE:

- NO SIMD GAIN
- NO OR LITTLE PARALLEL OPERATION
- NO "SCALAR GAP"

PERFORMANCE LIMITATION DETERMINED BY:

- NODE INTERCONNECTION BANDWIDTH

POTENTIAL FOR PARALLEL PROCESSING:

- HIGH TO VERY HIGH

COST-EFFECTIVENESS:

- LOWER

PROGRAMMING STYLE:

- COOPERATING PROCESSES OR SINGLE ASSIGNMENT

CONFIGURABILITY AND FLEXIBILITY OF APPLICATION:

- HIGH

Figure 2 Trade-offs between SIMD and MIMD architectures

## The Choice of Technology

For the sake of discussion, two main technologies called "microcomputer technology" (MCT) and "main frame technology" (MFT) shall be characterized.

Microcomputer Technology:

- TTL or MOS, highly integrated low-power logic and memory (some $10^4$ to $10^5$ transistors per chip);

- predominant use of standard ("off-the-shelf") components, therefore lower design cost;
- low cost of packaging and cooling (forced air only);
- lower operating speed.

Main Frame Technology:

- ECL, less highly integrated high-power logic (some $10^3$ transistors per chip); ECL or CMOS memory;
- predominant use of custom gate array logic, therefore higher design cost;
- higher cost of packaging and cooling;
- higher operating speed.

Of course, the cost-effectiveness of a computer depends not only on the technology but also on the architecture. Two examples shall illustrate that fact.

(A) A microcomputer with floating-point coprocessor has about the same cost per FLOPS as the MFT based vector machine. Hence, the conclusion can be drawn that the higher cost-effectiveness of MCT compensates for the lower efficiency of the conventional architecture.

(B) Comparing MCT vector machines with MFT vector machines, the former exhibit a much higher cost-effectiveness. The reason why nevertheless MFT is the dominating supercomputer technology is the much higher absolute performance MFT may provide.

This discussion can be summarized by stating that a supercomputer development based on MCT is less costly and risky. However, if one wants to reach the same performance as with MFT, a higher degree of parallel processing is required to make up for the lower operating speed of MCT. Whether the MCT solution is more cost-effective than its MFT counterpart is a question that can be answered affirmatively only if the vector machine architecture is chosen. In the case of a MIMD machine one will be quite satisfied if the cost-effectiveness of a MFT-based vector machine can be met. In this case the MIMD machine is the better choice, for it is more flexible and does not exhibit the "scalar gap".

The relationship between the two technologies, MCT and MFT, and the two architectural forms, SIMD and MIMD, can be summarized as follows:

> MIMD multiprocessor systems with a larger number of nodes (as needed to achieve supercomputer performance) can be realized only in MCT.
>
> SIMD vector machines may be realized in both, MCT and MFT.

To conclude this section, we shall take a look at the absolute performance obtainable in either technology by contrasting the two cases, SIMD vector machine and MIMD multiprocessor system, respectively.

SIMD Vector Machine - MFT

As paradigm for maximal performance obtainable in ECL technology (MFT), we take the vector machines of CRAY Research Inc. (CRAY-1, CRAY-X MP, CRAY-2). The following table presents the major parameters of these machines.

| Model | Clock Time $T_p$ | Max. number of pipelines | Max. Performance | Year |
|---|---|---|---|---|
| CRAY-1 | 12.5 ns | 1 | 160 MFLOPS | 1977 |
| CRAY-X MP | 9.5 ns | 2 | 400 MFLOPS | 1983 |
| CRAY-2 | approx. 5 ns | 4 | 1000 MFLOPS | 1986 |

SIMD Vector Machine - MCT

Highly integrated dynamic memory allows a stream of up to 16 million words per second to be moved to and from the memory. Static CMOS memory has become available with an access time of down to 15 - 25 nanoseconds. The paradigm of a highly integrated floating-point pipeline processor is the Weitek WT 2264/2265 chip set (Weitek, 1986), which is designed to satisfy the following performance specifications:

- IEEE single precision operations (ADD, SUBTRACT, MULTIPLY): 20.0 MFLOPS
- IEEE double precision operations (ADD, SUBTRACT, MULTIPLY): 15.5 MFLOPS

The Weitek processors form a 7-stage pipeline. This, in connection with the pipelining of the data moves, results in a pipeline gain of about 10, a gain that is lost in the case of scalar operations. Chaining multiplication and addition (accumulation), e.g., for the inner vector product, doubles the vector mode performance. Several such pipeline processors could be cascaded to obtain a "macro pipeline" with a multiple of the performance of one processor set, provided the memory bandwidth is not already exhausted by one single pipeline.

MIMD Multiprocessor Machine - MCT

The maximal performance of an MIMD multiprocessor system with N nodes, obtained under most favorable conditions, is

$$P_{max} = q \cdot N \cdot P_n$$

if $P_n$ is the performance of a node and q is an overhead factor, $q < 1$.
The conditions under which $P_{max}$ can be reached are:

1. The algorithms executed by the machine must have a sufficiently high inherent parallelism in order to allow for the linear performance increase.

2. There exists no communication bottleneck in the system.

The first condition must be satisfied by the application; the second condition must be satisfied by the architecture. Violation of the conditions may result in a dramatic performance decrease.

It was pointed out above that SIMD machines provide more MFLOPS per cost unit, while MIMD machines provide a higher flexibility of use by virtue of the fact that parallel processing is not restricted to vectors. Hence, the decision between the two architectural principles involves a tradeoff between the higher cost-effectiveness of SIMD and the higher flexibility of MIMD. However, the MIMD flexibility exists only if it is not unduly constraint by the system's communication structure. For example, a communication structure that allows each node to communicate only with its nearest neighbors would be such a constraint and, thus, handicap the MIMD machine's competitiveness in comparison to the SIMD machine. Consequently, the communication structure of a general purpose MIMD multiprocessor architecture should provide total internode connectivity.

### MIMD/SIMD Machine - MCT

Going after the order of magnitude in node performance gain by "vectorizing" the node operations is a temptation hard to resist to. The result is an MIMD/SIMD architecture, i.e. a MIMD multiprocessor architecture in which each node contains a floating point pipeline processor. To preserve the flexibility of the MIMD approach, the nodes of such a machine must be connected through a communication structure with a very high communication bandwidth. I.e., a certain proportion must be maintained between node performance (in MFLOPS) and communication bandwidth (in MBytes), the factor of proportionality being application dependent. If this condition is satisfied, than vectorizing the nodes pays even in non-vector applications such a multigrid PDE solvers. In this case, even the small sets of data that must be interchanged between grid point can be treated as (small) vectors and, consequently, lead to a vector gain.

It must be mentioned that by attempting to combine the advantages of SIMD and MIMD, one also combines the software complexity of both approaches, namely the need for either a vectorizing compiler (SIMD) or a vector language (e.g. FORTRAN 8X).

## Communication Structures in Parallel Computers

### SIMD Vector Machines

SIMD vector machines process operand data streams flowing from a storage to the pipeline processor, thereby producing result data streams flowing back to the storage. Thus their performance is determined either by the memory bandwidth or the processor bandwidth limitation, whichever comes first, and no further communication bottleneck exists. The same holds true for multipipeline machines in which the tasks running on the different processors are only "loosely coupled" (i.e., in which there is little data dependency,between the tasks).

### MIMD Multiprocessor Systems

To overcome the memory bandwidth limitation problem, each processor of a MIMD multiprocessor system must have its private memory. Such a processor-memory combination usually is called a "node". This approach puts the emphasis on the problem of providing an adequate interconnection structure (IS) to handle the internode communication.

An adequate IS should satisfy the following conditions:

(1) It must provide total connectivity to maintain the potential flexibility of a MIMD machine.

(2) It must exhibit a sufficiently high bandwidth to avoid communication bottlenecks.

(3) It must be technically and economically feasible.

(4) It must be highly reliable.

### The Problem With Interconnection Networks

Interconnection networks (IN) that are capable of connecting a number of source nodes with a number of destination nodes are considered by many as the solution to the interconnection problem in large scale SIMD multiprocessor systems. Many papers have been published dealing with such structures, their complexity, and their interconnection properties. Hardly any of these many papers is addressing the topic of the technical feasibility and the interconnection bandwidth obtainable in view of such mundane parameters as pin limitation, packaging problems, driving power limitation, cost, etc.

INs come in two major categories: single-stage (permutation) and multi-stage networks. In single-stage networks, a data packet may have to travel through the network several times in order to reach a given destination from a given source. In contrast, in a multi-stage network the data can flow directly from the source to the destination. Another distinction is that between circuit switching, where a physical connection is provided from source to destination, and packet switching, where a logical connection is provided for a packet travelling through the network. Furthermore, control may be centralized or decentralized.

If one takes a closer look, the seemingly large variety of INs proposed can be identified as variants of one of the 4 basic classes listed in the following table (Ermel, 1985).

| CLASS | NETWORK TYPE | COMPLEXITY |
|---|---|---|
| A | CROSSBAR SWITCH | 0 $(n^2)$ |
| B | BENES NETWORK | 0 (n·log n + n/2) |
| C | N-CUBE, BASELINE, BANYARD, OMEGA, FLIP, DELTA | 0 ((n/2) log n) |
| D | DATA MANIPULATOR, INVERSE DATA MANIPULATOR | 0 (n + n log n) |

Networks with (N·log N)-complexity are not suitable for circuit switching, since they exhibit at any time a large incidence of mutual blockages of data paths. This leaves the crossbar switch, which provides total point-to-point connectivity, as the only viable solution for a circuit switching network. Packet switching networks, on the other hand, can readily deal with blockages, since the nodes have the capability of storing and keeping a packet for the duration of a blockage.

The switching elements of a circuit switching crossbar network are extremely simple (simple bus connection by tri-state logic). In addition, some hardware is needed to arbitrate the access to the crossbar busses. The switching elements of packet switching networks, on the other hand, must be intelligent communication processors that are orders of magnitude more complex than the simple switching elements of a circuit switching network. By the same token, they are also much slower, as they have to execute a complex microprogram to handle a packet. Therefore, if one compares only the switching complexity of different networks, one may be comparing apples with oranges.

A more detailed feasibility study conducted by Ermel (1986) has shown that the size of a crossbar network should not exceed that of a 32 x 32 switching matrix in order to be packagable in a reasonable manner; and 64 x 64 would absolutely be the technical limit. This means that the direct interconnection of, say, 256 or more nodes through a crossbar switching network is technically not feasible. Packet switching networks for the interconnection of an equally large number of nodes, even if still feasible in terms of pin limitation and packaging constraints, would be too slow for supercomputers.

The way out of this dilemma is a two-stage approach by forming clusters of nodes and connecting these clusters either via a crossbar network or an equally fast packet switching network. The interconnection of the nodes of a cluster may be performed through either a very high-speed cluster bus or, again, a crossbar network.

## Software Problems

Historically the parallel processing user has been a rather knowledgeable scientist or engineer who is willing to assume the burden of creating application programs using rudimentary environments. Detailed architectural and operating system knowledge as well as the intricate ability to manually map parallel algorithms onto a virtual parallel architecture are some of the hurdles such users had to overcome.

### Programming of Single-Pipeline Vector Machines

The programming of single pipeline vector machines usually is carried out in FORTRAN. A number of vectorizing compilers have been developed which map the inner loops of conventional, sequential FORTRAN programs onto the vector operations of the machine. The conditions under which such a mapping is possible and the techniques involved have become a well-understood topic.

### Programming of Multi-Pipeline Vector Machines

Presently, multi-pipeline vector machines are programmed in a multitasking manner. Typically, a task is a part of a program that can be run in parallel with some other parts of the program. The work to be done is partitioned into at least as many tasks (processes) as there are pipelines. The system then maintains a task queue, to which an unoccupied pipeline can go in order to find a task to execute. This "task attraction" scheme works only in strongly coupled systems, i.e. systems with shared logical memory.

The primary tools for multitasking can be classified into three categories. They are: task creation, critical sections monitoring, and task synchronization. The most advanced tools can be found in the Denelco HEP system, whereas for example the Cray computers offer only low-level constructs such as LOCK and EVENT variables. The following little example demonstrates some of the problems:

```
SERIAL PROGRAM:     DO 1  I=1, NPIPES
                    1 CALL COMPUTES (I)
PARALLEL PROGRAM:   DO 1  I=1, NPIPES
                    1 CALL TASKSTART
                    (ID, COMPUTES, I)
```

The two programs do not produce the same results. The reason is that FORTRAN calls are made by reference and the main task with the I-loop and the other tasks are running asynchronously.

Despite this and other classical "multitasking" problems the simple "task partitioning" approach works on the presently existing vector machines with a small number of pipelines. The application programmer is assumed to be a knowlegdeable system programmer capable of using the low-level multitasking tools.

This approach will not be acceptable when the number of pipelines becomes larger. The above mentioned low-level multitasking tools are fare too error prone for problems of higher complexity.

### Experiences with MIMD Multi-Processor Systems

The experiences with multitasking programming so far gained were made by system programmers and can be summarized as follows:

(1) The behavior of even short parallel programs may be astonishingly complex.

(2) Tracking down a bug in a parallel program can be exceedingly difficult. This results from the combination of logical complexity, nonrepeatable behavior, and the lack of existing tools.

At best, only partial solutions can be expected. The following postulates can be given:

(1) Whenever possible, rewrite at least the "skeleton" of the parallel program from scratch.

(2) Given the great difficulty of finding bugs, much greater emphasis must be placed on writing code which is correct from the beginning.

(3) Use high-level synchronization policies wherever possible and hide (encapsulate) them as much as possible.

## Programming of MIMD Multiprocessor Systems

The problem of programming a MIMD multiprocessor system with a very large number of nodes has been attacked only lately. Realistically, we see four different ways how applications software may be written during the next 5 - 10 years. Each of them currently has some severe drawbacks that limit their usefulness. McGraw/Axelrod (1984) discuss the following four approaches:

(1) Extend existing languages, like FORTRAN, with new operations that allow users to express concurrency and synchronization.

(2) Extend existing compilers to identify concurrent operations wherever it can and insert the necessary synchronization (automatic parallelization).

(3) Add a new "language layer" on top of an existing language that describes the multitasking and the desired concurrency, while allowing the basic applications program to remain "relatively" unaltered (metalanguage approach).

(4) Integrate new languages and appropriate compilers which incorporate the concepts of concurrency and synchronization (e.g., a very high object-oriented and process-oriented procedural language of a functional language).

These four ways do not represent totally orthogonal approaches, and therefore there can be no hard lines drawn between them. All of them involve some amount of language and compiler alteration.

## SUPRENUM-1: A MIMD Supercomputer

### Rationale for SUPRENUM-1

SUPRENUM-1 is a MIMD/SIMD supercomputer development project funded by the Ministry of Research and Technology (BMFT) of the German Federal Government. To carry out the project a task force has been formed consisting of several research institutions and companies. Specifically the Gesellschaft für Mathematik und Datenverarbeitung (GMD) is involved in the following tasks:

- development of a first prototype hardware and system software (in close cooperation with the participating industry)
- development of specific program development environments
- development of application software, e.g., multigrid partial differential equation solvers (Trottenberg, 1984).

The rationale for the decision to build a MIMD/SIMD multiprocessor system rather than a super fast SIMD vector machine is multifold:

(1) The market for SIMD vector machines is highly competitive, whereas the market for MIMD/SIMD supercomputers is just beginning to evolve.

(2) The higher flexibility of the MIMD principle allows for a broader application application spectrum.

(3) The "scalar gap" problem of SIMD vector machines is mitigated in MIMD/SIMD multiprocessor systems.

(4) It is expected that the MCT-based MIMD/SIMD multiprocessor system will eventually become more cost-effective than the MFT-based vector machine of comparable performance.

(5) Given the appropriate hardware solution, the development cost of the MCT-based multiprocessor system is lower than would be the cost of developing a MFT-based vector machine.

(6) The MIMD multiprocessor system development will produce technological spin-offs that will benefit a whole spectrum of products (such a general spin-off effect could not be expected from the more specialized pipeline machine development).

The SUPRENUM-1 research and development project is product-oriented. This means that, in contrast to pure research, additional market-oriented requirements must be satisfied such as:

- the desired absolute performance must be obtained at competitive cost-effectiveness;
- the program development environment provided must find user acceptance;
- the machine must be manufacturable, testable, and maintainable;
- there exists a time window during which the research and development project must lead to a production model.

In order to meet the time window requirement, the architectural design of SUPRENUM-1 will be based upon already proven yet highly innovative concepts and solutions. In this sense, the SUPRENUM-1 will be based upon already proven yet highly innovative concepts and solutions. In this sense, the SUPRENUM-1 development is influenced in many ways by the solutions and experiences gained by the preceding UPPER-project (Behr/Giloi, 1984).

## SUPRENUM-1 Hardware Architecture

In order to obtain a manageable, bottleneck-free interconnection structure, SUPRENUM-1 has a hierarchical hardware structure consisting of nodes, clusters, and hyperclusters.

As indicated in Figure 3, the basic processing node of SUPRENUM-1 is a single board computer consisting of the following major components:

- powerful 32-bit ('front end') microprocessor, to function as program execution machine as well as scalar processor, in connection with 8 MByte of high-speed dynamic memory and an objectoriented memory management unit;
- powerful floating-point vector processor (MFLOPS, IEEE standard single and double precision), performing a variety of complex numerical operations under microprogram control;
- microprogram controlled dedicated communication coprocessor, to support the object-oriented message passing and to allow the objects of application-specific data types to be copied at high-speed from node to node.

Each node has a local operating system whose task is to boot-up the node on power-on, manage and schedule the processes in the node, exchange communication objects with other nodes, manage the local memory, and perform comprehensive self-testing and fault diagnosis routines.

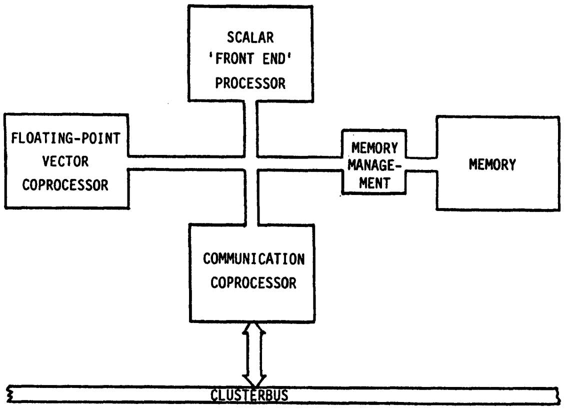


Figure 3 Structure of a node

Figure 4 depicts a block diagram of the cluster. The cluster contains up to 16 nodes plus a disk controller for local disks, the diagnosis node, and the communication unit for inter cluster communication. A cluster is accommodated in one 19" rack. The processors of the cluster communicate via an ultra-fast, duplicated parallel backplane-bus, which allows several communication partners to exchange messages simultaneously. The overall communication bandwidth of the cluster bus is 256 MByte/s a value that can hardly ever been exhausted.

Smaller systems may consist of one hypercluster ring, comprising up to 4 clusters (64 nodes) which are interconnected by one slotted ring bus (modified UPPERBUS, Zuber 1984) with a transmission bandwidth of 560 MBit/s.

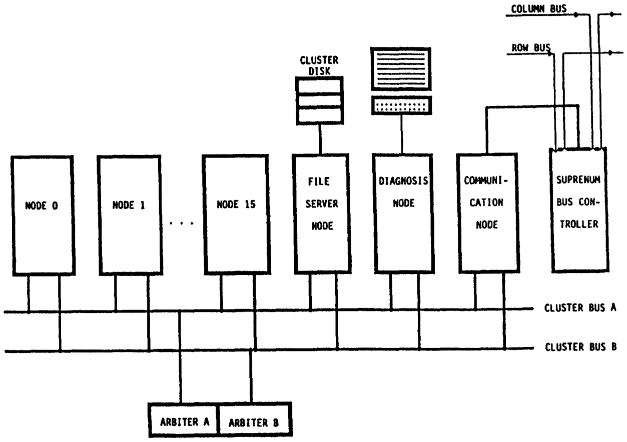


Figure 4 Structure of a cluster

Larger systems are structured in the form of a matrix of clusters whose rows and columns, respectively, are hypercluster rings. That is, each cluster belongs to two hyperclusters, row and column. The hypercluster bus is not fault-tolerant itself; rather, fault tolerance is achieved by alternate routing in the matrix. The hypercluster bus controller provides a gateway between the row and column each cluster belongs to, and it is intelligent enough to handle the alternate routing task arising in the case of bus transmission faults. The bus controller takes care of the protocol hierarchy that regulates the exchange of packets or larger logical entities, formed by a number of packets, via the slotted ring bus.

The SUPRENUM-1 system includes a separate operating system machine, whose tasks are to manage the global system resources such as global disk storage, take care of the workload distribution and system initialization, and control the recovery procedures required in the case of fault detection. The operating system machine, however, is not involved in the actual program execution. Program execution is strictly handled by the collective of local node operating systems. In addition to the operating system machine, SUPRENUM-1 contains a dedicated diagnosis and maintenance machine. Moreover, additional 'peripheral computers' can be added such as: one or more program development machines and special graphic processors for graphical representation of results. Note that these additional machines can be arbitrarily inserted into any of the rings of the orthogonal network of ring busses and that it is specifically the (ultra high bandwidth) ring bus structure that allows for the wide-range configurability of the system.

Figure 5 illustrates the matrix structure of a SUPRENUM-1 configuration with 256 nodes and a maximum processing power of 4 GFLOPS.

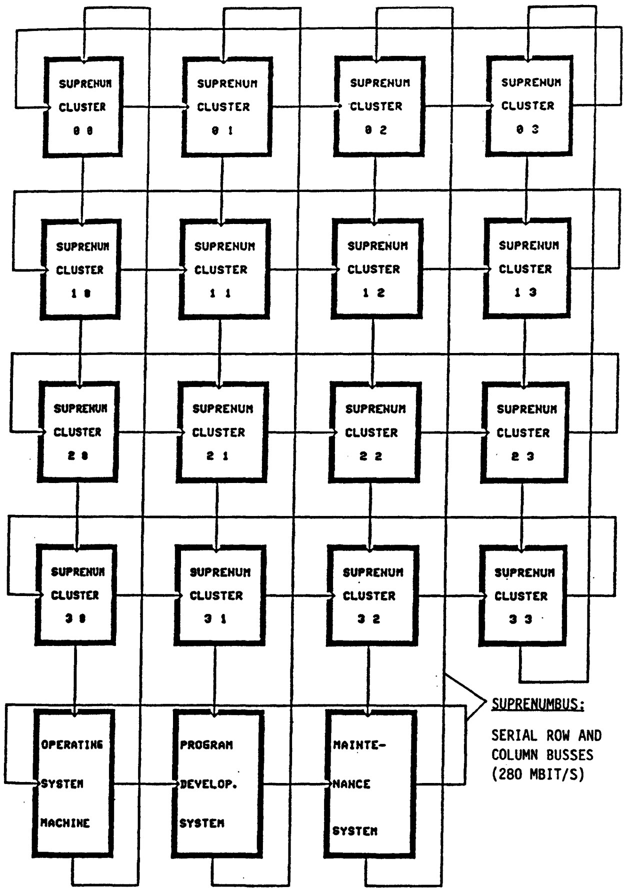


Figure 5 SUPRENUM System With 16 Clusters (256 Nodes)

## SUPRENUM-1 Software Architecture

### The SUPRENUM Software Environment

The SUPRENUM software system forms a hierarchy consisting of:

- 'firmware' (PROMed software) and software for node monitoring, process management, and inter process communication
- operating system machine software providing the language interface as well as the central database management
- appropriate parallel processing languages
- program constructor
- application constructor.

Every layer provides services to the upper layers. The language layer will be based upon the services of an abstract SUPRENUM machine.

The abstract SUPRENUM machine specifies the run-time environment the (distributed) 'global operating system' (collective of node operating systems) will support. Its basic features can be characterized as follows:

- The logical entity of computation is the process.
- The logical entity of communication is the communication object.
- Only communication objects can be shared by several processes; all other objects are local to the process who owns it.
- A structure is provided for aggregating processes with communication objects. This structure is called a task.
- The task provides the scope of protection.
- Multiple users may be given partitioned access to the machine.

The following basic features of the abstract SUPRENUM machine will be reflected in the programming languages:

- dynamic creation of explicit processes;
- asynchronous inter process communication (IPC);
- messages received at a single process entry point.

Asynchronous communication through messages received at multiple entry points have been explored by Behr/Giloi (1984) and Mühlenbein/Warhaut (1985). The approach is reflected in appropriate extensions of PASCAL (Hänisch, 1984) and MODULA 2 (Warhaut, 1986). In the SUPRENUM system, the languages extended for program decomposition into cooperating processes with data-driven synchronization will be FORTRAN and MODULA 2.

### The SUPRENUM Program Constructor

The program constructor consists of language-specific syntactic editors and interpreters. The constructor is based on the programming system generator "PSG" developed by Bahlke/Snelting (1985). The constructor can deal with incompletely specified program templates, called fragments, and comprises a 'hybrid' editor, which accepts textual input and/or prompted input from an abstract syntax tree.

Program fragments called program templates will be implemented for specific types of generic interprocess communication. One example is a program template for a ring process structure, where every process sends messages only to its left and right neighbor. Another example is the 2D-mesh, where each process can send messages to its four neighbors, etc.

Other components of the programming environments are run-time debuggers of different flavors. For example, language interpreters may be used for debugging small modules, simulators may be used for debugging larger programs, and a performance monitor may be used for performance-related debugging on the physical SUPRENUM machine.

### The SUPRENUM Application Constructor

The SUPRENUM Application Constructor will allow users who do not want to do low-level programming to implement applications by using existing application software packages.

The development of the SUPRENUM Application Constructor is a very research-oriented topic. The constructor will have to contain specialized application languages for restricted application domains, as well as a knowledge base for guiding the user in the task of generating applications by some heuristics.

First examples will be a very high level language for specifying the resolution of partial differential equation solving and an expert system for creating the appropriate multigrid programs.

## References

Bahlke/Snelting (1985)

R. Bahlke, G. Snelting: "The PSG Programming System Generator", ACM SIGPLAN Notices 20, 7 (July 1985)

Behr/Giloi (1984)

P.M. Behr, W.K. Giloi: "Obtaining a Secure, Fault-Tolerant, Distributed System With Maximized Performance", in G. Reijns (ed.): Hardware Supported Implementation of Concurrent Languages in Distributed Systems, North Holland, Amsterdam 1984

Ermel (1985)

W. Ermel: "Untersuchungen zur technischen Realisierbarkeit von Verbindungsnetzwerken für Multicomputer-Architekturen", Ph.D., Technical University of Berlin, 1985

Ermel (1986)

W. Ermel: "Vorschlag zum Aufbau eines 32x32 Crossbar Networks", GMD FIRST Internal Report 1986

Gurd (1985)

J.R. Gurd, C.C. Kirkham, I. Watson: "The Manchester Prototype Dataflow Computer", CACM 28, 1 (January 1985)

Hänisch (1984)

R. Hänisch: "Parallel Processing PASCAL Manual", GMD FIRST 1984

McGraw/Axelrod (1984)

J. McGraw, T.S. Axelrod: "Exploiting Multiprocessors - Issues and Options", Lawrence Livermoore Preprint UCRL-91734, 1984

Mühlenbein/Warhaut (1985)

H. Mühlenbein, S. Warhaut: "Concurrent Multigrid in an Object-Oriented Environment", in Degroot D. (ed.) Proc. 1985 ICPP

Trottenberg (1984)

U. Trottenberg (ed.): "Rechnerarchitekturen für numerische Simulation auf der Basis superschneller Lösungsverfahren", GMD Studien Nr. 88, Sept. 1984

Warhaut (1986)

S. Warhaut: "Concurrent MODULA 2 Specification", GMD Report 1986

Weitek (1986)

Weitek Corp.: "WTL 2264/ WTL 2265 Preliminary Data", Sunnyvale, CA, 1986

Zuber (1985)

G. Zuber: "UPPERBUS - A Bit-Serial Computer Interconnection Bus With 280/560 Mbps", GMD FIRST Technical Report 12, 1985

# DEPENDABILITY EVALUATION OF MULTICOMPUTER NETWORKS [†]

L. N. Bhuyan and C. R. Das

The Center for Advanced Computer Studies
University of Southwestern Louisiana
P. O. Box 44330
Lafayette, LA 70504

## ABSTRACT

Reliability and availability are two common measures used to evaluate the dependability of computer systems. In this paper we analyze the reliability and Computation-Communication Availability of multicomputer networks for multiple faults with and without repair. Simulation models are developed based on task requirements, graceful degradation and computation and communication capability of the system. The effect of component failure rate and repair rate on the dependability of the multicomputers are also presented. The model accepts the adjacency matrix of a multicomputer graph as the input and hence is suitable for all types of networks. Typical results are presented for 16-node loop, complete connection, hypercube, mesh and tree structures on a comparative basis.

## 1. INTRODUCTION

Based on interconnections, parallel and distributed computer architectures can be broadly divided into two categories, namely: multiprocessors and multicomputers[1]. A multiprocessor consists of a large number of processors connected to a number of memory modules through a circuit switched interconnection network. In a multicomputer, however, each processor has its own private memory and message/packet switching is used for communication between the processors (nodes). Reliability issues of multiprocessor systems were presented in [2, 3]. This paper is concerned with those issues of multicomputer systems. Multicomputers are also sometimes referred to as computer networks. Normally the former is used in conjunction with centrally located systems while the later is used in case of geographically distributed computers. We will, however, use these two terms without distinction. We will also use processors and nodes interchangeably.

Several structures such as loops, trees, full connection and hypercubes, etc. [4-6] have been proposed to interconnect a network of computers in a message/packet switching environment. Each structure possesses some unique advantages and disadvantages compared to another. For example, a bi-directional single loop (ring) structure has only two I/O ports per node, but the diameter (maximum number of hops between any two nodes along the shortest path) is $\frac{N}{2}$ in a system with $N$ nodes. Any two non-adjacent faulty nodes or links disconnect the loop. On the other hand a completely connected structure has ($N$-1) I/O ports per node, but the distance between any two nodes is unity. The structure is highly fault tolerant, but due to its high cost the structure is unsuitable for systems with large number of nodes. The cost and performance of other structures such as chordal ring, tree and hypercube, etc. lie between these two [4-7]. The performance of the multicomputers is usually measured in terms of the average distance between the nodes and the average traffic density on a link. These measures usually decide the average queueing delay or the saturating load of a computer network [5-7].

All these performance evaluations assume that the components of the systems are fault free. In a real situation, however, the nodes and links will fail randomly depending on their failure rates. Hence, certain dependability measures like, reliability, availability etc. [8, 9] of these multicomputer systems are also important. These dependability measures can also be quantified based on the performance requirement at the user level or system level [11-13]. For a degradable multicomputer one measure is the *task based dependability* . In this case the failures are tolerated as long as the minimum number of nodes for the execution of a task are available and the reliability and availability of a system are computed based on this task requirement. The following example clarifies this idea.

*Example-1*

Consider a bidirectional single loop structure with 16 nodes where a node and a link have failed in (0, $t$ ). Assuming a task requires 12 nodes for computation, the structure in Fig. 1.a is good enough, but the structure in Fig. 1.b is not. The former has a linear array of 13 nodes sufficient for computation while the later has two arrays of 8 and 7 nodes both of which are insufficient for the task.

The reliability at time $t$ is defined as the probability that the system is operational during (0, $t$ ) [10]. The aim of this paper is to compare the reliabilities of different multicomputer structures under similar task requirements. From *example* 1, it is clear that the reliability not only depends on the number of faults during (0, $t$ ), but it heavily depends on the exact location of the faults which is random in nature. The situation becomes more and more complicated with progress of time, not to mention other difficult structures such as trees, meshes and hypercubes, etc. Hence it is unlikely that an analytical solution to compute the reliability exists; we therefore resort to simulation.

The reliability, as discussed above, does not reveal the performance degradation due to failure of various components in (0, $t$ ). The computation availability ($CA$) of a gracefully degrading multicomputer is defined as the expected value of available computation at time $t$ [9]. This is directly proportional to the number of processors (nodes) operational at time $t$. Many other similar measures have been suggested by different authors to capture the dependability of

[†] This research was partly supported by the NSF grant no. DMC-8513041.

0190-3918/86/0000/0576 $01.00 © 1986 IEEE

degradable computer systems [13, 14]. We develop an availability model for computation and communication as described in the next section.

We will consider only hard failure of the nodes and links and compare the fault-tolerant properties of various multicomputers for both repairable and non-repairable conditions. By non-repairable we mean that there is no on-line maintenance facility available to repair/replace the failed components. The failed elements are detected and isolated by the maintenance processor and are repaired only when the system goes to the failed mode. A more optimistic situation is to consider on-line repair which means a component is repaired/replaced whenever it fails. In this case the system dependability is greatly influenced by the repair rate. The effect of the failure rate and repair rate on the dependability of the systems are also analyzed.

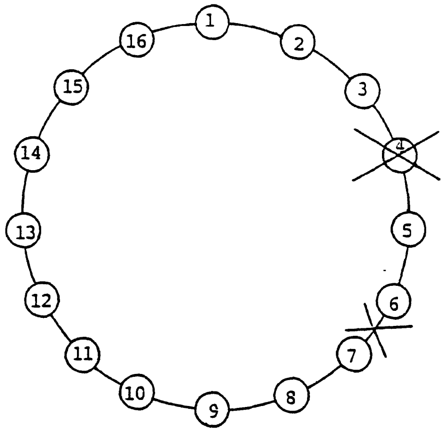


Fig. 1.a - A 16 node ring network after node 4 and link 6 have failed.

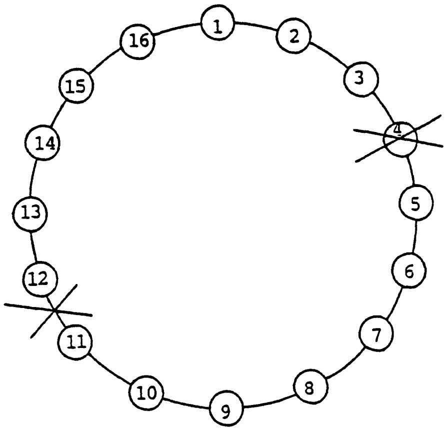


Fig. 1.b - A 16 node ring network after node 4 and link 11 have failed.

## 2. COMPUTATION AND COMMUNICATION

The most critical issues in the design of a distributed system are concerned with interprocess and interprocessor communication. It has been widely recognized that communication is at least as important as computation [15]. Hence, the communication capability of the degraded multicomputer must be considered with an equal weight taking into account both the node and link failures. It is also important to note that the component failures may bring in a structural change of the system. The loop structure of *example 1* , illustrated before, reduces to linear arrays after a node and a link failures. This structural change will result in a significant degradation in the communication ability of the network. Therefore, the previous availability models [9] do not represent the system performance adequately. Cherkassy et. al. [16] have considered both node and link failures in a tree network. However, because of the analytical complexity, they restrict their studies to single failures. Recently Tsuchiya [17] has presented a mathematical model for the availability analysis of distributed networks. But he does not consider performance degradation due to node and link failures. In this paper, we will study performance degradation with multiple failures and that too for various types of networks. Naturally, simulation seems to be the only way to accomplish this formidable task.

First, we have to determine a suitable measure for the computation and communication in a multicomputer system. The computation capability of a network is directly proportional to the number of operational nodes. The communication capability is usually measured in terms of the average delay of a message between a source and a destination. Under the usual assumptions of uniform traffic generation, fixed routing, etc. [5, 7], the delay is proportional to $\frac{\bar{d}}{1-\rho\bar{d}}$ [5], where $\bar{d}$ is the average number of hops a message passes through its route and $\rho$ is the utilization factor. Here $\rho = \frac{\gamma}{\mu C}$, where $\gamma$ is throughput in messages/sec, $\frac{1}{\mu}$ is the average message length in bits/message and $C$ is the total capacity of the network in bits/sec. When $\rho$ approaches $\frac{1}{\bar{d}}$ , the delay approaches infinity corresponding to a saturation traffic of $\gamma_{sat}$ . Then $\gamma_{sat} = \frac{\mu C}{\bar{d}}$ truly represents the communication capacity of a network. Here $\mu$ being a constant and $C$ being directly proportional to the number of links ($L$ ), $\gamma_{sat} = k_1\frac{L}{\bar{d}}$ for some constant $k_1$ . Although this is a good measure for the communication capability of the links, the number of nodes may be insufficient to handle that amount of traffic. In a network with $N$ nodes, $k_2N$ messages can be generated or processed in unit time for some constant $k_2$ . Hence, the actual communication capability is *Min* ( $k_1\frac{L}{\bar{d}}$, $k_2N$ ). With unit constants, we can safely assume *Min* ( $\frac{L}{\bar{d}}$,$N$ ) as a figure of merit. A similar measure was considered for the performance comparison of various multicomputers [7]. For simplicity (we guess), only the number of links ($L$ ) was chosen as the performance metric in [16].

The computation capability being proportional to $N$, we can define the computation-communication capability of a network as $N$.*Min* ( $N$, $\frac{L}{\bar{d}}$ ). The cost of a multicomputer system includes the cost of processors (nodes), links and I/O ports. If we start with the same number of nodes for all the structures we can assume that the cost is proportional to the number of links *(L)* . Taking performance and cost into account, we define the computation-communication availability of the system ($CCA_s(t)$) as:

$$CCA_s(t) = \frac{1}{L}\sum_{i=1}^{x} N_i .Min( N_i, \frac{L_i}{\overline{d}_i} ), \qquad (1)$$

where at time $t$, the network contains $x$ disjoint segments with the ith segment having $N_i$ nodes, $L_i$ links and average distance $\overline{d}_i$. $L$ is the total number of links of the initial configuration. We will consider disjoint segments that have more than two connected nodes [11]. Hence,

$$\sum_{i=1}^{x} N_i \leq N \quad \text{and} \quad \sum_{i=1}^{x} L_i \leq L$$

for $N_i \geq 2$. Note from the above CCA (t) expression that a completely connected structure will be node deficient where as a loop structure will be link deficient.

*Example-2*

Consider the situation depicted in Fig. 1. The $CCA_s(t)$ for Fig. 1.a is 1.63 and the $CCA_s(t)$ for Fig. 1.b is 1.38.

Computation-Communication Availability ($CCA$) as defined above can be interpreted differently for different applications. If the multicomputer only executes tasks that need a minimum of $I$ nodes, the $CCA$ is obtained by summing over the disjoint sets that satisfy this minimum requirement. When $I=2$, the availability is same as that denoted by the equation above. On the other hand for batch processing environments, where the multicomputer executes one task at a time, the $CCA$ of the system will be the maximum of the available computation-communications over all the disjoint multicomputer sets. The task can then be executed on any set that has at least $I$ connected nodes. If none of the disjoint sets has $I$ nodes the $CCA$ of the system is assumed to be zero, because it is of no use. These availability models are called *task based CCA* models in this paper in order to distinguish them from the absolute $CCA$ whose equation was derived earlier. The model is capable of computing both the absolute and task based $CCA$ 's of any multicomputer network.

## 3. SIMULATION TECHNIQUES

In this section we present the simulation techniques for the reliability and the $CCA$ evaluation of various multicomputer networks. We compute the task based reliability as used by Ingle and Seiwiorek [12]. It is defined as follows:

> "If a task needs at least $I$ processors for it's execution, then the system remains operational as long as these minimum resources are available on the system. Otherwise the system goes to the failed state. "

It is to be noted that this technique is a generalization of the previous methods used for reliability computation. For example, when $I=N$ ie. when a task needs all the processors and does not allow any graceful degradation, we get the good old reliability which is the series reliability of all the required components. For $I=2$, the model is similar to the one adopted in [11] and for $I=1$ it is equivalent to Beaudry's model [9].

### Failure and repair assumptions

It is assumed that the failure of the nodes and the links are exponentially distributed over time. A nonexponential assumption will only mean a change to the random selection of failing time and can be easily incorporated into the simulation. We assume homogeneity of all the nodes as well as of all the links to consider identical failure characteristics. Thus we define $\lambda_n$ and $\lambda_l$ as the failure rates of a node and a link respectively. The reliabilities of a node and a link are then given by $R_n(t) = e^{-\lambda_n t}$ and $R_l(t) = e^{-\lambda_l t}$. The node and link failure rates of a system are then given by $X\lambda_n$ and $Y\lambda_l$, where $X$ and $Y$ are the number of active nodes and links taking part in computation/communication at any time $t$. A node is considered active if it is a member of a set of connected nodes and a link is active if it is used for communication between two active nodes.

We consider on-line repair of the nodes and links with exponential distribution of repair times given by $\mu_n^{-1}$ and $\mu_l^{-1}$ respectively. We assume a small repair time of few hours that represents the actual repair duration of a component. However, we may not need such an instant repair facility because of the redundancy provided by the system or due to operating characteristics. So a variation of the above assumption is to stretch the (mean time to repair) MTTR duration. A practical implication of this policy is the unavailability of the repairman to attend the fault. If the unavailability period is very large the actual repair time is negligible and the repair rate is mostly determined by the mean unavailable time of the repairman. The repair is carried out until the system is in the working (up) state so that the system dependability can be computed.

### System representation

The representation of any multicomputer topology is given by an adjacency matrix $A$ using graph theoretic notations. The matrix elements $A[i,j]$ and $A[j,i]$ are '1' if there is a link between node $i$ and node $j$. Otherwise $A[i,j] = A[j,i] = 0$. A nonfailed node $i$, $1 \leq i \leq N$, is represented by making the diagonal element $A[i,i] = 1$. As an example the initial adjacency matrix for a 16 node loop network is given in Fig. 2.

$$\begin{bmatrix}
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0\\
0&0&0&0&0&C&0&0&0&0&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1
\end{bmatrix}$$

Fig. 2 Initial Adjacency matrix for a 16 node ring network.

We use a reachability matrix $R$ [3] to show the graph connectivity. The matrix specifies whether or not there exists a path between two nodes which are not necessarily adjacent. The connectivity of any arbitrary graph can be obtained from it's adjacency matrix using any standard algorithm [18]. It is evident that the initial reachability matrix $R$ of any connected network has all it's elements as '1' indicating that there is a path from any node to any other node in the network.

Whenever a node $i$ fails all the entries of the $i$ th row as well as of the $i$ th column of the $A$ matrix are made '0'. Similarly if the random failure of a link destroys the connection between nodes $i$ and $j$, then the $A[i,j]$ and $A[j,i]$ elements of the $A$ matrix are made '0'. So the $A$ matrix is modified depending on the two types of faults. This modification of the $A$ matrix is then reflected in the reachability matrix $R$, which can be divided into various disjoint matrices having all the elements as '1'. For example the $R$ matrix for Fig. 1.b is given in Fig. 3. It can be observed that the failure of node 4 and link 11 has resulted in two disjoint sets. In one of the sets the connected nodes are {1, 2, 3, 12, 13, 14, 15, 16 } and the second connected set is { 5, 6, 7, 8, 9, 10, 11 }. These two sets are obtained from the $R$ matrix of Fig. 3.

$$\begin{bmatrix}
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1\\
1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1
\end{bmatrix}$$

Fig. 3 Reachability matrix for Fig. 1.b.

As we will analyze the variation of reliability with time, we divide the selected time frame into uniform intervals. Whenever a node and/or link failure becomes due, the program goes through the following steps:

1) An element is selected at random from the existing active elements.
2) If the fault is covered [19] then go to step 3, else increment the failed count of the system and start a new experiment again from time $t = 0$ with the initial configuration of the system.
3) The number of active elements is decremented by one.
4) The $A$ matrix is modified depending on the type of the fault.
5) The next fail time of the node/link is computed.

With the instant repair policy we use a single repairman service facility. The sequence of the steps for repair are:

1) When ever a component fails its repair duration is computed if the repairman is readily available. Else the component is kept in a *FCFS* repair queue till the service is available.
2) The node/link is returned to the system at the completion of repair and the $A$ matrix is modified.
3) The next failed element from the queue is taken up for repair.

The initial $A$ matrix is thus modified with random number of '0' entries due to the failure/repair of nodes and links with the passage of time. We compute the corresponding $R$ matrix at the specified time intervals. The $R$ matrix gives the connectivity information of each node which is employed to find the system condition.

The $R$ matrix, obtained above, is searched exhaustively to obtain all the available valid submatrices. A submatrix is valid if it has all 1's so that it can represent a set of connected nodes (subgraph). The communication performance of a subgraph is then obtained from $Min\,(N_i, \frac{L_i}{\bar{d}_i})$, where $N_i$ is the number of active nodes in subgraph $i$, $L_i$ is the number of links that are used for communication and $\bar{d}_i$ is the average distance a message has to travel in the $i$ th subgraph. $N_i$ is obtained by counting the number of nodes in subgraph $i$. $L_i$ is obtained by counting all $A[j,k]$ such that $j, k \in$ {subgraph i} and $k \geq j+1$. The average distance for an arbitrary network is defined as:

$$\bar{d} = \frac{\sum_{j=1}^{N_i} \sum_{k=1}^{N_i} d_{jk}}{N_i\,(N_i - 1)}$$

where $d_{jk}$ is the shortest distance from node j to node k. So we use a shortest path algorithm [18] to find $d_{jk}$ from any node to any other node in the subgraph and finally compute $\bar{d}$. The adjacency matrix $A$ is used to find the shortest path between nodes of a subgraph. So the sequence of steps the program goes through to find the $R_s(t)$ and $CCA_s(t)$ at some specified time $t$ are:

1) Compute the $R$ matrix from the $A$ matrix and find the various disjoint groups from the $R$ matrix.
2) Select the number of groups 'x' that satisfy the minimum resource requirements I; if any.
3) If x=0, then the system can not satisfy the resource requirements. Increment the failed count and start the experiment again from the initial configuration of the system. Reset the system time.
4) Otherwise for each subgroup $x_i$, find the $N_i$, $L_i$ and $\bar{d}_i$ from the $A$ matrix as explained earlier.

We use multiple independent repetition technique to determine the $R_s(t)$ and $CCA_s(t)$. The failed count is used to obtain the system reliability and $N_i$, $L_i$, and $\bar{d}_i$ are used to find the $CCA_s(t)$ from equation 1.

## 4. SIMULATION RESULTS

The simulation model, discused in section 3, can be used to analyze any arbitrary network topology. It accepts the initial $A$ matrix and the task requirement $I$ as the input parameters and gives the reliability $R_s(t)$ and computation-communication availability $CCA_s(t)$ variation with time. We present here the fault-tolerant capability of five types of multicomputer networks. They are: Full connection (FC), mesh, Hypercube (HC), binary tree and ring networks. The topologies are shown in Fig. 4. Performance and cost parameters for these networks are reported in [5-7].

We have taken an initial configuration of 16 nodes ($N = 16$) for all the networks. Most of our results are based on a node failure rate $\lambda_n = 100$ in $10^6$ hours and link failure rate $\lambda_l = 20$ in $10^6$ hours. The processor failure rate is taken close to that of an LSI-11 processor [12]. The link failure rate is taken one fifth of the processor failure rate and does include the interfaces at both ends of the link. The effect of varying the node and link failure rates are also discussed. We would like to emphasize that the model accepts these failure rates as inputs and hence is suitable for any other failure rates that can be determined for another implementation based on a component count and technology.

Fig. 5 shows the variation of reliability with time for all the above networks when the task needs a minimum of 12 processors and with no repair facility. The results indicate that the FC has the maximum reliability as expected. The reliability of the HC is very close to that of the FC even if it's number of links are only 32 compared to 120 in the FC. This suggests that with a typical link failure rate of 20 in $10^6$ hours four alternate paths from each node is sufficient to provide reliability close to that of the complete connection.

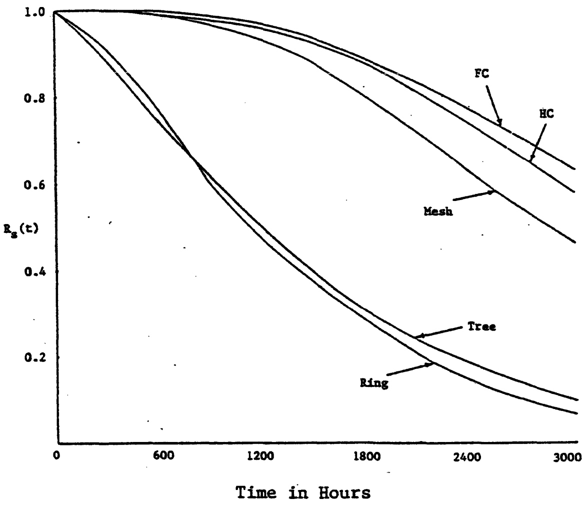


Fig. 5 Reliability variation with time for a task requiring 12 processors and without repair.

$\lambda_n = 0.0001$, $\lambda_l = 0.00002$, coverage = 1.0.

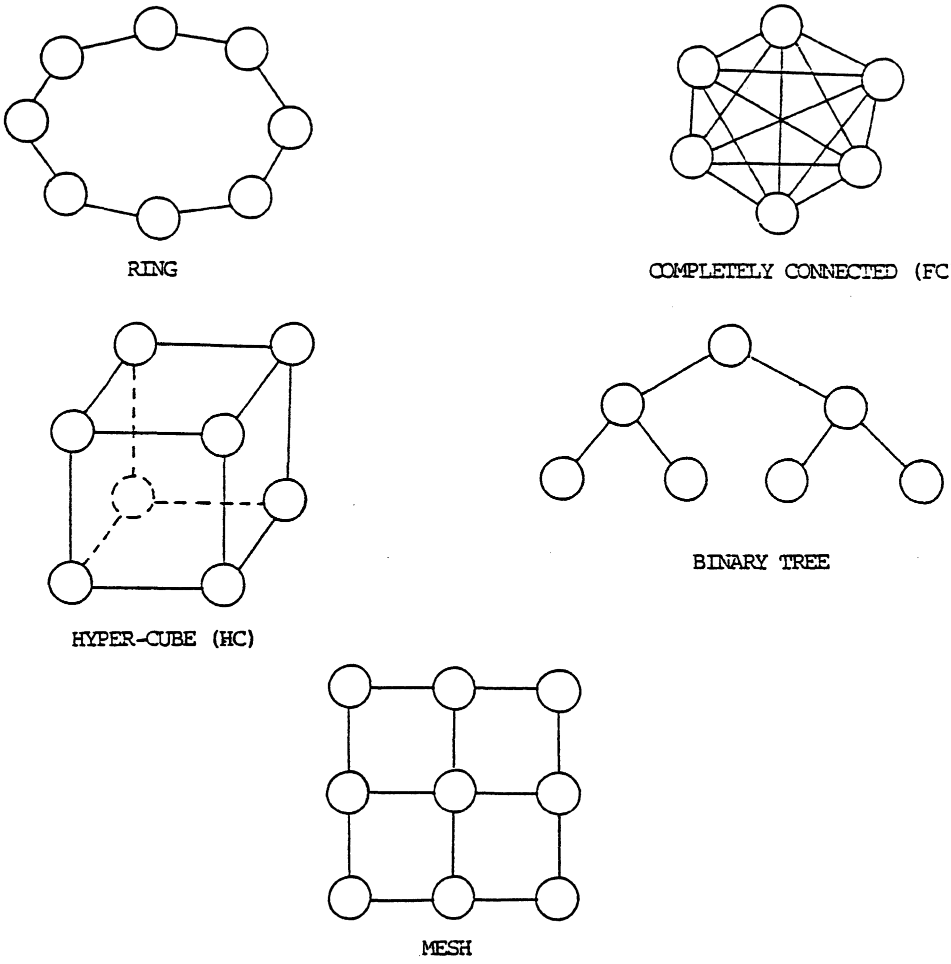


Fig. 4

The ( $N$ -1 ) links from each node in a FC is mostly required for better communication capability and do not increase the reliability linearly. The mesh connection is in the middle range of the reliability bounds. The tree and ring connections have similar poor reliability property. Table I compares the reliability of the five structures for $I=8$, $I=12$ and $I=16$. Naturally as we decrease the value of $I$ ( allow more graceful degradation to the system ) the reliability of the multicomputers increase.

TABLE I
Reliability variation with time for a task requiring I processors and without repair.
$\lambda_n$ =0.0001, $\lambda_l$ =0.00002,
Coverage=1.0.

| I | Time | FC | HC | Mesh | Tree | Ring |
|---|---|---|---|---|---|---|
| 8 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | 300 | 1.00 | 1.00 | 1.00 | 0.98 | 0.98 |
| | 600 | 1.00 | 1.00 | 0.99 | 0.94 | 0.95 |
| | 900 | 1.00 | 1.00 | 0.99 | 0.87 | 0.87 |
| | 1200 | 1.00 | 1.00 | 0.99 | 0.79 | 0.79 |
| | 1500 | 1.00 | 1.00 | 0.98 | 0.70 | 0.70 |
| | 1800 | 1.00 | 0.99 | 0.95 | 0.60 | 0.60 |
| | 2100 | 1.00 | 0.99 | 0.92 | 0.53 | 0.52 |
| | 2400 | 0.99 | 0.98 | 0.88 | 0.46 | 0.45 |
| | 2700 | 0.99 | 0.97 | 0.83 | 0.38 | 0.37 |
| | 3000 | 0.99 | 0.95 | 0.77 | 0.32 | 0.30 |
| 12 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | 300 | 1.00 | 1.00 | 1.00 | 0.88 | 0.91 |
| | 600 | 1.00 | 0.99 | 0.99 | 0.74 | 0.77 |
| | 900 | 0.99 | 0.98 | 0.97 | 0.61 | 0.60 |
| | 1200 | 0.97 | 0.96 | 0.93 | 0.50 | 0.48 |
| | 1500 | 0.94 | 0.93 | 0.88 | 0.39 | 0.37 |
| | 1800 | 0.88 | 0.87 | 0.80 | 0.30 | 0.27 |
| | 2100 | 0.83 | 0.82 | 0.72 | 0.24 | 0.20 |
| | 2400 | 0.78 | 0.74 | 0.63 | 0.18 | 0.14 |
| | 2700 | 0.71 | 0.65 | 0.54 | 0.14 | 0.10 |
| | 3000 | 0.63 | 0.58 | 0.46 | 0.09 | 0.07 |
| 16 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | 300 | 0.65 | 0.64 | 0.64 | 0.58 | 0.63 |
| | 600 | 0.41 | 0.40 | 0.40 | 0.32 | 0.37 |
| | 900 | 0.26 | 0.26 | 0.26 | 0.20 | 0.22 |
| | 1200 | 0.17 | 0.16 | 0.16 | 0.10 | 0.13 |
| | 1500 | 0.10 | 0.10 | 0.09 | 0.06 | 0.07 |
| | 1800 | 0.05 | 0.06 | 0.06 | 0.04 | 0.05 |
| | 2100 | 0.04 | 0.04 | 0.03 | 0.02 | 0.03 |
| | 2400 | 0.02 | 0.02 | 0.02 | 0.01 | 0.01 |
| | 2700 | 0.01 | 0.01 | 0.01 | 0.00 | 0.00 |
| | 3000 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

Now, let the system level requirements be that the system has to run for a 1000 hour mission time (unmaintained) and must have $R_s(1000) \geq 0.9$. Fig. 6 shows the effect of node failure rate on the system reliabilities at $t=1000$ hours for $I=8$. It is observed that the FC has the minimum threshold node reliability $R_{nT}$ and is 0.62. The HC and mesh topologies are next in order with individual $R_{nT}(1000)$ being 0.65 and 0.75. So based on the task requirement and operating conditions the above three networks can be designed with less reliable processors. $R_{nT}(1000)=0.92$ for the tree and ring structures. If we increase the task requirement to 12 nodes it is obvious that the threshold $R_n$ for all the networks will also increase. For $I=16$ the system reliability curve for all the networks will lie below the $R_n$ line and $R_{nT}(1000)$ must be $>0.9$ so that the required system reliability is maintained. A practical implication of these observations is that the FC, HC and mesh are considered more costly compared to star and ring topologies. This is based on the assumption that we use the same processors for all types of networks. But if we can design a FC/HC/mesh with less reliable processors and still can assure the system reliability, then why should we spend more by using the same reliable processors as we need for the star or ring architecture?

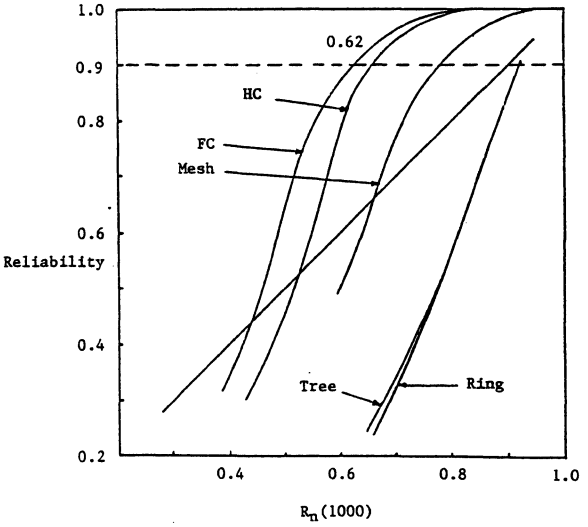


Fig. 6 . Effect of node failure rate on system reliability without repair at t=1000 hours for I=8.

$\lambda_n$ = 0.0001, $\lambda_\ell$ = 0.00002, coverage = 1.0

Fig. 7 shows the variation of $CCA_s(t)$ with time for the five topologies and with no-repair. We use $I=2$ to compute the absolute $CCA$ . It is seen that the FC behaves worst in this case. Even though the FC has the maximum communication performance ( $\bar{d} = 1$ ), its large number of links brings the performance/cost ratio down. The tree and the ring connections lie in the middle of the graph suggesting that they have a fair performance and low cost. They are suitable for applications where communication requirements are not stringent. The HC connection gives the best performance/cost behavior. This is because it has an optimum number of links to provide reliability and $CCA$ close to those of the FC.

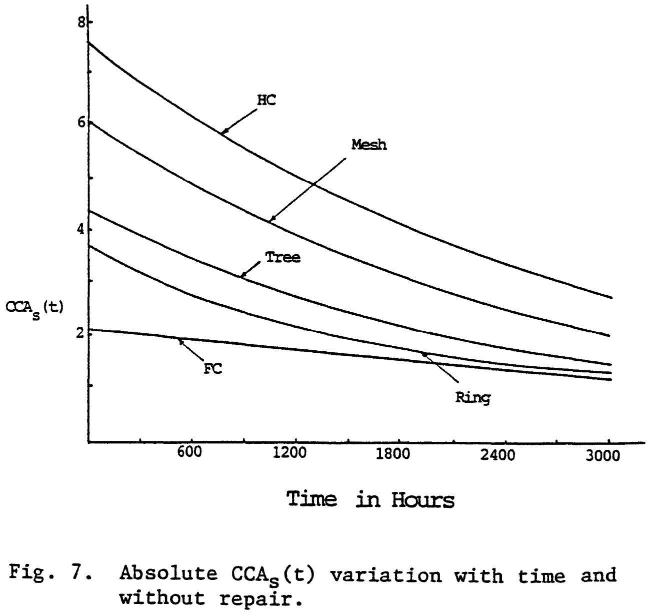


Fig. 7. Absolute $CCA_s(t)$ variation with time and without repair.

$\lambda_n$ = 0.0001, $\lambda_l$ = 0.00002, coverage = 1.0.

TABLE II
Task based $CCA_s(t)$ without repair, $\lambda_n$ =0.0001, $\lambda_l$ =0.00002, Coverage=1.0.

| I | Time | FC | HC | Mesh | Tree | Ring |
|---|---|---|---|---|---|---|
| 8 | 0 | 2.13 | 7.50 | 6.00 | 4.36 | 3.75 |
|  | 300 | 2.02 | 6.85 | 5.43 | 3.70 | 3.09 |
|  | 600 | 1.90 | 6.17 | 4.87 | 3.16 | 2.58 |
|  | 900 | 1.81 | 5.62 | 4.36 | 2.68 | 2.09 |
|  | 1200 | 1.71 | 5.06 | 3.90 | 2.24 | 1.71 |
|  | 1500 | 1.62 | 4.56 | 3.49 | 1.88 | 1.42 |
|  | 1800 | 1.52 | 4.14 | 3.08 | 1.52 | 1.14 |
|  | 2100 | 1.43 | 3.76 | 2.74 | 1.26 | 0.93 |
|  | 2400 | 1.35 | 3.40 | 2.44 | 1.05 | 0.77 |
|  | 2700 | 1.28 | 3.07 | 2.16 | 0.86 | 0.61 |
|  | 3000 | 1.20 | 2.75 | 1.89 | 0.70 | 0.48 |
| 12 | 0 | 2.13 | 7.50 | 6.00 | 4.36 | 3.75 |
|  | 300 | 2.00 | 6.85 | 5.39 | 3.58 | 2.99 |
|  | 600 | 1.91 | 6.17 | 4.79 | 2.84 | 2.32 |
|  | 900 | 1.80 | 5.67 | 4.30 | 2.23 | 1.72 |
|  | 1200 | 1.68 | 5.09 | 3.74 | 1.72 | 1.30 |
|  | 1500 | 1.57 | 4.54 | 3.30 | 1.34 | 0.96 |
|  | 1800 | 1.41 | 4.03 | 2.83 | 1.01 | 0.69 |
|  | 2100 | 1.30 | 3.50 | 2.40 | 0.79 | 0.48 |
|  | 2400 | 1.18 | 3.00 | 2.01 | 0.58 | 0.35 |
|  | 2700 | 1.04 | 2.54 | 1.66 | 0.43 | 0.24 |
|  | 3000 | 0.88 | 2.19 | 1.35 | 0.31 | 0.17 |
| 16 | 0 | 2.13 | 7.50 | 6.00 | 4.36 | 3.75 |
|  | 300 | 1.29 | 4.60 | 3.69 | 2.49 | 2.31 |
|  | 600 | 0.77 | 3.00 | 2.25 | 1.50 | 1.33 |
|  | 900 | 0.50 | 1.75 | 1.39 | 0.89 | 0.79 |
|  | 1200 | 0.31 | 1.21 | 0.96 | 0.46 | 0.46 |
|  | 1500 | 0.18 | 0.70 | 0.52 | 0.28 | 0.27 |
|  | 1800 | 0.11 | 0.48 | 0.35 | 0.18 | 0.18 |
|  | 2100 | 0.06 | 0.28 | 0.19 | 0.11 | 0.10 |
|  | 2400 | 0.04 | 0.14 | 0.10 | 0.05 | 0.06 |
|  | 2700 | 0.02 | 0.09 | 0.06 | 0.04 | 0.04 |
|  | 3000 | 0.02 | 0.07 | 0.04 | 0.03 | 0.02 |

Table II shows the task based $CCA$ comparison of the five topologies. It can be observed that the $CCA_s(t)$ of the FC shows the least degradation with time. This is because of very good fault- tolerance and communication attributes. The mesh connection has the second best $CCA$. Comparing the results of Fig. 7 with those of table II for $I=8$, we can observe that $\sum_{i=1}^{x} CCA_i(t)$ of all the subgraphs with minimum two connected nodes gives less $CCA$ degradation with time for all the structures except for the FC. This implies that the FC has very low probability of having a subgraph with less than 8 nodes in 3000 hours. But the $CCA$ of the ring and tree structures increases dramatically for $I=2$.

The task based $R_s(t)$ with repair is given in Fig. 8 for $I=12$, $\mu_n^{-1}=10$ hours and $\mu_l^{-1}=2$ hours. The effect of repair can be observed by comparing the results with those of the tabulated values given in table I. However, the reliability of the tree topology does not increase as of other structures because of it's unsymmetrical nature. Repair has no effect on the reliabilities of the five structures for $I=16$ (results are not shown in the Fig.).

The effect of varying the node repair interval on the normalized $CCA_s(1000)$ is plotted in Fig. 9 for $I=8$. Here we have assumed separate repair facility for the nodes and links. The results show that different topologies can have different repair intervals to maintain a specific system degradation.

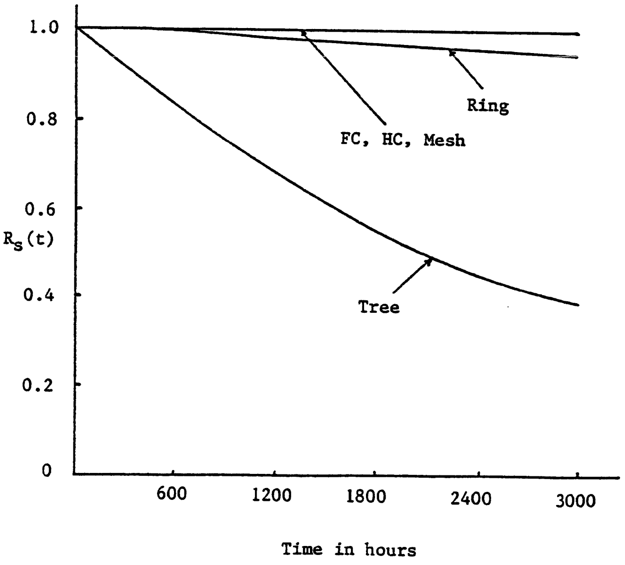


Fig. 8 Reliability variation with time for I=12 and with repair.

$\lambda_n$ = 0.0001, $\lambda_\ell$ = 0.00002,

$\mu_n$ = 0.1, $\mu_\ell$ = 0.5, coverage = 1.0.

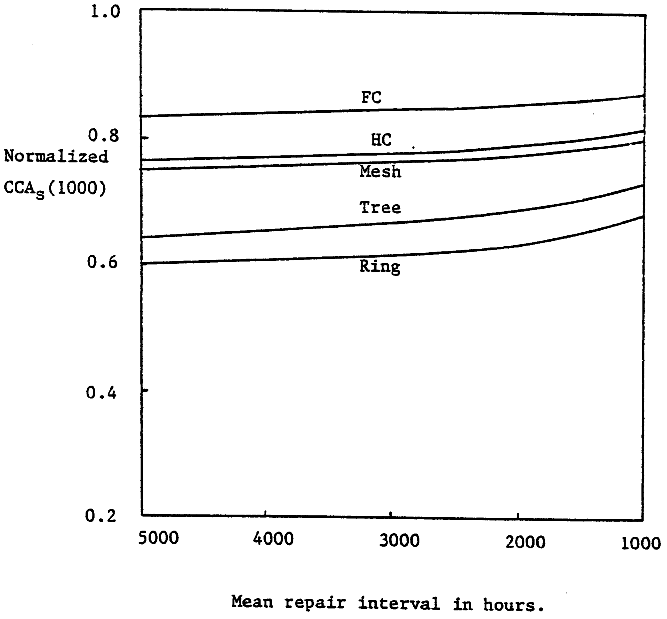


Fig. 9 Normalized $CCA_s(1000)$ against repair interval.

$\lambda_n$ = 0.0001, $\lambda_\ell$ = 0.000005

$\mu_\ell$ = 0.00005, coverage = 1.0.

## 5. CONCLUSIONS

Reliability and computation-communication availability ($CCA$) evaluation of multicomputer networks, with and without repair, are presented in this paper. Simulation is adopted because of the analytical intractability of the problem. The simulation model is quite general and is applicable to all multicomputer graphs. Typical results obtained in section 4, indicate that the hypercube structure performs the best from the reliability and availability stand points. It's cost and performance were earlier shown [5] to be a reasonable balance between loop and completely connected structures.

The effects of node failure rate and repair rate on the system dependability are quite interesting. It is observed that in order to maintain a specified system reliability the node reliability of different networks differs. This is because of the difference in degree of a node of different graphs. Similarly the required frequency of repair varies for various networks to keep the $CCA_s(t)$ within a specified degradable range. These observations lead to another interesting problem: Generally we express the cost of a network in terms of the number of links. This does not take into account the processor reliability at the design phase and the repair cost at the maintenance phase. But we have seen that the threshold node reliability and the node repair rate are lower for structures like FC/HC compared to those of ring/star networks for maintaining a static reliability or availability. Hence, a more appropriate cost model of a network is necessary taking into account node reliability, link cost and repair cost, as is done for software life cycle models. This will allow us to compare the cost of different networks over a specified mission interval.

## 6. REFERENCES

[1] B. W. Arden and H. Lee, "A Regular Network for Multicomputer Systems," IEEE Trans. on Computers, C-31, Jan. 82, pp. 60-69.

[2] C. R. Das and L. N. Bhuyan, "Bandwidth Availability of Multiple-bus Multiprocessors," IEEE Trans. on Computers, Special issue on Parallel Processing, Oct. 1985, pp. 918-926.

[3] C. R. Das and L. N. Bhuyan, "Reliability Simulation of Multiprocessor Systems," Int. Conf. on Parallel Processing, Aug. 1985, pp. 591-598.

[4] G. A. Anderson and E. D. Jenson, "Computer Interconnection Structures : Taxonomy, Characteristics and Examples," ACM Comp. Surveys, Dec. 1975, pp. 197-213.

[5] L. N. Bhuyan and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network," IEEE Trans. on Computers, April 1984, pp. 323-333.

[6] L. D. Wittie, "Communication Structures for a Large Multimicrocomputer Systems," IEEE Trans. on Computers, April 1984, pp. 264-273.

[7] D. A. Reed and H. D. Schwetman, "Cost-Performance Bounds for Multicomputer Networks," IEEE Trans. on Computers, Jan. 1983, pp. 83-95.

[8] J. C. Laprie, "Dependable Computing and Fault Tolerance : Concepts and Terminology," Digest FTCS-15, June 1985, pp. 2-11.

[9] M. D. Beaudry, "Performance Related Reliability Measures for Computing Systems," IEEE trans. on Computers, Jan. 1978, pp. 540-547.

[10] K. Trivedi, "Probability and Statistics with Reliability Queueing and Computer Science Applications," Prentice-Hall, 1982.

[11] K. Hwang and T. P. Chang, "Combinatorial Reliability Analysis of Multiprocessor Computers," IEEE Trans. on Reliability, Dec. 1982, pp. 469-473.

[12] A. D. Ingle and D. P. Seiwiorek, "Reliability Models for Multiprocessor Systems With and Without Periodic Maintenance," Proc. 7th Annu. Int. Conf. FTC, Los Angeles, CA, June 1977, pp. 3-9.

[13] J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems," IEEE Trans. on Computers, Aug. 1980, pp. 720-731.

[14] F. A. Gay and M. L. Ketelsen, "Performance Evaluation of Gracefully Degrading Systems," Proc. 9th Int. Conf. on FTC, Madison, June 1979, pp. 51-57.

[15] B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," IEEE. Trans. on Software Engineering, March 1981.

[16] V. Cherkassy, M. Malek and G. J. Lipovski, "Fail-Softness of Tree Based Local Area Networks," Proc. 5th Int. Conf. on Dist. Comp. Systems, May 1985, pp. 389-385.

[17] M. Tsuchiya, "Availability Analysis for the Design of Distributed Processing Networks," The Journal of Systems and Software 5, 1985, pp. 221-227.

[18] A. S. Tanenbaum, "Computer Networks," Prentice-Hall Inc, 1981.

[19] T. F. Arnold,"The Concept of Coverage and It's Effect on the Reliability Model of a Repairable System," IEEE Trans. on Computers, C-22, Mar. 1973, pp. 251-254.

# Maintenance Architecture and Its LSI Implementation of a Dataflow Computer with a Large Number of Processors

Kei Hiraki Kenji Nishida Satoshi Sekiguchi
Toshio Shimada

Computer Systems Division Electrotechnical Laboratory
Niihari-gun, Ibaraki, Japan 305

**Abstract**

This paper describes the maintenance architecture and its LSI implementation of the SIGMA-1, an ultra high speed data-flow computer for numerical computations in scientific and technological applications. The multi-PE version of the SIGMA-1 adopts the semi-custom LSI implementation approach. LSI implementation is closely related to testability and maintenability. In a parallel processing system with a large number of processors, the architecture for resource management, debugging and maintenance is important as the architecture for program execution. The SIGMA-1 uses combined SIMD/MIMD maintenance architecture. Design policy, implementation method, and improved architecture, including maintenance architecture, are discussed.

# 1 Introduction

Data-flow architecture is identified as the most suitable architecture for parallel processing systems. The main reasons are: The hardware construction of a large-scale parallel processing system using data-flow architecture is much easier than von Neumann architecture; and data-flow computing exploits all the parallelism in a program at the architecture level. Much research has been conducted [1-6] and several prototype data-flow processors have already been built [7-10].

The SIGMA-1 is an instruction-level data-flow computer developed at the Electrotechnical Laboratory. The main objective of the SIGMA-1 is to build an ultra-high speed computing environment, to execute numerical computations in scientific and technological applications such as Monte Carlo simulations and particle simulations that cannot be executed efficiently on a conventional vector processor.

The prototype SIGMA-1 processor with a processing element (PE) and a structure element (SE) is in full operation now[11]. The performance of this prototype is approximately equal to the performance of a von Neumann processor constructed using the same technology [12]. The estimated performance of the SIGMA-1 with 200 processing elements exceeds 100 MFLOPS.

This SIGMA-1 prototype has brought out several problems in constructing a data-flow computer system with a large number of processors. The prototype SIGMA-1 processor is constructed by conventional TTL MSI technology. Size, power consumption and hardware reliability will be serious problems if the same implementation technique is used in the construction of the final SIGMA-1 system. Therefore, LSI implementation of the SIGMA-1 is necessary to construct a system with two hundred processing elements.

In a parallel processing system with a very large number of processors, the architecture for resource management, debugging and maintenance (maintenance architecture) is as important as the program execution architecture. LSI implementation of data-flow architecture makes more difficult to perform operations needed in resource management, testing, maintenance, and debugging.

This paper describes LSI implementation and maintenance architecture of the SIGMA-1 and discusses the problems in the constructions of a data-flow computer with large number of processors. Section 2 outlines architecture of the SIGMA-1 and problems in constructing data-flow computing system with a very large number of processor and the way to avoid them in the SIGMA-1. Section 3 discusses the LSI implementation approach for constructing a data-flow processor including interconnection network. Section 4 describes the maintenance architecture of the SIGMA-1 that performs resource management, maintenance, and debugging.

# 2 SIGMA-1 Architecture

## 2.1 Architecture

Figure 1 shows the global architecture of the SIGMA-1. Four processing elements (PEs) and four structure processing elements (SEs) connected by a local network (LNET) form a group. A process or a subroutine is assigned to one of these groups. This group acts as a high-performance processor in the execution of assignment task.

0190-3918/86/0000/0584 $01.00 © 1986 IEEE

Therefore, interconnection within a group must have high speed and high capacity, otherwise the performance of sequential computing segments of programs decreases. The SIGMA-1's local network, 10 by 10 crossbar packet switching network eliminates this problem.

The global network, a two-stage omega network which connects up to 64 local networks is used for communicating processes and accessing structure processing elements. The SIGMA-1 adopts a new load distribution method [14] implemented in an interconnection network. The computer simulation [14] shows that this load distribution by that method is very close to ideal distribution.

Figure 2 shows block diagrams of a PE and an SE. A PE executes all the instructions and data processing operations except structure operations, and an SE executes structure operations such as read, write, allocate and deallocate.

A PE consists of a FIFO buffer (B), an instruction fetch unit (F), a matching unit (M), an execution unit (E) including a floating point arithmetic unit, and a distribution unit (D). Five units in the PE are arranged to operate in a two-stage pipeline. The B unit stores 8K packets, where a packet consists of 89 bits. The M unit stores 64K packets and fires two-input operations. Single bank chained hashing is used for performing matching. The F unit reads the instruction memory according to the address information on the input packet and also reads immediate operands from the instruction memory. The E unit consists of an integer ALU, a floating point ALU, a structure address generator, an input data-type checker and a sequencer. The D unit consists of a PE address generator, a loop count controller, and an output buffer.

An SE consists of a FIFO buffer (SB) and a structure memory unit (S) which includes a structure memory, hierarchical flags, and waiting queue control circuits. Detailed construction is described in [15].

Experience with the single processor SIGMA-1 shows several problems in constructing a data-flow computer system with a large number of processors. The way to avoid the main problems, synchronization and performance on sequential computation of programs, are discussed below.

## 2.2 Synchronization

Synchronization out of concurrent processes and synchronization between processing elements and memory systems are included in the first problem. A data-flow architecture solves the synchronization problems based on data dependency in a parallel processing system since the instruction the arrival of input data triggers the instruction execution. However, in a data-flow architecture it is very difficult to solve synchronization problems which are independent of data dependency since the execution of such instructions is partially decided by the data dependency of the program. This kind of synchronization is necessary for managing activities on processors, detecting program termination, mutual exclusion and sequentializing associated with I/O operation.

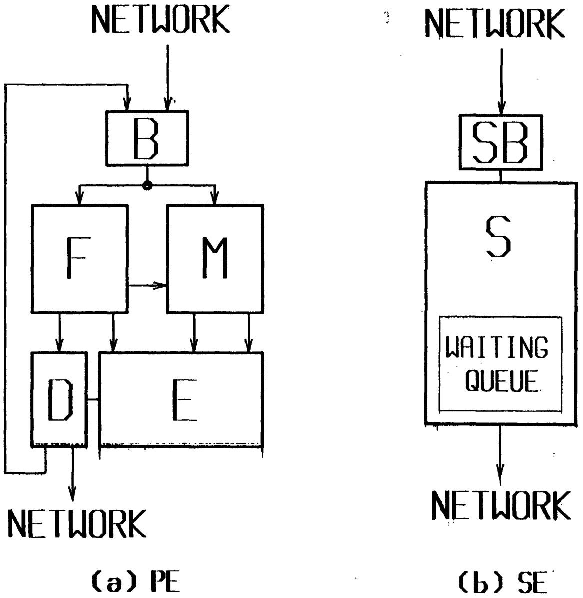


Fig.2 Block Diagrams of Processing Elements

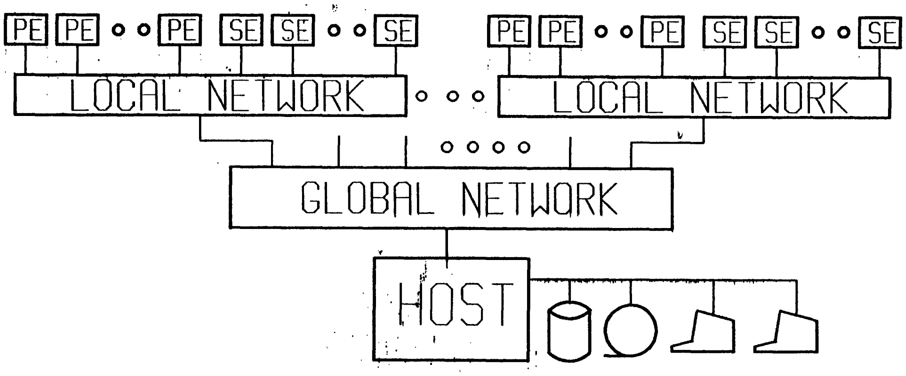


Fig. 1 Global Construction of the SIGMA-1

The SIGMA-1 solves this synchronization problem using operations based on static data-flow model and the maintenance architecture discussed in section 4. In order to realize such operations, the SIGMA-1 utilizes the order preserving property within processing elements and networks and instruction firing and execution without a matching memory. Detailed synchronization based on static operations is discussed in [13].

## 2.3 Performance on Sequential Computing Segments in Programs

Performance of the sequential computations in a program is important in most matrix computations such as linear equations and eigenvalue problems. Figure 3 shows sequential computing segment of matrix computations. Here, the results from operations on all matrix elements (eliminating step or rotation) are accumulated into a scalar variable, then distributed to all elements or all vectors in pivoting in Gaussian elimination and convergence test in iterative programs are performed. In circular pipelined data-flow architecture, all data manipulated by instructions circulate through all processor pipeline stages. However in a von Neumann processor, the data circulates only in the execution stages. This is shown in figure 4. In general, circular pipelined data-flow architecture decreases the performance of the sequential computing segments of a program even in the matrix computations. In order to avoid this problem, the SIGMA-1 uses 2-stage short pipeline control architecture and advanced packet sending mechanism. At the result of this improvement, single input instructions are executed in every 2 clock cycles and two input instructions are executed in every 3 clock cycles in a pure sequential program.

# 3 LSI implementation

The prototype PE constructed consists of eight printed circuit boards with approximately 1900 TTL MSI and MOS memory chips. The number of boards, size, power consumption and reliability prohibit the use of prototype PE's in the construction of the final SIGMA-1 system which has 200 processors. Thus LSI implementation is necessary for constructing the final system.

Data-flow architecture has following characteristics:

1. A data-flow processor does not have registers or history sensitive memories.
2. Pipeline stages in a data-flow processor does not interfere each other except for input and output connection to the neighbouring stages. Independency of each unit enables simple unit design.
3. Communication between processing elements is performed as packet communication. This simplifies interfacing between processing elements.
4. Five memory accesses, two read and two write (insert and delete) accesses to a matching memory, and a read access to an instruction memory are necessary for a two input instruction execution. Figure 5 shows typical memory accesses for an instruction. This is 2.5 times that of a von Neumann processor memory access, where one read access for instruction fetch and one read/write access to data access is performed.
5. Since the concurrency in a program is determined by data dependency, a large buffer that stores excessive parallelism during execution is necessary.

Characteristics (1) to (3) support LSI implementation, while (4) and (5) imply that the data-flow processor requires wider connection between a processing element and memories.

Several approaches in implementing data-flow architecture are considered in the following paragraphs. The first is a single chip approach. Tthere are approximately 60,000 gates in a SIGMA-1 PE. Therefore, it is possible to design a single chip PE by current VLSI technology either adding a sufficient number of pins to the chip or installing buffer memories and matching memories in the chip. However, these conditions cannot be satisfied by the current VLSI technology. Hence, memory interface signals must be multiplexed on a limited number of pins on a chip. Since a data-flow processor requires wider connection between a PE and memories, data-flow architecture is much better than von Neumann architecture if implemented in a VLSI chip.

The second approach is to construct a processing element using ready-made LSIs such as AMD2901 and a microprocessor. Ready made LSIs for fixed-point and floating-point ALUs are most suitable for constructing an experimental processor. However, no ready-made LSIs are available for the most important units, the matching unit, the instruction fetch unit, and the distribution unit in a data-flow processor since they are not common in von Neumann processors. Therefore, the performance of a data-flow processor is reduced if constructed by ready made LSIs.

The single chip approaches are inappropriate for constructing an experimental data-flow processor. We have chosen the third approach, semi-custom LSI approach, for constructing the final SIGMA-1 processing elements. In commercial computers, semi-custom LSI approach has been widely accepted. Design and manufacturing cost for semi-custom design are comparatively high. In addition, modifications to the manufactured LSI is impossible. This is an obstacle for LSI implementation of an experimental processor.

Table 1 shows hardware of the prototype and the final processing element. The final PE consists of semi-custom LSIs, gate arrays and standard-cell LSIs, CMOS memories, and a number of SSIs for drivers and receivers. A PE has eight types of gate arrays, a standard cell LSI,

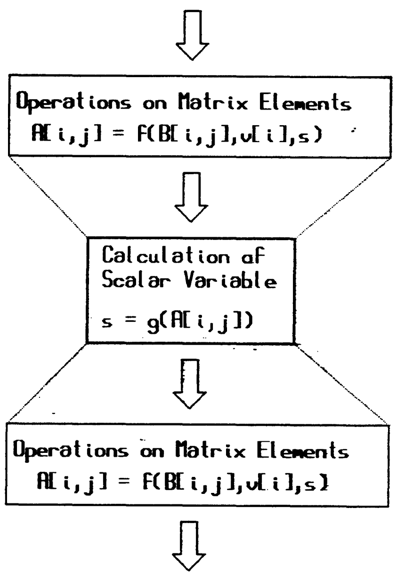

Operations on Matrix Elements
A[i,j] = F(B[i,j],v[i],s)
Calculation of Scalar Variable
s = g(A[i,j])
Operations on Matrix Elements
A[i,j] = F(B[i,j],v[i],s]

Fig.3 Sequential Computing Segment of Matrix Computation

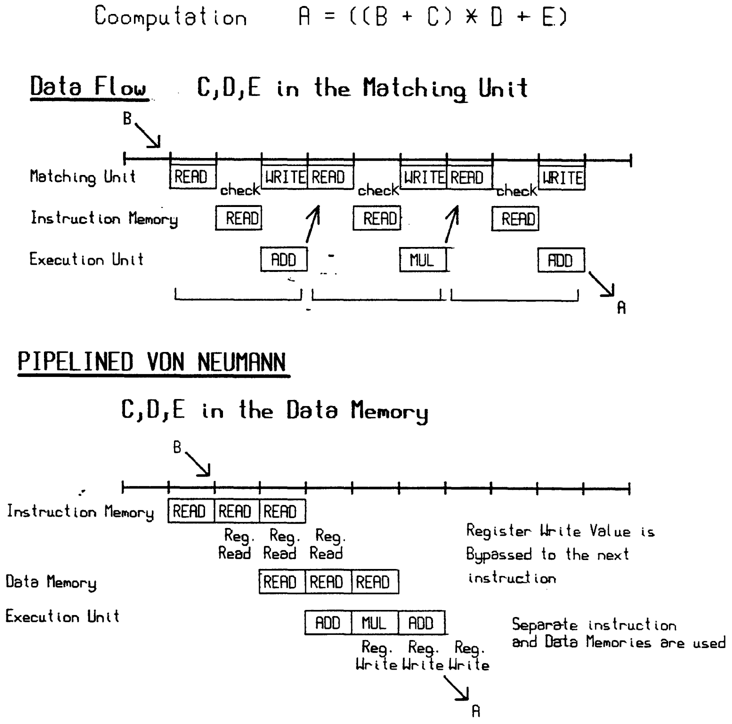


Fig. 4 Execution of a Pure Sequential Program

and contains 28 LSIs. About 42% of ICs in a PE are in the matching unit, (see table 1). This indicates that the matching unit is the most important unit in the SIGMA-1.

These LSIs and memories are assembled on a printed circuit board. The total number of gates in a PE, excluding memories, is approximately 81,000. The total memory capacity in a PE is approximately 1 MB. This includes gates for test and maintenance circuits which will be explained in the next section. The net number of gates, estimated at 54,000, is almost the same as that of a pipelined von Neumann processor employing the same technology. Since the component count is 1570 and there are six printed circuit boards per a PE, both the number of components and the number of printed circuit boards are reduced by one fifth to one sixth of those in a prototype hardware.

# 4 Maintenance Architecture

## 4.1 Maintenance architecture for a dataflow processor

Computer system time for an user includes program execution time, hardware and software initialization, program loading and unloading, execution of an operating system including resource management, program debugging, and hardware testing and maintenance operations. It is necessary to decrease all the computer time to construct a high-speed computing system. So far, resource manage-

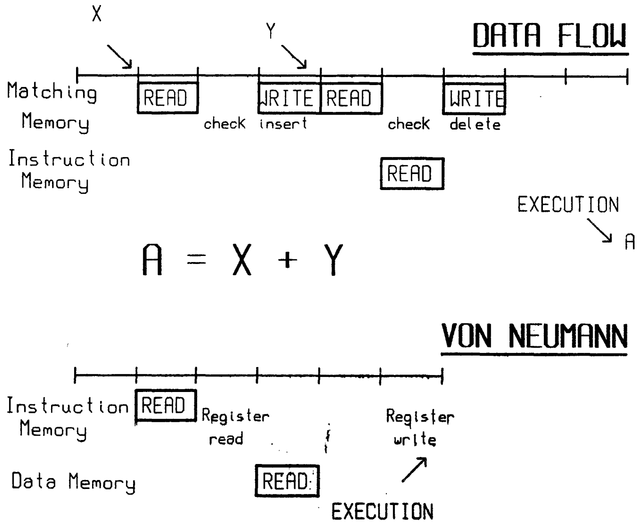


Fig. 5 Memory Accesses for an Instruction

ment, maintenance, and debugging operations on a parallel processor system are performed mutually by many element processors or by a single service processor [16]. If these operations are performed mutually by many processors, it is difficult to control them when overflow or deadlock in hardware resource occurs. If these operations are performed by a single service processor, resource management, maintenance, and debugging can be controlled even if overflow or deadlock occurs. However, the time required for these operations increases linearly with the number of element processors in a system. Hence, the single service processor approach would not be fruitful in a parallel processing system with a very large number of processors. In order to achieve a tolerable time for these operations, a special purpose parallel architecture for resource management, maintenance, and debugging is necessary. This architecture is the "maintenance architecture" of a processor, while the "program architecture" executes the program.

Maintenance architecture is more important in parallel processing environment with a very large number of processors than in a uni-processing environment. The performance of program architecture does not increase linearly with the increasing number of processors in a system. In general, the number of maintenance operations increases slightly more than linearly as the number of processors increases, and interconnection hardware is added to element processors. Consequently, the ratio of program execution time to non program execution system time becomes lower with the increasing number of processors in a system.

When the program architecture is data-flow, non data-flow maintenance architecture is essential because a data-flow architecture alone cannot perform certain kinds of testing and maintenance operations. History sensitive operations or computations utilizing side effects are essentially necessary for resource management, maintenance and debugging. For example, neither the arrival of a data token at a certain node nor data tokens sent to an incorrect position can be detected only by data dependency relationship. Basic hardware tests, such as the connectivity test between components, cannot be performed on data-flow architecture. A von Neumann processor can perform these tests since it can handle history sensitive or side effect operations. Therefore, either von Neumann architecture or non-data-flow architecture is required for maintenance architecture in a data-flow processor. A correct program is successfully executed but the behaviour of incorrect programs or wrong hardware cannot be analyzed by the data-flow architecture itself, since the execution of a data-flow processor is solely determined by data

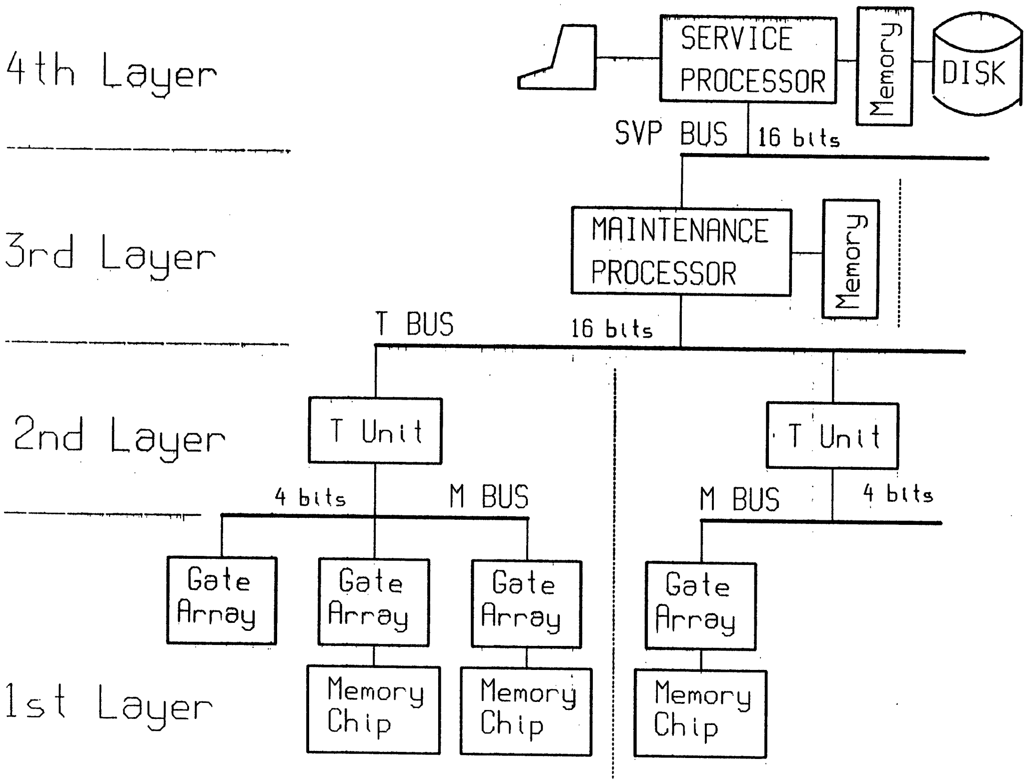


Fig.6 Maintenance Architecture of the SIGMA-1

availability. Hence, a processor solely based on data-flow architecture cannot exist as a standalone computer system, and it is necessary to combine it with non-data-flow maintenance architecture.

## 4.2 SIGMA-1 Maintenance Architecture

The main objective of the SIGMA-1 maintenance architecture is to perform resource management, maintenance, and debugging as fast as a single von Neumann type processor in the 200-PE SIGMA-1 system. For this purpose, the parallelism in the maintenance architecture must be more than that in program architecture because performance of the program architecture does not linearly increase with the increasing number of processors but resource management and maintenance operations increases slightly more than the number of processors in program architecture. Resource management, maintenance and debugging require global control and global data exchanges. Therefore, these operations cannot be executed in independent service processors attached to processor groups in the system.

In order to fulfil these requirements, the SIGMA-1 maintenance architecture adopts a combined SIMD/MIMD parallel architecture. Figure 7 is a block diagram of maintenance architecture in the SIGMA-1. The first and second layers are special purpose SIMD maintenance processors and perform basic operations on hardware components. The third and fourth layers operates as von Neumann MIMD processors and perform high and abstract level resource management, maintenance, and debugging.

The first layer of the SIGMA-1 maintenance architecture consists of maintenance circuits, including conventional scan-in and scan-out circuits in PEs and SEs. This layer accesses all memories, registers and important control signals inside and outside LSIs. All memory cells, registers, and important logic signals are accessed according to M (maintenance) commands sent through the M Bus from T units. All the maintenance items in the first layer are uniquely addressed in the system. Data transfer rate by the first layer is limited, since the data transfer width of the M Bus is limited to four bits and accesses to memories and registers are limited to one bit at a time. However, all the maintenance circuits in the first layer operate concurrently in SIMD fashion. About 33% of logic gates are used in the maintenance circuits. This layer also controls several hardware resources such as the number of packets in a FIFO buffer (B) in a PE, the load factor of a matching unit, execution count of a certain instruction, and deadlock status. This information is passed to higher layers as attention signals and used in the run time control program in the third and fourth layer processors.

The second layer consists of maintenance units (T units) installed one per PE, SE and network board. A T unit is a microprogram controlled special purpose processor that consists of one gate array (2600 gates) and a microprogram ROM. The objective of this layer is to combine bitwise operations in the first layer into wordwise logical operations. The microprogram in the T unit translates the logical maintenance command into a sequence of M commands. These commands are sent to the first layer and gather data from the first layer. The second layer watches for attention signals and controls the system clock signal if necessary. The T unit receives commands and exchanges maintenance data with the maintenance processor through the T Bus.

The third layer consists of maintenance processors, one per group comprising four PEs, four SEs, and a local or global network. A maintenance processor is a commercial microprocessor (8086) with no secondary storage device. This layer checks data from T units, generates maintenance commands to T units, and extracts error or debug information in the data from T units. The fourth layer is a service processor. The service processor, VAX 11/750, is the host processor of the SIGMA-1. This layer is in charge of global human interface control.

The final SIGMA-1 system consists of one service processor, 40 maintenance processors, 360 T units, and over 10,000 maintenance circuits in LSIs.

The maintenance architecture in the SIGMA-1 supports fault tolerant operations. Break down in a processing element breaks down is notified to the service processor immediately. Then the system clock signal to the faulty hardware stops and change configuration registers in networks and processing elements are modified correctly. Thereafter the system operates normally. In this case, maintenance operations can be performed even if the system clock is not applied to processing elements and masks handshake signals in the SIGMA-1 hardware.

# 5 Conclusion

Several problems in constructing a large-scale data-flow computing system with a very large number of processors have been discussed in this paper. Data-flow architecture can solve only some of the problems encountered in a parallel processing system. In the SIGMA-1, most of the other problems are solved by architectural improvements in processing element, network and maintenance architecture.

Semi-custom LSI implementation of the SIGMA-1 indicates that a high-performance data-flow processor with about 1 MB memory can be installed on a printed circuit board. For construct a parallel processing system with a very large number of processors, the data-flow architecture is more suitable than the von Neumann architecture, since the connections to network are much easier in data-flow architecture.

From the hardware point of view, the merit of data-flow architecture is the easy expandability to a large system. Therefore, single chip VLSI implementation is indispensable. However, there are several problems to solve for achieving a single-chip high performance data-flow parallel processor. Some of them are:

1. Design of an efficient, reduced and refined instruction set to reduce the instruction set and the number of instructions executed in a program,
2. Design more efficient matching unit for compact installation of a PE, and
3. Efficient hardware resources management in a PE and an SE.

The preliminary version of the SIGMA-1 PE and SE have been in operation since the latter half of 1984. The final version of the SIGMA-1 was designed and fabricated at the end of 1985. A single group with four PE, four SE and a local network will be in operation at the end of 1986. The final system, consists of 200 processing elements, with a total predicted performance of 100 MFLOPS will be in operation by the end of 1987.

*Acknowledgement*

We wish to thank our colleague, Dr. T. Yuba, Mr. Y. Uchibori, Mr. Y. Yamaguchi and Mr. K. Toda at ETL for stimulating discussions. We especially thank Mr. J. Herath at Keio Univ. for helpful comments. Also we are grateful to Dr. H. Kashiwagi for continuous encouragement. This research is supported by the High-Speed Computing System for Scientific and Technological Use Project of MITI, Japan.

# References

[1] Chamberlin, D. D., "The Single-assignment Approach to Parallel Processing," Proc. Nat. Comp. Conf., Vol.39, AFIPS, 1971, pp.263-269.

[2] Miller, R. E. and Cocke, J., "Configurable Computers: A new Class of General Purpose Machine," Int. Symp. Theoretical Programming, Lecture Notes in Computer Science, Vol.5, Springer-Verlag, 1974, pp.285-298.

[3] Dennis, J. B. and Misunas, D. P., "A Preliminary Architecture for a Basic Dataflow Processor," Proc. 2nd Ann. Int. Symp. Computer Architecture, IEEE, 1975, pp.224-229.

[4] Gurd, J. and Watson, I., "A Multilayered Data Flow Architecture," Proc. Int. Conf. Parallel Processing, IEEE, 1977, p.94.

[5] Arvind, Kathail, V. and Pingali, K., "A Dataflow Architecture with Tagged Tokens," Tech. Rep. TR-174, Lab. Computer Science, M.I.T., 1980.

[6] Plas, A., Comte, D., Gelly, O., Syre, J. C. and Durrieu, G., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," Proc. Int. Conf. Parallel Processing, IEEE, 1976, pp. 293-302.

[7] Gurd, J. and Watson, I., "Data Driven System for High Speed Parallel Computing - part 2: Hardware design," Computer Design, Vol.19, No.6, 1980, pp.97-106.

[8] Kishi, M., Yasuhara, H. and Kawamura, Y., "DDDP: A Distributed Data Driven Processor," Proc. 10th Ann. Int. Symp. Computer Architecture, IEEE, 1983, pp.236-242.

[9] Yamaguchi, Y., Toda, K., Herath, J. and Yuba, T., "A Performance Evaluation of Lisp-based Data-driven Machine (EM-3)," Proc. Int. Conf. Fifth Gen. Computer Systems, ICOT, 1984, pp.512-532.

[10] Takahashi, N. and Amamiya, M., "A Data Flow Processor Array System: Design and Analysis," Proc. 10th Ann. Int. Symp. Computer Architecture, IEEE, 1983, pp.243-250.

[11] Hiraki, K, Shimada, T. and Nishida, K., "A Hardware Design of the SIGMA-1 - A Data Flow Computer for Scientific Computations," Proc. Int. Conf. Parallel Processing, IEEE, 1984, pp.851-855.

[12] Shimada, T., Sekiguchi, S., Hiraki, K. and Nishida, K., "An Evaluation of a Single Processor of a Data Flow Computer SIGMA-1 for Scientific Computations," Tech. Rep. IECE Japan, EC-85-38, 1985 (in Japanese).

[13] Hiraki, K., Nishida, K., Sekiguchi, S. and Shimada, T., "A LSI Architecture of the SIGMA-1: A Data Driven Computer for Scientific Computations," Tech. Rep. IECE Japan, EC-85-39, 1985 (in Japanese).

[14] Hiraki, K., Sekiguchi, S. and Shimada, T."Load Scheduling Mechanism Using Inter-PE Network," Trans. of IECE Japan, Vol.J69-D, No.2, pp.180-189, 1986 (in Japanese).

[15] Hiraki, K., "The SIGMA-1 Instruction Set," Research Memo, Electrotechnical Laboratory, No.85-22J, 1985.

[16] Hosseini, S.H., Kuhl, J.G. and Reddy, S.M., "A diagnosis algorithm for distributed computing systems with dynamic failure and repair," IEEE Trans. Comput. Vol.C-33, No.3,1984, pp.223-233.

Table 1 Hardware of the SIGMA-1

| | Prototype | | LSI Version | | | | |
|---|---|---|---|---|---|---|---|
| Technology | Advanced | STTL | CMOS gate array | | | | |
| | | | CMOS standard cell LSI | | | | |
| | CMOS | SRAM | CMOS SRAM, DRAM | | | | |
| Unit | N | L | N | L | M | G | NG |
| B Unit | 178 | 166 | 23 | 11 | 6 | 14016 | 9252 |
| F Unit | 237 | 180 | 80 | 23 | 6 | 13251 | 7903 |
| M Unit | 385 | 274 | 144 | 28 | 7 | 16835 | 10233 |
| E Unit | 267 | 256 | 45 | 37 | 2 | 13298* | 9726* |
| E Unit (FALU) | 406 | 406 | 12 | 9 | 1 | 7500** | 7500** |
| D Unit | 97 | 97 | 39 | 39 | 6 | 13968 | 9630 |
| T Unit | 0 | 0 | 11 | 10 | 1 | 2367 | 0 |
| PE Total | 1570 | 1379 | 354 | 157 | 29 | 81235 | 54244 |
| SB Unit | 78 | 30 | 23 | 11 | 6 | 14016 | 9252 |
| S Unit | 242 | 176 | 319 | 212 | 1 | 8000 | 7000 |
| T Unit | 0 | 0 | 11 | 11 | 1 | 2367 | 0 |
| SE Total | 320 | 206 | 353 | 234 | 8 | 24383 | 16252 |
| Grand Total | 1890 | 1585 | 707 | 391 | 37 | 105618 | 70496 |
| PCB Count | | 8 | | 2 | | | |

| | N | M | G |
|---|---|---|---|
| Local Network | 32 | 16 | 128000 |
| Global Network | 512 | 256 | 2048000 |

N Number of IC chips
L Number of Logic ICs (not including RAM and ROM)
M Number of LSIs
G Number of gates (translated to 2 input nand gate count)
NG Number of gates excluding maintenance circuits, where drivers are not counted.

* Excluding floating-point ALU
** Floating-point ALU (hard-wired vs. AMD29325)

# TOKEN RELABELING IN A TAGGED DATA-FLOW ARCHITECTURE*

**J.L. Gaudiot and Y.H. Wei**

Computer Research Institute
University of Southern California
Los Angeles, California

(213) 743-0249

**Abstract:** The problem of array handling is a major issue in the design of data-flow multiprocessor systems. We demonstrate here two solutions (namely the I-structures and the token relabeling approach) and apply them to two well known numerical algorithms. The Fast Fourier Transform and the LU decomposition algorithm have been chosen since they are representative of a large class of algorithms and provide good benchmark for the evaluation of the performance of data-flow systems. We base our research upon the MIT tagged data-flow architecture. The results of a deterministic simulation of Arvind's data-flow machine are presented and show the contrast between the two structure representation methods.

## I. INTRODUCTION

Data-flow systems have been proposed as an alternative to the cumbersome programming methods offered by conventional von Neumann computers. The traditional model of computation implies a central control (the Program Counter) as well as a global state (the memory system represented by the *variables* in the high-level programming language). On the other hand, in a data-flow system, the executability of an instruction is decided by the availability of its operands, thereby implementing a ***distributed control*** very amenable to parallel implementation. For more details, the interested reader may refer to excellent surveys of data-flow principles by Treleaven (1982), Srini (1986) or to a special issue of IEEE Computer magazine (1982).

While these principles describe the interaction of *scalar* elements, careful attention must be given when dealing with larger structures. In this paper, we study in more detail the I-structures (Arvind and Thomas, 1978) and the token relabeling approach (Gaudiot, 1986, and 1985). In section II, we describe these two methods. The two algorithms used for illustration are described in section III and their data-flow graph representations are then explained. The simulated machine and the results from the simulation are shown in section IV. Concluding remarks are made in section V.

## II. ARRAY HANDLING

The basic data-flow principles as described above apply mostly to scalar operations. When large data structures are involved, a different mode of processing must be entered. Indeed, the underlying rule of data-flow systems is the ***single assignment*** principle. It implies that no variable can be assigned a value more than once in the course of the program. A scalar token (a value) cannot be modified but a new token can be created after the appropriate processing. Large data structures can be likewise created after "modification" but the copying operation can impose large overhead if a single element of large element array should be modified. For this reason, several schemes have been developed in the past. We will now describe some of the most representative ones.

### 2.1. Heaps

This scheme has been originally described by Dennis (1974). It involves representing an array as a tree of pointers. When a single array element must be modified, only a sequence of pointer modification steps is involved. For an $n$ element array, the cost of modifying a heap is proportional to $\log n$ which compares favorably with a cost proportional to $n$ when the original array is entirely copied. There are several disadvantages which have been associated with the heaps (Arvind and Gostelow, 1982). These include a sequentialization of some array operations, a centralization of array accesses, storage overhead, etc.

### 2.2. I-structures

The I-structures were introduced by Arvind and Thomas (1980) in order to permit pipelining between the producer and the consumer of structures. In other words, an array element should be readable before the entire array has been produced. This scheme can be implemented by the addition of "presence bits" which can be associated with every cell of storage. When a request is made to a "full" cell, the data can be forwarded to the requestor. When the cell is found to be empty, a tag is left, indicating the forwarding address of the original requestor. When the data is ready, it can be directly forwarded to its intended destination.

This method obviously reduces the latency between the creation of a data element and its consumption and allows a more flexible scheduling of operations. However, it introduces additional overhead at execution time and also requires the design of a complex I-structure handling hardware mechanism.

### 2.3. Token relabeling

This scheme was described by Gaudiot (1985) for a data-flow system which uses the U-interpretation principles. When a producer and a consumer of an array can be readily identified, the notion of structure can be entirely ignored at the low-level. Instead, the *tag* associated with each token under the rules of the U-interpretation is used as

*This material is based upon work supported in part by the National Science Foundation under Grant No. ECS-8404345 and by the USC Faculty Research and Innovation Fund

identification of the index of the array element of the high-level language. In other words, it can be simply said that, when an array A is created, its A(i) element is tagged with "i" (Note that the tag is in fact a more complex structure but this simplified description is sufficient in the context of this paper). This approach can be applied very easily to some operations such as the Fibonnaci numbers (Fig. 1) where R1 and R2 are actors which respectively add 1 and 2 to the input iteration tags in order to allow the result produced by the "add" operator during iteration "i" to be re-used during iteration "i+1" and "i+2".

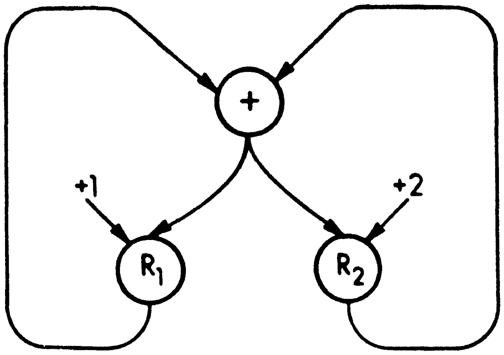


Fig. 1. Token relabeling

More complex program structures are involved in order to apply the scheme to *scatter* and *gather* operations. These constructs, such as the *gather* situation illustrated in Fig. 2, would not be easy to implement (Gaudiot, 1985):

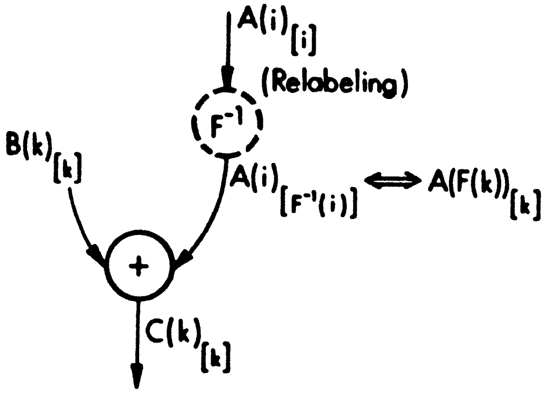


Fig. 2. Gather

```
    DO 1 I=1,100
1   C(i) = B(i) + A(F(i))
```

An inverse function $F^{-1}$ unknown at compile time would be needed to perform the relabeling of data-flow tokens. It has been shown that such a calculation is not necessary. Instead, we introduce in the calling program a sequence generator which produces the F(i)'s, tagged by i (Fig. 3). A relabeling actor (called $\chi$) exchanges the order of the tag and that of the data value. Both the A elements and the i (tagged F(i)) are input to a special actor $\delta$. Without any inverse function, the element A has been matched with the proper F(i); we have found A(F(i)) still attached to its original F(i) iteration label. The special $\delta$ actor is a relabeling actor which takes the A input and relabels it with the content of its other input. In other words, it outputs A(F(i)), with a label i.

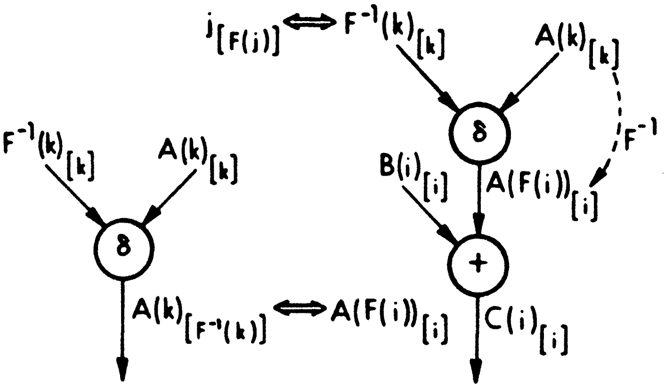


Fig. 3. The $\delta$ actor

This approach has many advantages: better run-time memory management as no intermediary storage is needed, no need for array operations, smaller hardware and execution overhead. However, it requires the complete consumption of a data structure lest some tokens remain orphaned for the rest of the execution of the program.

## III. TWO APPLICATIONS

In this section, we present several data-flow graphs for two parallel algorithms. The crucial consideration in the design of these graphs has been to successfully exploit all the parallelism inherent to the algorithms. Since we are interested in contrasting the performance of the token relabeling scheme with that of the I-structure implementation, we construct two graphs for each algorithm. In each, we apply one of the two different structure handling schemes under study, namely the Token Relabeling approach and the I-structure method. The applications examined in this section have been run on a deterministic simulation program and the results will be shown in the next section.

### 3.1. The FFT application

*3.1.1. The algorithm:* Fig. 4 shows the eight-point decimation-in-frequency FFT computation. In the general case, the FFT computation contains only butterfly constructs with a regular pattern of connections among these butterflies and different multiplicative constants W. The butterfly construct (Fig. 5) obeys the following equations (Oppenheim and Schafer, 1975):

$$x(j+1,p)=x(j,p)+x(j,q) \qquad (1)$$
$$x(j+1,q)=x(j,p)-x(j,q))W_N^r$$

where $x(j,i)$ denotes the data coming into stage $j$, at point $i$. There are $N$ inputs, and $n = \log N$ stages, with the following constraints $0 \le i \le N-1$, and $0 \le j \le \log N-1$. The relation between $p$ and $q$ in a particular stage for a given size of the problem uniquely specifies the connections of butterflies between two successive stages. This means that the routing of data tokens is determined by $p$ and $q$. In addition to the data routing issue, the parameter $r$ is another factor which distinguishes the computations of butterflies located in various place. At a particular stage $j$, the parameters in the equations are derived as follows:

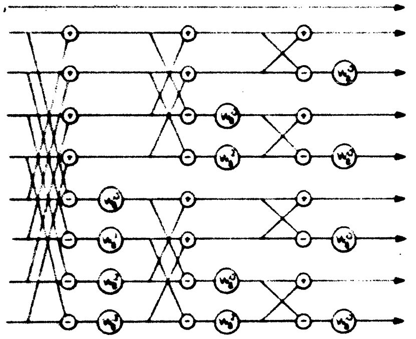


Fig. 4. Eight-point decimation-in-frequency FFT

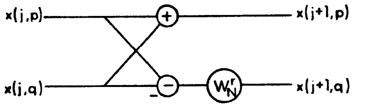


Fig. 5. Butterfly construct

$$
\begin{aligned}
p &= i \ \Lambda\ \overline{2^{n-j-1}} \\
q &= i \ \mathbf{V}\ 2^{n-j-1} \\
r &= (i \mid 2^{n-j-1}) * 2^{j} \\
W_N^r &= e^{-j\pi((i \mid 2^{n-j-1})/2^{n-j-1})}
\end{aligned}
\tag{2}
$$

for all $0 \leq i \leq N$-1, where 'Λ' is a bitwise AND function, 'V' is a bitwise OR function, '|' is an modulo function, and $\overline{k}$ is the one's complement of $k$.

Assuming that $k = 2^{n-j-1}$, then the butterfly becomes:

$$
\begin{aligned}
x(j+1, i\Lambda\overline{k}) &= x(j, i\Lambda\overline{k}) + x(j, iVk) \\
x(j+1, iVk) &= (x(j, i\Lambda\overline{k}) - x(j, iVk))e^{-j\pi(i \mid k)/k}
\end{aligned}
\tag{3}
$$

for all $i$ such that $0 \leq i \leq N$-1.

*3.1.2. The Token Relabeling implementation of FFT:*

As the access pattern of FFT computation is rather regular, the relabeling function is relatively simple. The butterfly construct is implemented by a corresponding data-flow graph which accepts input tokens from two terminals associated with index numbers $j$ and $i$ in the iteration portion of its tag.

In order to express the FFT butterfly of equations (1) and (2) into a data-flow graph, the equations are rewritten as:

$$
\begin{aligned}
x(j+1, i\Lambda\overline{k}) &= x(j, i\Lambda\overline{k}) + x(j, F_1(i\Lambda\overline{k})) \\
x(j+1, iVk) &= (x(j, F_2(iVk)) \\
&\quad - x(j, iVk))e^{-j\pi(i \mid k)/k}
\end{aligned}
\tag{4}
$$

It can be shown that the functions $F_1$ and $F_2$ can be expressed as follows:

$F_2(p) = p \oplus k$
$F_1(q) = q \oplus k$ where $\oplus$ is an exclusive OR

The butterfly is redrawn as shown in Fig. 6 where two index modification blocks $F_1^{-1}$ and $F_2^{-1}$ have been inserted.

The relabeling function $F^{-1}$ is generally not available. This is because it cannot be calculated at compile time. Gaudiot (1985) demonstrated a method which entirely avoids the calculation of $F^{-1}$. This was described in a previous section of this paper. Here, two primitive actors $\delta_r$ and $\delta_w$ are defined for enabling easy operations on the iteration number portion of the tag. The function of the $\delta_r$ actor is to extract the iteration number from the input token while that of the $\delta_w$ actor is to set the iteration field of the tag of one of the input tokens with the data field of another input token (Fig. 7). Note that this $\delta_w$ corresponds to the $\delta$ actor as previously defined by Gaudiot (1985).

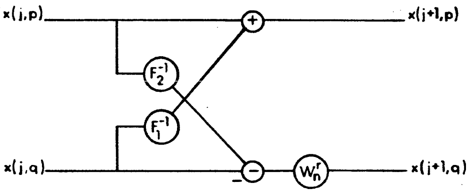


Fig. 6. Modified butterfly construct

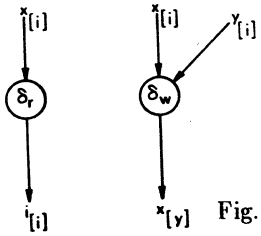


Fig. 7. $\delta$ actors

In general, tag modification can be done by specially designed actors. However, the set of tag modification actors should be kept as small as possible so as not to increase processor complexity. A general solution for the realization of the relabeling function is shown in Fig. 8. In this diagram, the following notation is used: $data_{[i]}$ indicates a data element with an iteration field tag $i$. $\delta_r$ extracts the iteration number from the token $x2_{[i]}$, which $x1_{[F(i)]}$ will be matched to, $i_{[i]}$ is processed by function F. $i_{[i]}$ is then relabeled by $F(i)_{[i]}$ such that $i$ is able to match with $x1$ and relabel $x1$. Finally $x1_{[i]}$ is obtained. The graph containing one $\delta_r$ actor, two $\delta_w$ actors and the function $F$ performs $F^{-1}$ for $x2$. The function $F$ is known from the algorithm and is implemented by the usual arithmetic and logic actors. This general method does not need other specially designed tag modification actors. In the butterfly, $F_1^{-1}$ and $F_2^{-1}$ are realized as shown in Fig. 9.

Fig. 10 is the block diagram of the FFT computation. A stream of one dimension data tokens first passes through the input adaptor which prepares extra context levels and proper iteration numbers for each data to identify the location of data in the FFT graph. The set of input data $x(j,i)$ is separated by the distributor into two groups and which are later forwarded to the two terminals of the butterfly. The butterfly block processes paired tokens and generates one pair of new tokens for the next stage. The loop control block then checks whether the last stage of computation has been reached. If not, the iteration number which stands for the FFT stage is incremented by one and the token is looped back. Otherwise, the token is routed to the output module. A set of constants "$m$" is produced by the constant and $k$-generator and is consumed by the loop control block in order to check for the end of loop. As tokens are produced by the loop, they are postprocessed by the output block. The order is bit-reversed so that a correct set of data could be ready. Both the token relabeling function $F$ and the function $W$ in the butterfly are function of $K$ and of the indices of the token.

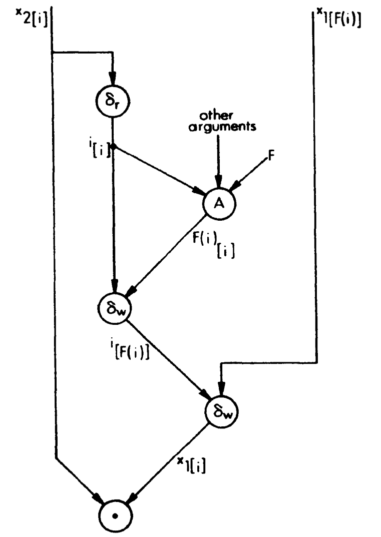


Fig. 8. The relabeling function

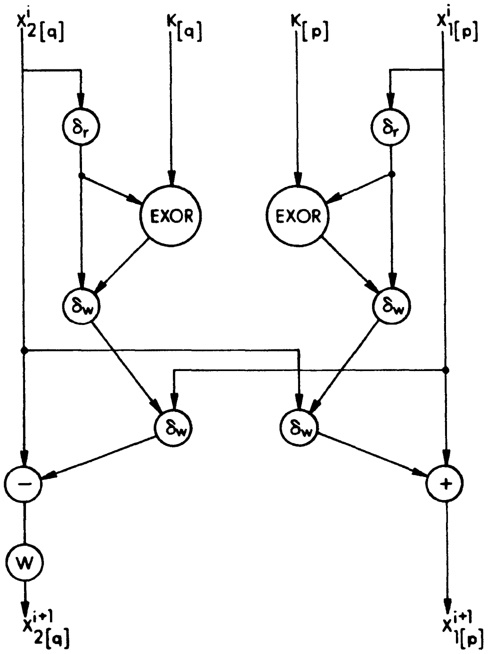


Fig. 9. Realization of $F^{-1}$

*3.1.3. Block diagram of FFT I-Structure graph:* The data-flow graph which corresponds to the "conventional" I-structure organization is represented in a block diagram form in Fig. 11. Instead of entirely unraveling the FFT graph at runtime, an array (using the I-structure representation) is used to buffer the data between stages of the FFT computation. There are four I-structure buffers in the computation of an 8-point FFT since there are 3 stages. Note that since the I-structure definition allows the consumption of an array when it has been partially produced, pipelining is allowed between the stages without waiting for the entire production of the intermediary data. At least on this scheduling point of view, the I-structure representation is similar to the Token Relabeling method. However, as is apparent on the graph of Fig. 11, several additional actors (namely SELECT and APPEND) are required to handle the I-structures. These are not necessary in the Token Relabeling method.

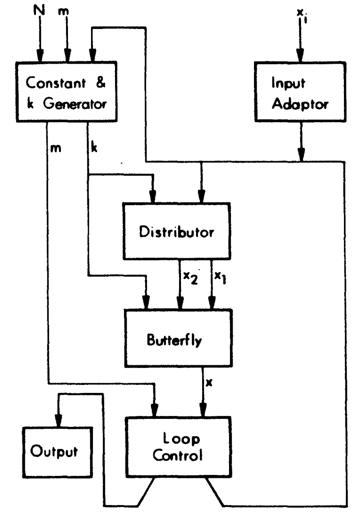


Fig. 10. Block diagram of FFT TR graph

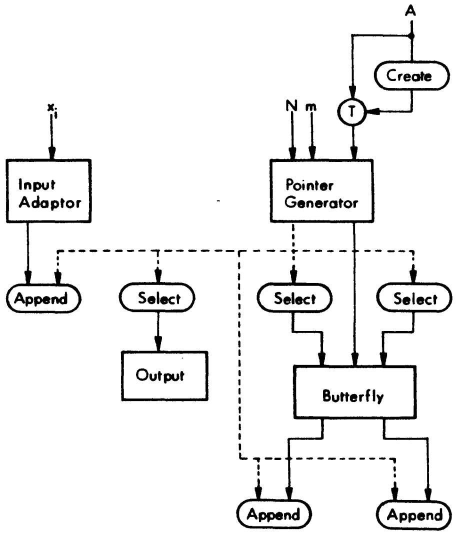


Fig. 11. Block diagram of FFT IS graph

### 3.2. The LU decomposition

The LU decomposition of a matrix is another problem adopted for the evaluation of the behavior of the Token Relabeling and I-structure techniques. There are several reasons in the choice of this algorithm:

- It is a two dimensional problem
- The computation of LU-Decomposition (also referred to simply as LU-D) requires two broadcast operations which can be difficult to efficiently implement in a data-flow environment.

We will only describe the outline of the algorithm to explain the block diagrams of Token Relabeling and I-structure data-flow graphs. Note that the details of this algorithm have been presented by Hwang and Cheng (1982). In Fig. 12, it is shown how the matrix is partitioned into three parts. Part I contains the components of the matrix in first row. Part II contains the components of the matrix in the first column except the one in the first row. The rest of the matrix belongs to part III. The overall outline of the algorithm is presented below:

```
loop size_of_problem > 0
    output I and vertically broadcast I to II and III
    update II
    output II and horizontally broadcast II to III
    update III and set III as new problem
end loop
```

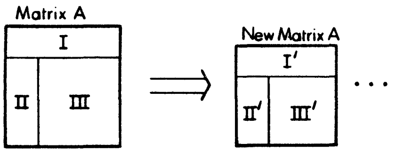


Fig. 12. Matrix partitioning for LU-D algorithm

A new problem with smaller size is created and fed to the next iteration after the current iteration has been completed. Each iteration generates part of the components of the U matrix and of the L matrix.

*3.2.1. LU-D Token Relabeling graph:* Fig. 13 shows the block diagram of the Token Relabeling data-flow graph for LU decomposition. The distributor separates tokens into three streams (I, II, and III). The tokens in stream I are sent out as the partial result of the U matrix. In the mean time, the first broadcast block broadcasts tokens in I to II and III. Receiving tokens from the broadcast block, the second distributor separates tokens into two new streams (II and III). In the new II block, paired tokens in II are processed and a new version of II components is generated. These new II components are sent out as the partial results of L matrix. And, they are also broadcasted to III. The new block III processes three tokens which come from the two broadcast blocks and the distributor and produces a new version of III components. This new version is later sent back as a new smaller problem to be solved. No loop is needed since the size of "new III" will eventually be zero and no more actors will be fired.

Since the token relabeling function is not limited any longer by only the actors $D$ and $D^{-1}$, any function is allowed to modify the tag. As a result, any token in the graph can be arbitrarily chosen and passed to any iteration at will, through the use of the tag detection and tag modification primitives. The broadcast of a token to other iterations of computation is a significant example of the advantages that can be obtained from this method. Systematic mapping of tokens onto the proper iteration is required in order to apply this technique. In this application, the distributor and broadcast blocks indeed realize this mapping.

*3.2.2. LU-D I-structure graph:* As in the FFT algorithm, the difference between the block diagrams of the Token Relabeling and I-structure graphs is in the data accessing modes. The I-structure graph (Fig. 14) uses array accessing actors while the Token Relabeling graph properly labels tokens and uses distributors to route tokens. Both of them share the same major functional blocks and follow the same algorithm. Since the I-structures provide synchronization mechanisms among tokens, the graph is constructed by several functional blocks without interconnections among them. However, pointer generation is complicated by the fact that sets of names and indices of arrays to various iterations for selections and appends of arrays must be generated.

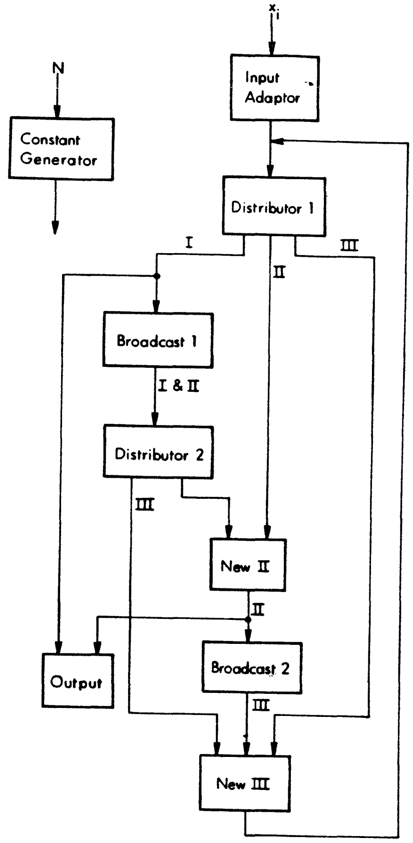


Fig. 13. Block diagram of LU-D TR graph

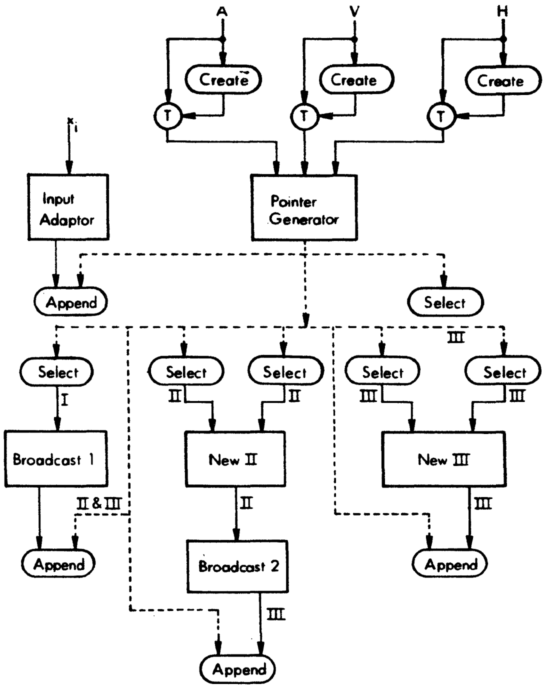


Fig. 14. Block diagram of LU-D IS graph

## IV. SIMULATION AND ANALYSIS

A simulation approach was taken to compare the performance of the Token Relabeling scheme (TR) and of the I-structure scheme (IS). The two applications described in section 3 were simulated under both sets of array representation.

### 4.1. The architecture model

The architecture model of the Arvind/MIT tagged data-flow machine (Arvind, Kathail and Pingali, 1980) was adopted for the machine model of the simulator. It simulates 64 Processing Elements (PEs) interconnected by a packet switching 6-dimension Hypercube network. Each PE can be viewed as three units:

- The *Associative Memory Unit* in which incoming tokens are associatively compared with previously arrived tokens. Matched tokens are then sent, along with the op. code to the next unit.
- The *Processing Unit* receives the ready instruction packets and processes them according to the op. code of the template.
- The *Token Formatting Unit* receives results from the PU and forwards them to the associative memory unit or to another PE through the message passing network.
- The *I-structure controller* where array access operations are handled. Such actors as SELECT or APPEND are executed by this unit.

### 4.2. Simulation results

As in the tagged data-flow architecture, the simulated graphs are executed under the U-interpretation principles. In addition to gathering statistics, this simulator actually computes the results from the simulated data-flow graph and provides correct results. We now describe the results from the simulation. The measures of performance we concentrated on are listed below:

*4.2.1. The graph complexity:* Simply counting the number of actors in the two graphs would not give an accurate measure of the complexity of the graphs. Instead, a dynamic measure must be introduced by counting the number of actors which are actually executed. A convenient method to obtain this figure is to simulate the graphs on a *single processor*. Simulation shows that for the FFT algorithm, the design of the IS graph has higher complexity than that of the TR, while for the LUD algorithm, the TR graph is better.

*4.2.2. Speed up:* Fig. 15 shows the speed-up of two graphs for an FFT computation. The TR graph exhibits a much better speed-up behavior than the IS graph does. All the components of the arrays can be distributed to various processors and processed truly independently. With the communication overhead of structure accessing, the IS graph turns out to have poor speed-up behavior.

*4.2.3. Execution time:* Fig. 16 shows the execution times of FFT-TR and FFT-IS graphs in a multiprocessor environment. For both graphs, tasks are distributed to processors according to the same mapping function.

For a problem with a size smaller than twice the number of processors in the system, the number of processors used for solving the problem is half of the size of problem. Therefore, in Fig. 16, the curve should follow a logarithmic function (log $N$) instead of an $N \cdot \log N$ function in the ideal case (where $N$ is problem size). The TR graph is close to this prediction since its speed-up shown earlier is almost linear. The IS graph appears to follow the $N \cdot \log N$ function because the rate of speed increase becomes slower while the processor number increases. This effect was shown in Fig. 15.

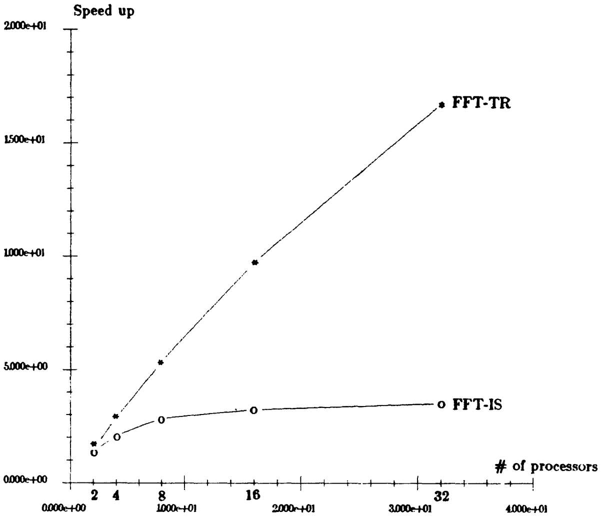


Fig. 15. Speed-up factor

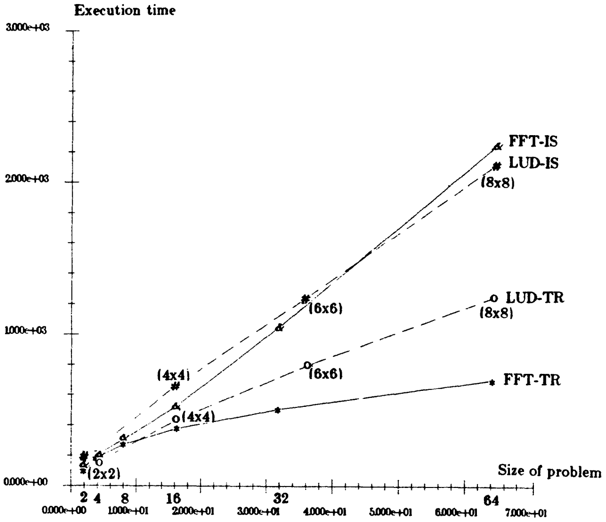


Fig. 16. Execution time

*4.2.4. Number of active tokens:* In Fig. 17, it is shown that the peak number of active tokens in the system for the IS graph is more than twice that for the TR graph in every case. One reason for this is that the requests for structure accessing in the IS computation are always issued before the data are ready. Another reason is that sets of structure names and indices are always generated before the data is computed. Fig. 18 shows that the computation of the IS graph has more than twice the average number of active tokens in the system than the computation of the TR graph has. The reasons can be found in the following:

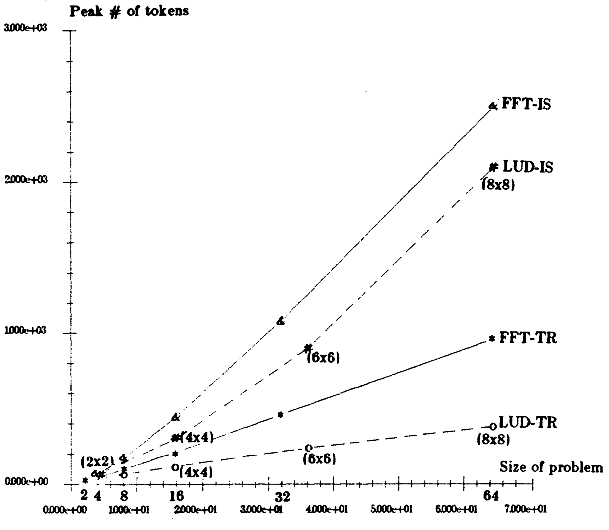


Fig. 17. Peak number of active tokens

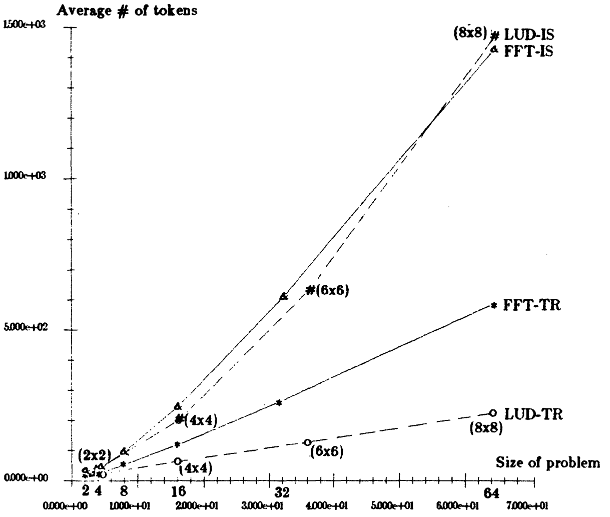


Fig. 18. Average number of active tokens

1) structure names, indices are waiting for data arrival at APPEND actors
2) the requests for data issued by SELECT actors are waiting in the I-structure controller.

*4.2.5. Network load:* Fig. 19 shows that the computation of the IS graph causes much heavier load for the network than the computation of the TR graph does. The higher average number of active tokens in the system introduces more load on the network. Therefore, the figure shows a two to three times increase in network load for IS computation over that for the TR computation.

### 4.3. Analysis

In our performance analysis of the data-flow multiprocessor, many parameters can be observed. We have kept those most closely related to the problems of array handling schemes evaluation.

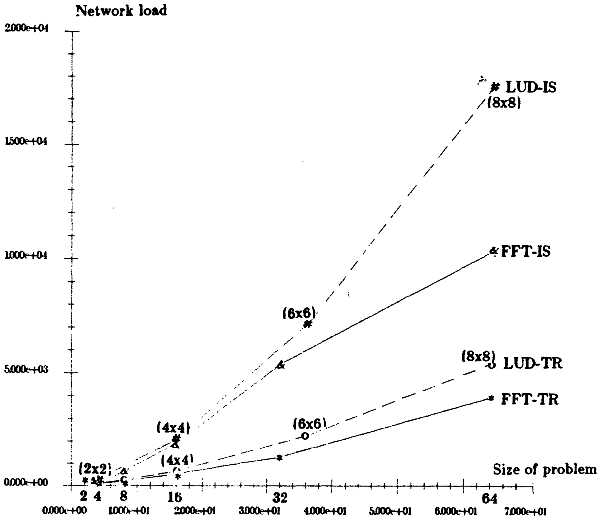


Fig. 19. Network load

*4.3.1. Speed:* Because both the TR scheme and the IS scheme implement graphs based on the same algorithm, the time complexity of the two graphs is similar. The comparison of their execution times eloquently displays the difference in time overhead respectively incurred by the TR and IS schemes. It should be noted that:

1. The execution speed of the TR graph is noticeably higher than that of IS graph. This is because the TR scheme implements the concept of direct data forwarding: A token generated by an actor carries the information about its destination actor and is transmitted without going through any intermediate structure storage. On the other hand, a token in the IS graph format is sent to a designated structure storage and then passed to a destination actor upon request. The overhead is twofold: communication overhead and I-structure handling.

2. The peak number of tokens in the system for the IS graph is larger than that for the TR graph since the request for a data element often exists long before the data is finally computed. Another measure is the average number of token in system. Again the average of tokens of IS graph computation is higher than that of TR graph one. The reason for the higher average number of tokens can be found in the large amount of structure accessing activities.

3. The speed up curve for the TR graph is better than that for the IS graph. The reason is that the I-structure scheme requires requests through complex I-structure controllers.

*4.3.2. Network load:* The IS implementation increases the load on the network more than the TR does because of 1) indirect data movement, and 2) structure names and indices.

*4.3.3. Complexity of the architecture:* For the TR scheme, structured data is converted into scalar elements which are stored in the matching store. This is already supported by the basic data-flow principles. On the other hand, the I-structure scheme requires its own controller.

*4.3.4. Design and generation of data-flow graphs:* A TR graph may be more difficult to design or to generate because the token relabeling function may not be simple and may not be easy to find. The relationship between high level description of an algorithm and the relabeling graph is not always clear which may make the transformation difficult.

However, an IS graph contains concepts close to conventional sequential programming technique. Therefore, the mapping from high level description to graph is simpler. Since the I-structure provides synchronization among dependent stages, loop control is often unnecessary. The body of the graph is simple, but the index generator is complex.

## V. CONCLUSIONS

We have studied in this paper the implementation of two array representation methods. The underlying model of execution was chosen to be the U-interpreter as executed on the MIT tagged data-flow machine. The I-structure method (Arvind and Thomas, 1980) was compared to the token relabeling method (Gaudiot, 1985, and 1986).

Using a deterministic simulation approach, two numerical algorithms were simulated with the two array representation methods. The results of the simulation above pointed to a net speed-up of the Token Relabeling over the I-structure in both cases. Indeed, the TR method presents the following advantages over the I-structure implementation:

- Simplification of the accessing pattern since the tokens need not transit through a special structure controller.
- The data-flow principles of execution can be retained for array handling. This improves schedulability.
- Automatic garbage collection

However, while the token relabeling method can be applied successfully in many cases, there are several drawbacks which often will render the I-structure more applicable:

- The token relabeling assumes a complete consumption of the input stream. Therefore, program constructs which do not completely consume the input stream could require an I-structure implementation.
- Due to the relabeling actors, the graph for the TR implementation is more complex than that of the IS. This implies that for multiple function token relabeling, the overhead might overshadow the advantages.

## REFERENCES

[1] Arvind, and Thomas, R.E., "I-structures: An efficient data type for functional languages," Rep. LCS/TM-178, Lab. for Computer Science, MIT, June 1980.

[2] Arvind, and Gostelow, K.P., "The U-interpreter," *IEEE Computer,* Vol. 15, No. 2, February 1982.

[3] Arvind, Kathail, V., and Pingali, K., "A data-flow architecture with tagged tokens," Laboratory for Computer Science (TM-174), MIT, Cambridge, Massachusetts, September 1980.

[4] Dennis, J.B., "First version of a data flow procedure language," in *Programming Symp.: Proc. Colloque sur la Programmation* (Paris, France, Apr. 1974), B. Robinet, Ed., *Lecture notes in Computer Science,* vol. 19, Springer-Verlag, New York, 1974, pp. 362-376.

[5] Ercegovac, M.D., Patel, D.R., and Lang, T., "Functional languages and data-flow architectures," in *Proc. Summer Computer Simulation Conference,* 1983.

[6] Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., "A second opinion on data-flow machines and languages," *IEEE Computer,* February 1982, pp. 58-69.

[7] Gaudiot, J.L., Vedder, R.W., Tucker, G.K, Finn, D., and Campbell, M.L., "A Distributed VLSI Architecture for Efficient Signal and Data Processing," in *IEEE Transactions on Computers, Special Issue on Distributed Computing Systems,* December 1985.

[8] Gaudiot, J.L., "Methods for handling structures in data-flow systems," in *Proceedings of the 12th International Symposium on Computer Architecture,* Boston, Massachusetts, June 1985.

[9] Gaudiot, J.L., "Structure handling in data-flow systems," in *IEEE Transactions on Computers,* June 1986.

[10] Gostelow, K.P., and Thomas, R.E., "Performance of a simulated data-flow computer," *IEEE Transactions on Computers, Vol. C-29, No. 10,* October 1980, pp. 905-919.

[11] Hong, Y-C., Payne, T.H., and Fergusson, L.O., "An architecture for a data-flow multiprocessor," in *Proc. 1985 International Conference on Parallel Processing,* August 1985, pp. 349-355.

[12] Hwang, K., and Briggs, F.A., *Parallel Processing,* Academic Press, 1984.

[13] Hwang, K., and Cheng, Y.H., "Partitioned matrix algorithms for VLSI arithmetic systems," in *IEEE Transactions on Computers,* December 1982, pp. 1215-1224.

[14] IEEE Computer magazine, *Special issue on data-flow systems,* February 1982.

[15] McGraw, J.R., and Skedzielewski, S.K., "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2," Lawrence Livermore National Laboratory Technical Report M-146, March, 1985.

[16] Oppenheim, A.V., and Schafer, R.W., *Digital Signal Processing,* Prentice Hall, 1975.

[17] Srini, V.P., "An architectural comparison of dataflow systems," in *IEEE Computer,* Vol. 19, Number 3, March 1986, pp. 68-88.

[18] Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P., "Data-driven and demand-driven computer architectures," *ACM Computing Surveys, Vol. 14, No. 1,* March 1982.

# Timing Analysis and Design Optimization of VLSI Data Flow Arrays*

*S. Y. Kung, S. C. Lo, and P. S. Lewis*†

University of Southern California
Signal & Image Processing Institute
Department of Electrical Engineering
University Park/MC-0272
Los Angeles, California 90089
U.S.A.

*ABSTRACT*

Motivated primarily by their potential VLSI implementation, systolic and wavefront arrays have recently attracted significant research interest. A critical research topic is to formalize and systemize the design of such arrays directly from algorithm descriptions. Signal Flow Graphs (SFGs) provide a popular description for recursive parallel algorithms used in digital signal processing. In this paper, a Data Flow Graph (DFG) is used as an abstract model for wavefront array processors. We address first the issue of transforming a SFG into a DFG by an equivalence transformation. Then we discuss the timing analysis for generalized (cyclic or acyclic) DFG networks. Finally, we outline the algorithms to assign minimal number of queues required on all the edges of the DFG and yet achieve the best possible throughput rate. As a side-product of the timing analysis theorem, the deadlock problem associated with the DFG is also resolved naturally.

## 1. Introduction

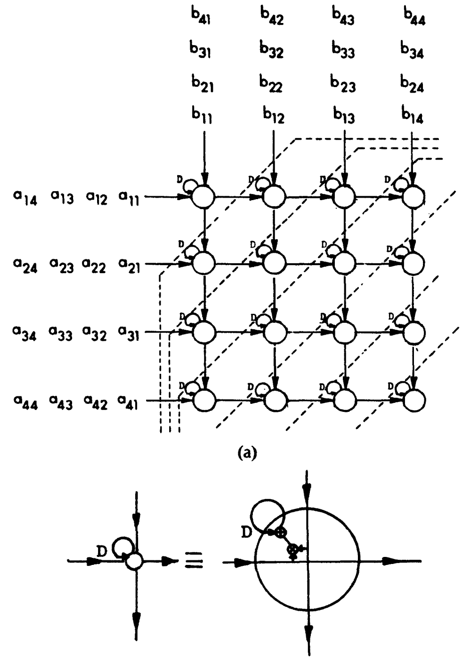
Systolic arrays[1] and wavefront arrays [2] have recently been introduced as efficient VLSI parallel processors. Since then, numerous researchers have attempted to formalize the design of these arrays.[3] In previous publications,[4, 5] we have proposed methods of mapping algorithmic descriptions onto Signal Flow Graphs (SFGs). These SFGs then serve as a notational tool for a particular space-time parallel implementation. This notation facilitates the understanding of recursive algorithms in terms of the computations required for each recursion and provides a convenient tool for mapping algorithms onto parallel processing arrays. A systolic realization of an algorithm expressed as a SFG is easily derived via a cut-set based systolization procedure. To derive a data-driven wavefront realization of a SFG, we introduced a theorem demonstrating the functional equivalence of each SFG to a particular Data Flow Graph (DFG). A DFG serves as an abstract model of a wavefront architecture, so this equivalence relationship provides a correct wavefront implementation for any SFG. Note also that in our treatment, a DFG is a *cyclic graph* instead of the usual acyclic assumption used by many other authors.

Optimization techniques for synchronous systolic realizations of SFGs have been studied at length.[6, 7] Corresponding techniques for wavefront realizations currently do not exist, due to the additional complexity of analyzing asynchronous parallel behavior. In this paper we address the general issue of transforming a SFG to a DFG, which is correct, deadlock-free, and if desired, optimal in terms of throughput and/or hardware. In the following sections, we first review the SFG model of computation. We then introduce the DFG model as an abstraction of a network of wavefront array processors, and explore some of its properties. The timing analysis of an arbitrary DFG is treated in great detail and constitutes the core of this paper. In addition, several Theorems are presented, and algorithms, which transform SFGs into optimal DFGs, are discussed.

### 1.1. Signal Flow Graphs

A SFG is a directed graph G = (V, E, D(e)), where V is the set of nodes (vertices) and E is the set of edges. Nodes model the computation and edges model the communication in a parallel algorithm. Each edge e has a non-negative integer weight, denoted as D(e), which represents the number of delays (D's) on the edge. Directed loops of edges with zero delays are disallowed. It is assumed that the computations in nodes and communication between nodes take *zero time*. A ***recursion*** is defined as the computation inside a SFG for a single set of input data. So, once a data set is input to a SFG in a recursion, it is assumed to go through the nodes instantly, except where blocked by delays (D's) on edges. If the data is blocked by a delay, it will stay in the delay (D) until the next recursion. Therefore the delays in the SFG have the function of separating two consecutive recursions, and keeping the state of the system. To start, all data in D's are assigned according to the initial conditions of the algorithm. With these observations, it is not difficult to see that the SFG actually displays the activities in one recursion of the algorithm. This simplifies the understanding of the complicated space-time activities associated with parallel processing.

For illustration, an example SFG for matrix multiplication of two matrices, C = A x B is shown in Figure 1(a). In each recursion, a new column of matrix A and a new row of matrix B are sent into the array. Each node will multiply the two data coming from up and left, and add it to the partial sum, which is fed back from itself by a delay edge. The operation of a node is shown in Figure 1(b).


$b_{41}$ $b_{42}$ $b_{43}$ $b_{44}$
$b_{31}$ $b_{32}$ $b_{33}$ $b_{34}$
$b_{21}$ $b_{22}$ $b_{23}$ $b_{24}$
$b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$
$a_{14}$ $a_{13}$ $a_{12}$ $a_{11}$
$a_{24}$ $a_{23}$ $a_{22}$ $a_{21}$
$a_{34}$ $a_{33}$ $a_{32}$ $a_{31}$
$a_{44}$ $a_{43}$ $a_{42}$ $a_{41}$
D
≡

(a)

(b)

**Figure 1 (a):** *SFG Array for Matrix Multiplication*
**(b):** *Node Operations*

* This research was supported in part by the National Science Foundation under Grant ECS-82-13358 and by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under Contract No. N00014-85-K-0469 and N00014-85-K-0599.

† On leave from the Los Alamos National Laboratory, Los Alamos, New Mexico

### 1.2. Systolic Arrays

The transformation of such a SFG description to a systolic array can often be accomplished automatically, via e.g. a cut-set retiming procedure.[4, 8] Systolic arrays are very amenable to VLSI implementation and they have the important advantages of modularity, regularity, local interconnection, highly pipelined multiprocessing, and continuous flow of data between the *processing elements (PEs)*. They are especially suitable to certain classes of computation-bound algorithms and have a good number of digital signal processing applications. In fact, they are often derived as a direct mapping of a computational algorithm onto a processor array.[5]

The disadvantages of systolic arrays lie in the fact that the data movements in a systolic array are controlled by global timing-reference "beats". From a hardware perspective, global synchronization incurs problems of *clock skew, fault-tolerance, and peak power.* The burden of having to synchronize the entire computing network will be intolerable for ultra-large-scale arrays. From a software perspective, in order to synchronize the activities in a systolic array, an exact number of additional delays are required. A simple solution to these problems is to take advantage of the control-flow locality, as well as the data-flow locality, inherent in most DSP algorithms. This permits a data-driven, self-timed approach to array processing. Conceptually, this approach replaces the requirement of correct "timing" by correct "sequencing". This concept can be realized in dataflow computers and wavefront arrays.

### 1.3. Wavefront Arrays

Interconnection and memory conflict problems remain very expensive in a general-purpose dataflow multiprocessor. Such problems can be greatly alleviated if *modularity* and *locality* are incorporated into dataflow multiprocessors. This motivates the concept of Wavefront Array Processors (WAPs).[2] Moreover, there are many one- or two- dimensional digital filters and parallel matrix algorithms which are generally expressed in a SFG form. Therefore, it is desirable to explore the relationship between the SFG forms and the corresponding wavefront arrays.

Conventionally, the approach to deriving wavefront arrays is to trace the computational wavefronts and pipeline the fronts on the processor array. In this paper, however, we introduce a new approach based on converting a SFG array into a Data Flow Graph (DFG) array. The DFG is then converted into a wavefront array by properly imposing several key elements in data flow computing. To achieve this, we introduce a general methodology for the timing analysis of DFGs, and use it to optimize the conversions in terms of throughput and queue requirements.

### 1.4. Data Flow Graphs

Data Flow Graphs (DFGs) are used extensively in the area of computer architecture, perhaps most commonly in data flow research.[9] Our purpose is to use a DFG as a formal abstraction of a network of wavefront array PEs. To suit our needs, we use the following definition of a DFG.

A basic DFG is a directed graph G = (V, E, D(e), Q(e)), in which nodes in V model computation, and directed edges in E model asynchronous communication. Each edge e has a queue capacity, represented by an positive integer weight Q(e). Each edge e is also associated with a non-negative integer weight D(e), representing the number of initial data tokens on the edge (initial state). The state of a DFG is represented by the distribution of tokens on its edges. Each edge may contain a nonnegative number of tokens that is less than or equal to its queue capacity. These token may be thought of as filling in a part of the edge queue. This leaves the remainder of the queue empty. Each empty queue slot is called a "space". Therefore, each edge e is also associated with a non-negative integer weight S(e), representing the number of initial spaces on the edge. Obviously, the total edge queue capacity Q(e) is equal to D(e) + S(e), the sum of the initial tokens and spaces on an edge. Hence, the state of a DFG may also be represented by either the distribution of tokens or spaces on its edges.

A node is enabled when all input edges contain a positive number of tokens and all output edges contain a positive number of spaces. The state of a DFG is altered by the firing of enabled nodes. The new state is determined by subtracting one token (and adding one space) from each input edge of the fired node, and adding one token (and subtracting one space) from each output edge.

DFGs have the following properties†:

(1) *Persistence* - Once a node is enabled, it remains enabled until it is fired.

(2) *Conservation* - In any nondirected loop, the sum of the total number of tokens on edges in one direction and the total number of spaces on edges in the opposite direction is unchanged by state transition. A special case of this is the directed loop, in which the total number of tokens (and the total number of spaces) remains constant.

## 2. Equivalence Transformation Theorem

### 2.1. The Equivalence Relation between SFG and DFG

**Theorem 1 :** (Equivalence Transformation between SFG's and DFGs) Barring deadlock situations, the computation of any SFG can be equivalently executed by a self-timed, data-driven machine with a topologically identical DFG. The number of initial tokens assigned on each DFG edge is equal to the number of delays in the corresponding SFG edge. []

*Proof:* What needs to be verified is that the global timing in the SFG can be (comfortably) replaced by the corresponding sequencing of the data tokens in the DFG. Note that the transfer of the data tokens is now "timed" by the processing node. This ensures that the relative "time" between data tokens received at the node is the same as it was in the SFG, as far as that individual node is concerned. By induction, this can be extended to show the correctness of the sequencing in the entire network.‡ []

*Remark:* In the above transformation, any queue capacities greater than or equal to the numbers of initial tokens on the edges are acceptable, from the point of view of functional correctness. The discussion on throughput of any DFG equivalent of a SFG is presented in the next section.

For convenience, we shall term this transformation the **SFG/DFG Equivalence Transformation**. The SFG/DFG equivalence transformation helps establish a theoretical footing for the wavefront array as well as provide insight towards programming techniques. The transformation implies that all regular SFG's can be easily converted into wavefront arrays, making modularly designed wavefront processing elements very attractive to use. Based on the equivalence relationship, the correctness of the DFG is assured.

The D's in the SFG locate the proper setting of the initial conditions in the corresponding wavefront array. We stress that the initial data token distribution plays a very important role in assuring the correct sequencing in a data-driven computing network. The initial state assignment is straightforward: for each delay in the SFG there is an initial data token (regarded as an initial value) assigned to the corresponding DFG edge. We also note that an arbitrary length queue may be inserted on any edge without affecting the validity of the equivalence transformation. It may however, affect the deadlock situation, and may significantly influence the throughput rate, as discussed in Sections 3-5.

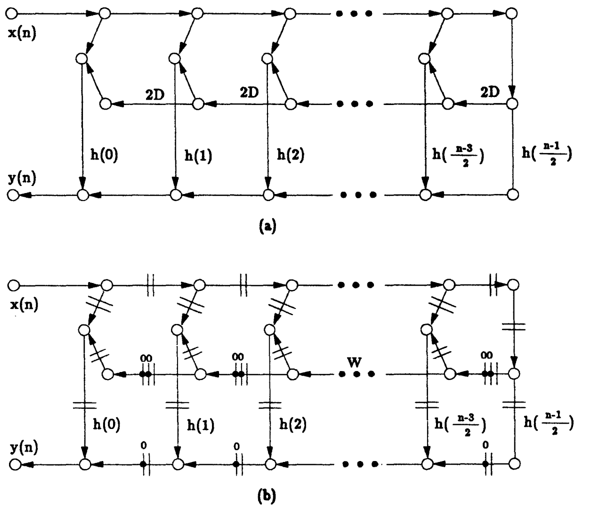
### 2.2. Example: Linear Phase Filter Design

To illustrate the role of the initial states and the correctness of data sequencing, as guaranteed by the equivalence relationship, let us discuss the SFG/DFG equivalence transformation via a linear phase filter example. Linear phase filters have two key features: they have a symmetrical impulse response function, i.e., h(n) = h(N-1-n), and they do not add phase distortion to the signal. Figure 2(a) shows a SFG which takes advantage of the symmetry property, and reduces the amount of multiplier hardware by one half. By the SFG/DFG equivalence transformation, the dataflow graph is derived as in Figure 2(b). In order to ensure the correct sequencing of data, the W data should propagate twice as slowly as the Y data does. Note that the queues play the role of ensuring such a correct sequencing.

Let us now explain the initial conditions shown in Figure 2(b). Note that one initial zero-valued token is placed on each Y-data edge; and two initial zero-valued tokens are placed on each W-data edge.

† These can be easily demonstrated by examining a simple Petri Net [10] model of a DFG.

The first zero-valued token of the Y-data edge, when requested by the Y-summing node, will be passed to meet the V-data token arriving from the upper node. When the operation is done, the way is cleared for sending the next Y-data token from the right-hand PE. The situation is similar for the W summing node, but only one zero-valued token is "used" and the W-data is still one token away from meeting an X-data in the summing node. It will have to wait until the Y-data and the second "0" meet in the lower summing node. This explains why the propagation of W is slower than Y. (This is just what is needed to ensure a correct sequencing of data transfers.)



*Notation for the DFG:*
*An empty buffer is denoted as a bar on an edge, and a full buffer, a bar with a dot on it.*

**Figure 2(a):** *Linear Phase Filter SFG*
**(b):** *Linear Phase Filter DFG*

## 3. Throughput Considerations and Examples

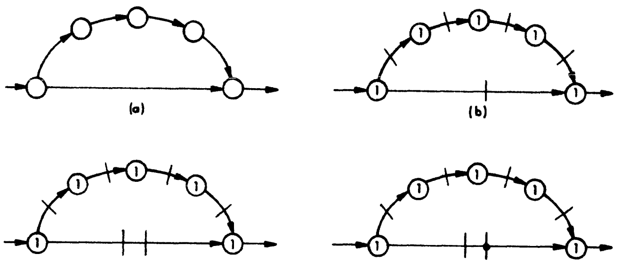
Once the correctness of the DFG is assured, the questions of pipelining speed (i.e. throughput rate) should be explored. To facilitate these analyses, we shall incorporate the notion of time into our DFG model:

*Each node is assigned a deterministic computation time (positive number) and the node fires after it has been enabled for its computation time.* Now a complete specification of a DFG consists of a 5-tuple, G = (V, E, D(e), Q(e), T(v)), where each node v is assigned a positive number T(v), representing the computation time of the node. The persistence property of DFG results in a deterministic behavior of state transitions over time.

### 3.1. An Example Showing the Effect of Queues on Throughput

Figure 3(a) shows a SFG which has two directed paths departing at one end and merging at the other. On the other hand, two corresponding DFGs are shown in Figure 3(b) and 3(c). The operation times of the nodes are indicated by the numbers inside the nodes. With only one space on the lower branch in Figure 3(b), the DFG can accept input tokens at times 1, 6, 11, ... . However, in Figure 3(c), an equivalent DFG with two spaces on the lower path can accept input tokens at times 1, 2, 6, 7, 11, 12, ... . In Figure 3(d), we show a similar DFG (not corresponding to the SFG in Figure 3(a)), in which there is one space and one token on the lower path. And this DFG can accept input only at times 1, 6, 11, ... .

It is clear from this example that the queue capacities and token distribution on the edges play an important role in determining the pipelining period of the DFG.



*Notation for the DFG:*
*The number in a node is the operation time for that node.*

**Figure 3(a):** *A SFG* **(b):** *First DFG corresponding to (a)*
**(c):** *Second DFG corresponding to (a)* **(d):** *Another DFG*

## 4. Timing Analysis for General DFG Networks

The previous example has demonstrated that the pipelining period depends heavily on the queue capacities of the DFG edges. In this section, we shall provide a complete timing analysis for general (i.e. acyclic and cyclic) DFG networks. Our objectives can be stated as twofold: (1) Given a DFG, with defined initial tokens and spaces on the edges, "is it deadlock free?" and "what is the average pipelining period?" (2) Given a desired pipelining period, "how to assign minimal queues on the edges of a DFG to achieve the maximal speed?" Both the questions may be answered by a unified approach based on a duality concept of "token" and "space".

### 4.1. Duality of Token and Space

Figure 4 (a) shows a directed loop with many spaces, but only one token. Since the number of tokens in a directed loop is conserved, there will always be only one token in this loop. The token can traverse the loop repeatedly by a series of firing of the nodes in the loop, and each trip around the loop takes $T_L$ time, which is the sum of all node operation times in the loop. The pipelining period is $T_L$ . However, if we put one more token in the loop, then by the time one token traverses around the loop, the other one also finishes its own trip around the loop. So, the period is only half, $T_L/2$.

Suppose we keep putting more tokens into the loop. We can observe that the period decreases for a while, and then it increases. Let us look at a contrasting case in Figure 4(b). Here, we have many tokens, but only one space. Since the firing of a node requires at least one space on all output edges, we see that only one node can fire at any time. So, effectively, this space will traverse (in reverse direction) around the loop in $T_L$ time also. The pipelining period of this loop is again $T_L$ . By the same analogy, if we put in one more space in the loop, the period will become $T_L/2$. Recall that the number of spaces in a directed loop is conserved.

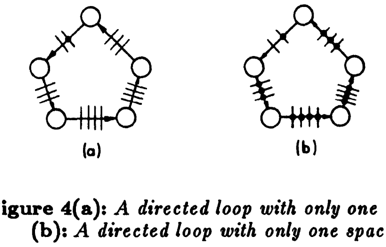


**Figure 4(a):** *A directed loop with only one token.*
**(b):** *A directed loop with only one space*

From this example we see clearly that tokens and spaces play a very similar role in firing nodes. They are both the "resources" to be used by the nodes in order to fire. They display a duality relationship which is quite analogous to the duality of "electrons" and "holes" in semi-conductors. More precisely, a space in one direction plays the same role as a token in the reverse direction. There have been some observations on the roles of tokens and spaces in a DFG,[12, 13] but not to the extent treated here.

‡ The conclusion that the correctness of the computation is independent of the node computation times also follows from results derived by Karp and Miller.[11]

### 4.2. Augmented Flow Graph (AFG)

Based on this key observation, we can simplify our discussion by adopting an augmented graph of the original DFG, which is termed the *Augmented Flow Graph (AFG)*. The augmented graph is constructed by starting with the DFG and adding a reverse edge e' for every existing edge e. We leave the number of tokens on e unchanged. To e', we assign a number of tokens equal to the number of spaces on e. This conversion is shown in Figure 5(a). In effect, we treat spaces as tokens in the AFG.

Recall that, in a DFG, a node can fire only when there is at least one token on all the input edges and one space on all output edges. However, in an AFG, spaces are represented by dual tokens. Therefore, the firing rule of a node in an AFG is also modified to: ***a node is enabled when and only when there exists at least one token on every input edge.*** (It then fires after it has been enabled for its computation time.) It can then be shown that the operation of this AFG is the same as the original DFG. In particular, the throughput analysis will remain the same.†

The DFG properties can be restated in terms of the AFG:

(1) *Persistence* - Once a node is enabled, it remains enabled until it is fired.

(2) *Conservation* - The number of tokens in a directed loop remains constant.

As an example of this conversion, we show in Figure 5(b) a DFG, which is a directed loop, and in 5(c), its AFG. Note that because of augmenting all edges in the original loop, we obtain two directed loops. The inner loop contains only tokens and the outer loop contains only spaces. We also show in Figure 5(d) a DFG, which is an undirected loop, and in (e), the AFG of (d). Notice that in this case the two directed loops in (e) contain both spaces and tokens. The tokens and spaces here play exactly the same role in the timing analysis, which will be explained in the following theorem.

### 4.3. Pipelining Period

**Theorem 2 :** Given a DFG with preassigned initial data tokens and spaces, the pipelining period $\alpha$ of the DFG is:

$$\alpha = Max \left\{ T_V,\, T_P,\, \frac{T_L}{(\, D_{LC} + S_{LCC}\,)},\, \frac{T_L}{(D_{LCC} + S_{LC}\,)} \right\} \quad (1a)$$

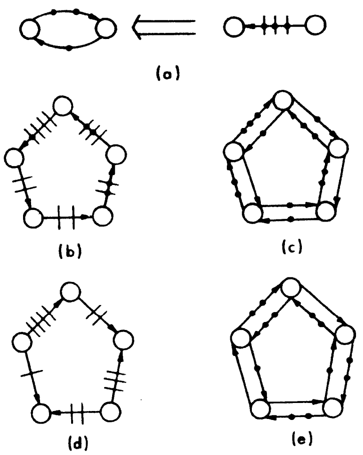
Where L is any ***undirected*** loop in the DFG, $T_L$ is the sum of all node operation times in loop L, $D_{LC}$ is the total number of tokens on edges in the clockwise direction in L, and $S_{LC}$ is the total number of spaces on edges in the clockwise direction in L. $D_{LCC}$ and $S_{LCC}$ are similarly defined for the counter clockwise direction in L. $T_V$ is the maximum of the operation times of all nodes in the DFG. Lastly, $T_P$ is the maximum of any sum of the operation times of a pair of nodes, which are connected by an edge with only one queue on it. []

The above theorem has an equivalent but somewhat simpler statement in terms of the more convenient AFG notation.

**Theorem 2 (Rephrased Version):** Given an AFG derived from a DFG with preassigned initial data tokens and spaces, the pipelining period $\alpha$ of the AFG (or DFG) is:

$$\alpha = Max \left\{ T_V\, ,\, \frac{T_{DL}}{K_{DL}} \right\} \quad (1b)$$

where DL is any ***directed*** loop in the AFG, $T_{DL}$ is the sum of all node operation time in loop L, $K_{DL}$ is the total number of tokens (including dual tokens) in L, and $T_V$ is the maximum of the operation times of all nodes in the AFG.‡ []



*Notation for the AFG:*
*Tokens in the AFG are denoted as dots on the edges.*

**Figure 5(a):** *Rule for converting a DFG to an AFG*
**(b):** *a directed loop DFG* **(c):** *its AFG*
**(d):** *an undirected loop DFG* **(e):** *its AFG*

Note that if DL is a clockwise directed loop in the AFG generated by the undirected DFG loop L, then $K_{DL} = D_{LC} + S_{LCC}$. Likewise, for counterclockwise DL, $K_{DL} = S_{LC} + D_{LCC}$. Therefore, the above two statements are the same in content and differ only in notation. For convenience, our proof will be given only in terms of the AFG notation. For this, let us first establish several useful lemmas.

**Lemma 1:** The pipelining period of an AFG (or DFG) must be greater than or equal to every node operation time. []

*Proof:* This is obvious, since no internal pipelining is assumed inside a node. []

**Lemma 2:** Consider a directed loop L in the AFG. The pipelining period $\alpha_{DL}$ of this loop, when it is operating independently of the remaining part of the AFG, is:

$$\alpha_{DL} = \frac{T_{DL}}{K_{DL}}\,. \quad (2)$$

*Proof:* To show this, first recall that, from the conservation property of an AFG, the number of data tokens in a directed loop DL is constant. If we mark one token which begins its trip around the loop at one particular node, it is clear that there will be $K_{DL}$ tokens fired at that particular node during the period when this marked token completes its loop trip. This marked token will need $T_{DL}$ in time to return to the starting node and this firing pattern will repeat at a period $T_{DL}$, assuming that this loop operates independently of the rest of the AFG. So, we obtain the average period for that loop DL, which is $T_{DL}/K_{DL}$. Note also that ***all nodes in this loop must operate at the same pipelining period.*** []

**Lemma 3:** All nodes in the AFG operate at a uniform period in the steady state. []

*Proof:* While all the nodes in the DFG are in general ***weakly*** connected only, the nodes in the AFG are always ***strongly*** connected, i.e. there exist directed paths in both directions between any pair of nodes in the graph. Consequently, all nodes in the AFG have to operate at a uniform period in the steady state. Were this is not the case, some edges of the AFG will eventually have an unbounded number of tokens. Recall that there are only finite initial tokens in the AFG, and all tokens are contained in some directed loops. By the conservation of tokens in the directed loops, the number of tokens in the AFG remains finite. []

*Proof of Theorem 2:* The proof of Theorem 2 has two parts: "necessity" and "sufficiency".

**Necessity:** To prove the necessity part, we need to show that the DFG can not operate with a pipelining period less than the $\alpha$ in Eq. (1). Since the AFG is strongly connected, according to Lemma 3, all the loops will operate at the same period. Obviously, the slowest

† With this definition, AFGs form a subclass of marked Petri Net Graphs, augmented with node times.

‡ Similar results have been reported by Reiter[14] in the scheduling context and by Ramamoorthy and Ho[15] for petri nets. Ramamoorthy and Ho's analysis, dealing with a class of petri nets that they define as "decision-free, consistent, extended timed petri nets", is the closest to our own. However their treatment neglects the case where

$$Max \left\{ T_V \right\} > Max \left\{ \frac{T_{DL}}{K_{DL}} \right\}.$$

In addition, their proof, while formally presented, utilizes some non-rigorous techniques in the manipulation of infinite terms.

loop will prevail; therefore, the pipelining period for the total system will be at least the maximum of all periods associated with the individual directed loops. Thus, the necessity part is proved.

**Sufficiency:** Here we want to prove that the DFG can operate at the period of $\alpha$ in Eq. (1).

Consider two directed loops A and B. Assume that $\alpha_A = \alpha_B$, and loops A and B are coupled at some nodes. Then the pipelining period of both loops A and B together will be $\alpha_A$, except for some "phase difference" between A and B when both loops start computing. After some finite amount of time, both A and B will be synchronized at the period of $\alpha_A$.

Now assume that $\alpha_A > \alpha_B$. We first modify the loop B into B' by adding extra operation time into nodes in loop B that are not also in loop A, such that the new period for loop B' becomes $\alpha_A$. This is always possible, since the number of tokens in a directed loop is constant, and we can increase the period by prolonging the total node time around a loop. By step (2), now the two loops A and B' can operate at the period of $\alpha_A$.

Comparing loops B and B', it is clear that any node time in B' should be greater than or equal to the corresponding node time in B. It is not hard to envision that the loop B can always operate at the speed of loop B', since the nodes of B are as fast or faster than their B' counterparts. Since loop B' can operate at the period of $\alpha_A$, it is straightforward to see that loop B can operate at this speed also. So, the resulting loops A and B can operate together at the period of $\alpha_A$.

By induction, the above argument can be extended to any number of directed loops in the AFG. Namely, if the slowest period of all directed loops in the AFG is $\alpha$, all nodes in the AFG will eventually operate at a period of $\alpha$, and the time needed for the synchronization of all loops will be finite, since there are only finite loops in the AFG. Thus, the sufficiency part is proved. Q.E.D. []

A special case of this formula concerns the effect of putting only one queue on an edge in a DFG. In Figure 6(a), we show a simple DFG with two nodes, and only one queue on the edge between the two nodes. In Figure 6(b), the corresponding AFG is shown. It is clear that an edge between two nodes in a DFG induces a small loop in its AFG counterpart. This results in a special effect for the case when there is only one queue on the edge. According to Eq. (1b), the pipelining period must be greater than or equal to T(1) + T(2). This implies that these two nodes can only operate sequentially, i.e., only one can operate at any instant. This is the reason that the term $T_P$ appears in Eq. (1a).

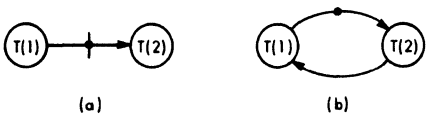


**Figure 6(a):** *a DFG with one buffer on the edge*
**(b):** *the equivalent AFG of (a)*

Another observation is that while there are many directed loops in the AFG, only some of them are ***simple directed loops***, i.e. loops which visit nodes only once, except for the node where the loop "begins" and "ends". In Figure 7, there are two simple directed loops, denoted as $(n_1\, e_1\, n_2\, e_3\, n_1)$, $(n_2\, e_2\, n_3\, e_4\, n_2)$. An example of non-simple directed loop is $(n_1\, e_1\, n_2\, e_2\, n_3\, e_4\, n_2\, e_3\, n_1)$. Fortunately, we need to consider *only the simple directed loops* for the pipelining period computation. This claim is proved in the following lemma:

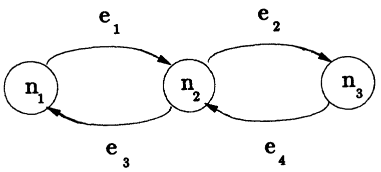


**Figure 7:** *AFG Example*

**Lemma 4:** The pipelining period of a non-simple directed loop DL in an AFG is less than or equal to the maximum of the periods of all simple directed loops contained in DL. []

*Proof:* Without loss of generality, we can use Figure 7 as the AFG. We want to show that

$$\text{For} \left[ \begin{array}{ll} T_1 = T(n_1) + T(n_2) & K_1 = K(e_1) + K(e_3) \\ T_2 = T(n_2) + T(n_3) & K_2 = K(e_2) + K(e_4) \end{array} \right.$$

$$Max \left\{ \frac{T_1}{K_1}, \frac{T_2}{K_2} \right\} \geq \frac{(T_1 + T_2)}{(K_1 + K_2)}$$

$$\textit{Assuming that:} \frac{T_1}{K_1} \geq \frac{T_2}{K_{2,}} \textit{ it is easy see that:}$$

$$\frac{T_1}{K_1} \geq \frac{(T_1 + T_2)}{(K_1 + K_2)}$$

Other cases can be proved similarly. []

It can be deduced from this lemma that we can consider *only the simple directed loops* for the pipelining period computation. So, in Eq. (1b), we can assume that DL is a simple directed loop.

### 4.4. Deadlock Analysis

The issue of deadlock can be regarded as a special application of the above Theorem. The result is stated in the following Corollary:

**Corollary:** The DFG is deadlock free if and only if there exists a finite solution $\alpha$ for Eq. (1). A finite solution for $\alpha$ exists if and only if the AFG contains no empty directed loops. []

*Proof:* It follows directly from Theorem 2. []

An example of deadlock is shown in Figure 8.

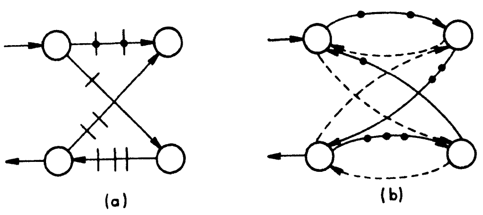


**Figure 8(a):** *A deadlocked DFG* **(b):** *Its corresponding AFG*
*Note that there are no tokens in the loop indicated by dashed edges. According to Eq. (1b), there exists no finite solution for $\alpha$.*

### 5. Optimization of Throughput

Theorem 2 gives the pipelining period of a DFG when the initial token assignments and number of queues on edges are given. The dual problem of determining the assignment of minimum queue capacities on the edges in a DFG to achieve the minimum (or optimal) pipelining period $\alpha^*$ is treated in the following theorem:

**Theorem 3 :** Given a DFG with initial token assignment, (i) the minimum (or optimal) pipelining period and (ii) the minimum queue capacities on the edges to achieve that pipelining rate can be determined as follows:

(i) The minimum pipelining period $\alpha^*$ is

$$\alpha^* = Max \left\{ T_V \text{ , } all \ \frac{T_{DL}}{D_{DL}} \right\} \tag{3}$$

where DL is a ***simple directed loop*** in the DFG, $T_{DL}$ is the total node time in DL, and $D_{DL}$ is the total number of tokens in DL. $T_V$ is the maximum of the operation times of all nodes in the DFG.

(ii) The assignment of the minimum queue capacities on the edges of the DFG requires that the queue capacities must be the minimum numbers satisfying the condition in Eq. (1a) in Theorem 2:

$$S_{LC} \geq \frac{T_L}{\alpha^*} - D_{LCC} \tag{4a}$$

$$S_{LCC} \geq \frac{T_L}{\alpha^*} - D_{LC} \tag{4b}$$

*for any simple undirected loop* L in the DFG. and,

$$Q(e) \geq \left\lceil \frac{T(source) + T(sink)}{\alpha^*} \right\rceil \quad (4c)$$

$$Q(e) \geq D(e) \quad (4d)$$

for any edge e in the DFG. []

Note that $\alpha^*$, $T_L$, and $D_{LC}$, $D_{LCC}$, $S_{LC}$, and $S_{LCC}$, are defined in Theorem 2. And Q(e) is the queue capacity of an edge e, T(source) and T(sink) are the operation times of the source and sink nodes of the edge e.

*Proof:* Note that the optimal pipelining period may be obtained by putting $\infty$ spaces on all edges in the DFG. Theoretically, it is clearly the best one can do in order to maximize the pipelining rate. In this case, for all DFG undirected loops with both clockwise and counterclockwise edges (all but the directed loops), $S_{LC} = S_{LCC} = \infty$. As a result their individual pipelining periods will be 0 (according to Eq. (2)), which can never adversely affect the overall pipelining period. Thus, such undirected loops may be excluded for the purpose of applying Eq. (1) for determining $\alpha^*$. Therefore, only those directed loops in the DFG remain to be considered. In this case, Eq. (1) becomes Eq. (3) and the Step (i) is thus verified. The validity of Step (ii) follows directly from Theorem 2 and Eq. (1). (Eq. (4d) is needed to insure spaces for the initial tokens.) []

In fact, the formula Eq.(4) in the step (ii) of this theorem is also valid for any given desired pipelining period which is greater than or equal to $\alpha^*$. In general, there are non-unique solutions to this assignment. Among several possible approaches to deriving a feasible solution, we propose in the next subsection a simple cut-set procedure [4, 8] to tackle this assignment problem.

## 6. Linear Queue Assignment Algorithms

### 6.1. Comparison with Retiming for Synchronous Systems

In this section, we will present a cut-set retiming scheme to analyze the timing of an asynchronous system, i.e. a DFG. This cut-set procedure basically extends the earlier timing analysis research on synchronous circuits.[6, 7] In order to deal with generalized asynchronous data flow arrays, the DFG modeling is required to abstract an asynchronous circuit with initial conditions. Note that initial conditions affect the optimal pipelining period $\alpha^*$.

Moreover, compared with the systolic array, the wavefront array represents an effective means to deal with multi-rate processing elements. In systolic arrays, a common procedure is to adopt a uniform clock unit based on the slowest node, which is not a good scheme in terms of pipelining speed. It is also noted that, the DFG modeling, with queues accommodating initial tokens, has an important advantage. It allows us to avoid the tedious (and sometimes impossible) procedure of initial tokens reassignment as required in systolic design.

### 6.2. Cut-Set Procedure for Asynchronous Systems

A direct approach to compute the optimal queue assignment would involve tracing all the simple loops - a very time consuming task. To avoid this, we propose a cut-set retiming procedure (to be detailed in the Appendix), to assign queues on the edges in the DFG. In short, a cut-set retiming procedure consists of two basic transformations of a computing network which preserve the functional correctness of the network. One transformation is time scaling, i.e. slowing down the clock rate. The other is the transferring of time delays along a cut-set of the computing network. The objective of cut-set retiming is to transform from the SFG in which nodes have zero-delays, to a *retimed* DFG in which the operation time of the node is assigned to its output edges. With these two basic operations, ***The cut-set procedure determines the optimal pipelining period of the DFG, $\alpha^*$, and also assigns to each edge e of the DFG an appropriate time delay, denoted as t(e), needed to achieve this optimal period.*** This procedure is stated in the following theorem:

**Theorem 4 :** After the cut-set procedure, the queue capacity Q(e), needed for the edge e in a DFG for optimal pipelining period $\alpha^*$, can be computed as:

$$Q(e) = Max \left\{ D(e), \left\lceil \frac{(t(e) + T(sink))}{\alpha^*} \right\rceil \right\} \quad (5)$$

where the "ceiling function" $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. Also, t(e) and $\alpha^*$ are defined as above (and derived in the Appendix). D(e) is the number of initial tokens on edge e, T(sink) is the operation time of the sink node corresponding to the edge e. []

*Proof:* We want to show that the queue capacities obtained from Eq.(5) satisfy Eq.(4). According to the rule used in the cut-set retiming procedure, t(e) should be greater than or equal to the operation time of its source node. (Note that in our retimed graph the node operation time has to be absorbed by its output edges.) Now, consider any undirected loop L in the retimed DFG. For the purpose of illustration, an undirected loop and its retimed DFG are shown in Figure 9.

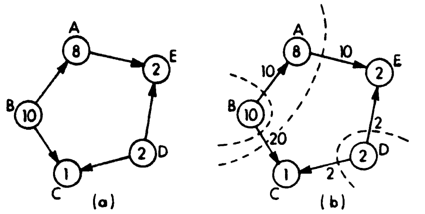


**Figure 9(a):** *An undirected loop in the DFG.*
**(b):** *The retimed DFG.*

Let us first assume that there are no tokens in this loop. We want to assign queue capacities to the counter-clockwise (CC) edges first. From the cut-set procedure, the total time assigned to the counter-clockwise (CC) edges is the same as the total time assigned to the clockwise (C) edges. Note that the total time assigned to the C edges (or the CC edges) is greater than or equal to the sum of node times of nodes A, B, D, since they are the source nodes of the C edges. Therefore; if we add the total time of the CC edges and the total operation times of the sink nodes (in this case, they are nodes C and E) of the CC edges together, as implied by Eq.(5), the sum is greater than or equal to the loop time $T_L$. So, the queue assignment of the CC edges satisfies Eq.(4b), i.e.

$$S_{LCC} \geq \frac{T_L}{\alpha^*} - D_{LC}$$

We can show that the queue assignment for the clockwise C edges satisfies Eq.(4a), follows basically the same argument except exchanging the roles of C edges and CC edges.

Now let us consider the case when there are $D_{LC}$ tokens on some C edges in the undirected loop L. By the cut-set retiming procedure, for each token on a C edge a time-delay $\alpha^*$ is assigned on that edge initially. As a result, after the cut-set retiming, the total time of the CC edges is greater than or equal to (total source node times of the C edges) - $\alpha^* D_{LC}$. If we sum up this total time and all the sink nodes time of the CC edges, the sum is greater than or equal to $T_L - \alpha^* D_{LC}$. Therefore, Eq.(4b) is again satisfied. Of course, the counter-clockwise case can be proved similarly. Moreover, if the loop under consideration happens to be a directed loop, it can be simply regarded as a special case, and there is no need of any different treatment.

Next, we want to show that Eq. (4c) is also satisfied for all edges. This is true since t(e) is always greater than or equal to its source node time, T(source), and therefore t(e) + T(sink) is greater than or equal to T(source) + T(sink) of node e. It is then clear that Eq.(4c) is satisfied for all edges in the retimed DFG. Eq.(4d) is also satisfied because D(e) is included in the maximum operator in Eq.(5).

Finally, we note that since the queue capacities have to be integers, this quotient is rounded up by the ceiling operation. It is obvious that the queue capacity Q(e) must be greater than or equal to D(e) in order to accommodate all the initial data tokens on edge e. Therefore, the maximum of the above two integers is taken for the queue assignment. Q.E.D. []

### 6.3. Linear Programming Formulation

The cut-set queue assignment above is not minimal, in large part because the retiming is not unique. In general, there are many SFG retimings that will meet the constraint that each edge delay is greater than or equal to its source node computation time. To minimize the queues, we would like the "minimal retiming", or the retiming that minimizes the overall sum of the edge delays.

To specify a particular retiming, we can use notation similar the notation introduced for the synchronous retiming case.[6] This notation is equivalent to describing a complete retiming in terms of cut-sets around each individual node. A retiming is specified by a mapping R from graph nodes to numbers. The number assigned to each node corresponds to the delay added to the outgoing edges and subtracted from the incoming edges. The delay transferred to an edge by a retiming is R(source) - R(sink). Using this notation, we can reformulate the retiming as a linear programming problem as follows:

The cost function is the sum of the edge delays after retiming. With initial edge delay of $\alpha^* D(e)$, the retimed edge delay is $\alpha^* D(e) + R(source) - R(sink)$. Since $\alpha^* D(e)$ is a constant, we need to minimize:

$$\sum_{edges} \left\{ R(source) - R(sink) \right\} \qquad (6a)$$

under the constraint:

$$\textit{For all edges}:$$

$$R(source) - R(sink) \geq T(source) - \alpha^* D(e) \qquad (6b)$$

Once the minimal retiming is derived, then queues are assigned by equation (5).

## 7. Integer Programming Algorithms

Since queues must have integer lengths, the "linearized" methods of the previous section do not necessarily yield the absolute minimal queue assignment. To eliminate this small quantization error, we can deal directly with the integer quantities, expressing the problem in an integer programming formulation.

### 7.1. Simple Minimal Queue Assignment

The assignment of the minimum queue capacities on the edges of the DFG requires that the queue capacities must be the minimum numbers satisfying the conditions in Eq. (7a,b,c,d), i.e.

minimize $\sum_{e:\,edges} Q(e)$, under the linear constraints:

$$S_{LC} \geq \frac{T_L}{\alpha^*} - D_{LCC} \qquad (7a)$$

$$S_{LCC} \geq \frac{T_L}{\alpha^*} - D_{LC} \qquad (7b)$$

for any simple undirected loop L in the DFG. And,

$$S(e) \geq \left\lceil \frac{(T(source) + T(sink))}{\alpha^*} \right\rceil - D(e) \qquad (7c)$$

$$Q(e) \geq D(e) \qquad (7d)$$

for any edge e in the DFG.

The validity of this formulation simply follows Theorem 3. Note that here the variables are all integers, thus giving a integer linear programming formulation.

### 7.2. Minimal Queue Assignment with Initial Token Redistribution

In some instances we find that part of the queue capacity of an edge, needed to accommodate initial tokens, is never used in the rest of the computation after the initialization stage. In these cases a smaller edge queue capacity can support the computation after initialization. Use of this smaller capacity necessitates the redistribution of initial tokens so that all can fit into the reduced edge capacities. It turns out that this redistribution can be formulated along the same lines as we proposed for the integer programming formulation and can be integrated into a larger size integer programming problem to solve the real minimal queue assignment problem. Space limitations preclude us from including the details.

## 8. Applications of the Timing Analysis

### 8.1. Improving Processor Utilization

It is clear from our timing analysis that any node in the DFG can at best operate at the optimal period, i.e. once in $\alpha^*$ time. If $\alpha^*$ is large compared to the node computation times, the utilization of nodes is quite low. There are two ways to improve this situation. One way is to use the technique of *processor sharing*,[4] i.e. we use one real PE to simulate several nodes, so as to improve the utilization of PE's. For example, if the average node utilization in a DFG is only 1/3, i.e. all nodes compute for only one third of time in average, it is possible that we can group 3 nodes into one real PE to obtain full utilization of PE's. It is not trivial to partition the nodes in an arbitrary DFG and assign them to real PE's to achieve maximal PE utilization.

Another way to get around this problem of low node utilization is to change $\alpha^*$ by changing the node operation times in a DFG. Note that the operation of a node can be changed by using different hardware realizations. Then it is possible to reassign the node times of a DFG to obtain better $\alpha^*$, and thus better utilization.

### 8.2. A Lattice Filter Example

The SFG of a Lattice Filter is shown in Figure 10(a), and one equivalent DFG of the SFG is shown in Figure 10(b). The initial data tokens, queue capacities, and node operation times are also displayed.

We first apply the main theorem to determine the pipelining period of the DFG in Figure 10(b). The augmented flow graph, AFG, is shown in Figure 10(c). And the loop which has the maximum period $\alpha$ is also shown in Figure 10(d). From this loop, we obtain $\alpha$ = 11.

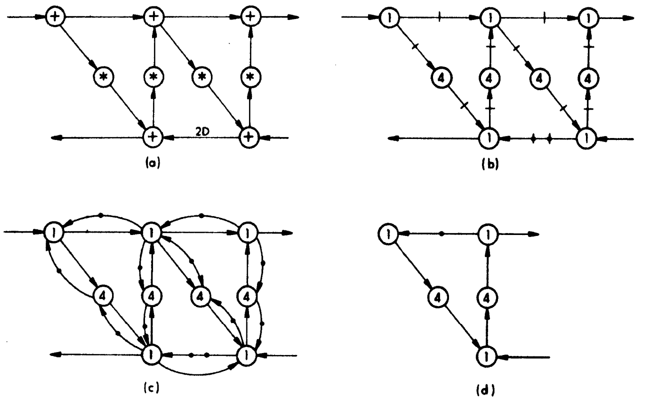


**Figure 10(a):** *a Lattice Filter SFG* **(b):** *one equivalent DFG of (a)* **(c):** *the AFG of (b)* **(d):** *the slowest loop in the AFG*

To assign minimum queues on the edges in order to achieve the optimal pipelining period, we first derive the retimed DFG of the Lattice Filter by the cut-set procedure. The retimed DFG of the Lattice Filter is shown in Figure 11(a). The optimal pipelining period $\alpha^*$ obtained by the cut-set procedure is 5.5. Applying Eq.(5) to all edges in the DFG, the assignment of minimum queues is shown in Figure 11(b).

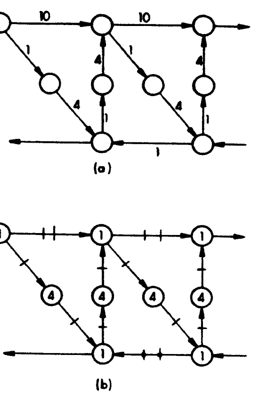


**Figure 11(a):** *the retimed DFG*
**(b):** *queue assignment for optimal period*

## 9. Conclusion

We have presented a systematic approach of converting a parallel algorithm described by a Signal Flow Graph to a functionally equivalent Data Flow Graph, which sets the stage toward its wavefront array implementation. Based on the notion of token and space (and their intriguing duality property), pipeline timing analysis and optimal queue design of generalized dataflow arrays are investigated. Of the four algorithms we proposed for the optimal queue assignment, the cut-set procedure is computationally easiest; however, it does not give an optimal or minimal assignment. The linear programming formulation gives closer approximation of the minimal solution at the expense of more computation. Finally, both integer programming formulations with or without initial token redistribution yield true minimal assignments at the cost of even more computation. It is expected that dataflow, wavefront, and fault-tolerant architectures will play a central role in the future VLSI or wafer-scale computing systems. It is our belief that, for these applications, the theorems and the timing analyses proposed in this paper will prove to be very useful.

## Appendix : Transforming a SFG to a Retimed DFG

Let us for the time being assume that each node in the SFG actually takes some known amount of time (though each node may take different times). To analyze the queue assignment, we first introduce a *retimed* DFG, in which the operation time of the node is assigned to its output edges. This is illustrated in Figure 12.

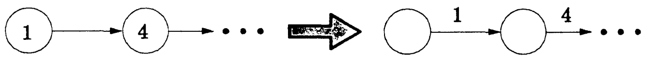


**Figure 12 :** *Assignment of node times to output edges in the retimed DFG.*

In order to synchronize the operations among nodes in the retimed DFG, edges will also be assigned extra time delays to serve the synchronization purpose. This assignment is based on the cut-set retiming procedure.[4]

### A Cut-set Retiming Procedure

We first define a cut-set for a SFG as: A ***cut-set*** in a SFG is a minimal set of edges which partitions the SFG into two parts. The retiming procedure is based on two simple rules:

(i) **Time-Rescaling:** All delays D may be scaled, i.e., $D \rightarrow \alpha$ , by a single positive number $\alpha$. Correspondingly, the input and output rates also have to be scaled down by a factor $\alpha$. The time-rescaling factor (or, equivalently, the slow-down factor) $\alpha$ is determined by the slowest loop in the SFG array.

(ii) **Delay-Transfer:** Given any cut-set of the SFG, we can group the edges of the cut-set into ***in-bound edges*** and ***out-bound edges***, depending upon the directions assigned to the edges. The Rule (ii) allows advancing k time-units on all the out-bound edges and delaying k time-units on the in-bound edges, and vice versa. It is clear that, for a (time-invariant) SFG, the general system behavior is not affected because the effects of lags and advances cancel each other in the overall timing. Note that the input-input and input-output timing relationships will also remain exactly the same only if they are located on the same side. Otherwise, they should be adjusted by a lag of +k time-units or an advance of -k time-units.

The optimal $\alpha^*$ for the retimed DFG is decided by the slowest loop in the SFG. In other words, $\alpha^*$ should be the minimum number such that after time scaling, there are enough delays to be distributed for all loops. Note that when there are no directed loops in the SFG, the optimal time interval between two input data, namely, $\alpha^*$ is the maximum of all node operation times. This is required because a pipelined system can not operate faster than the time needed for the slowest section in the pipeline.

### Procedure for Transforming a SFG to a Retimed DFG Array

A retimed DFG array is derived from the original SFG by the following procedure:

(1) Set initial $\alpha$ as the maximum of all node operation times. Scale all delays.

(2) For each target edge (edge that does not have more or equal amount of delays as required by the node it incidents from), find a cut-set containing the target edge and such that after suitable delay-transfer along the cut-set, the target edge will have the exact amount delay it needs, and at the same time, no new target edges are generated by this delay-transfer.

(3) If such a cut-set can not be found, a directed loop L which contains the target edge will be found. Rescale the $\alpha$, i.e. $\alpha = T_L / D_L$.

(4) Repeat the above step until there are no more target edges. The final $\alpha$ is the optimal pipelining period $\alpha^*$. []

The detailed algorithm for this has been previously published.[8] Only the important ideas have been outlined here.

### References

1. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Symposium*, pp. 256-282, SIAM, 1978.

2. S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Transactions on Computers, Special Issue on Parallel and Distributed Computers*, vol. C-31, no. 11, pp. 1054-1066, 1982.

3. J. A. B. Fortes, K. S. Fu, and B. W. Wah, "Systematic Approaches to the Design of Algorithmic Specified Systolic Arrays," in *Proc. IEEE ICASSP*, Tampa, Florida, 1985.

4. S. Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. IEEE*, vol. 72, pp. 867-884, IEEE, 1984.

5. S. Y. Kung, P. S. Lewis, and S. C. Lo, "On Optimally Mapping Algorithms to Systolic Arrays with Application to the Transitive Closure Problem," in *Proc. 1986 IEEE Int. Sym. on Circuits and Systems*, pp. 1316-1322, 1986.

6. C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Proceedings, Caltech VLSI Conference*, Pasadena, CA, 1983.

7. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Journal of VLSI and Computer Systems*, vol. 1, no. 1, pp. 41-68, 1983.

8. S. Y. Kung, S. C. Lo, and J. Annevelink, "Temporal Localization and Systolization of Signal Flow Graph (SFG) Computing Networks," in *Proc. SPIE, Real-Time Signal Processing*, San Diego, CA, 1984.

9. J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, pp. 48-56, 1980.

10. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

11. R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, vol. 14, no. 6, pp. 1390-1411, 1966.

12. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in *Second Ann. Sym. on Computer Architecture*, pp. 126-132, 1975.

13. J. D. Brock and L. B. Montz, "Translation and Optimization of Data Flow Programs," in *Proc. 1979 Int. Conf. Parallel Processing*, pp. 46-54, 1979.

14. R. Reiter, "Scheduling Parallel Computations," *J. ACM*, vol. 14, no. 4, pp. 590-599, 1968.

15. C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Trans. Software Engineering*, vol. SE-6, no. 5, pp. 440-449, 1980.

# PERFORMANCE ANALYSIS OF DATAFLOW SIGNAL PROCESSING ALGORITHMS

Leah H. Jamieson

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
and
Computer Science Laboratory
SRI International

Edward A. Ashcroft

Computer Science Laboratory
SRI International
Menlo Park, California 94025

Abstract -- This paper presents preliminary work on the analysis of dataflow algorithms. The focus is on developing techniques for modeling dataflow computation to obtain both theoretical and architecture-dependent performance estimates. Operator nets provide the basic model of a computation. The dataflow language used is Lucid. An example from the field of digital signal processing is used to demonstrate a theoretical data dependency analysis, analysis on an operator net and Lucid program, and performance figures derivable in terms of dataflow architecture models.

## Introduction

Dataflow is emerging as one of the important models of parallel processing, providing the potential to exploit the maximum concurrency inherent in an algorithm without requiring explicit statement of the parallelism. However, because the execution sequence is not expressed explicitly in the dataflow program, conventional algorithm analysis techniques for predicting the performance of an algorithm are not immediately applicable. In this paper, we use examples of analyses on a digital signal processing algorithm to demonstrate some analysis methods that provide both theoretical and architecture-dependent performance estimates.

## Algorithm, Language, and Architecture Models

Our model of a computation will be an *operator net* [2], which is a directed graph whose nodes represent operations or functions to be performed on the data items (datons) which arrive at the node via the incoming edges. An operator net is built up from subgraphs which specify subcomputations. The dataflow language used is Lucid [7], which allows a direct linear representation of operator nets. Lucid is a functional, definitional language in which the values of variables are infinite time-varying sequences. The structure of Lucid programs resembles the mathematical statement of a function definition. The model of a Lucid program is a filter which processes a time-varying infinite input stream, with the output of the program being the output stream which results from application of the filter to the input. Lucid programs therefore have a "built-in" time index. The model is in many ways well matched with operations in digital signal processing.

We will consider dynamic or tagged dataflow models [1, 7], in which each daton is associated with a tag, and the datons which are to be used together as the arguments of an operator or function are identified by matching their tags. The basic model of execution in such architectures is a ring. Data-driven architectures (e.g., the Manchester Dataflow Computer [3, 7]) are modeled by a single ring; demand-driven machines (e.g., the Eduction Engine [4]) are modeled by two intersecting rings, one for handling demands and one for applying operators to the results of demands; the hybrid Eazyflow Architecture [5] is modeled by intersecting data, demand, and application rings. The Eduction Engine and Eazyflow Architecture have been designed to perform direct execution of operator nets, expressed in linear form as Lucid programs.

## Analyses

Three types of analyses are demonstrated. A data dependency analysis based on the theoretical statement of the computation exposes the available parallelism. Analysis of the operator net and/or Lucid code provides a means of determining the parallelism that can be extracted from the program. From these analyses, requirements on the architecture can be derived.

The digital filtering computation

$$y_n = \sum_{k=0}^{p} a_k x_{n-k} + \sum_{k=1}^{q} b_k y_{n-k} \quad \text{for } n \geqslant 0 \qquad (1)$$

will be used to illustrate the analyses. We assume that the data arrival rate is $S$ (in signal processing terms, a sampling rate of $S$), the sample period is $\delta$, and that the data output rate matches the data input rate. (A faster data output rate is not possible; a slower rate implies eventual infinite buffering of inputs.) Except where specified otherwise, we will take the multiply-accumulate, denoted *mac*, ($sum = sum + w*v$, where $sum$, $w$, and $v$ are scalars) as the basic operation performed. In (1), $x_n = y_n = 0$ for $n < 0$.

### Inhłrent Parallelism

Conventional analysis of the data dependencies in the computation defines the maximum degree of parallelism attainable. Graphically, this can be represented by structures such as Petri nets. In many cases, this analysis can be performed directly on the mathematical statement of the problem. For example, in the digital filtering case, expansion of Eqn. (1) shows that each $x_i$ will be used in $P = p + 1$ terms and that each $y_i$ will be used in $Q = q$ terms. In any time interval $\delta$, then, it will be possible to initiate $P + Q$ *mac* operations. Because of the associativity of addition the *mac*'s can be performed in any order, so the only constraint on the computation is that imposed by the availability of the $x_i$'s and $y_i$'s. The maximum degree of parallelism (the number of *mac*'s that can be in progress simultaneously) will therefore be $\alpha(P+Q)$, where $\alpha = \lceil t_{mac} / \delta \rceil$ and $t_{mac}$ is the total time to perform one multiply-

This work was supported by NSF Grant Number DCR-84 13727, by funds from the DARPA Strategic Computing Program, monitored by the Office of Naval Research under ONR Contract Number N00014-85-C-0775, and by SRI International.

0190-3918/86/0000/0608 $01.00 © 1986 IEEE

accumulate. Depending on the architecture, $t_{mac}$ may represent only the actual time for the arithmetic operation or it may, in fact, include the complete cycle of instruction-fetch/decode/execute or token-match/instruction-fetch/execute.

### Operator Net and Lucid Parallelism

The operator net and Lucid program provide equivalent representations of the computation. In addition to representing the data dependencies, these forms also incorporate temporal dependencies introduced by the language constructs used. In particular, the effect of the program strategy and data structures used can be derived from the operator-net/Lucid representation. Since Lucid programs implicitly operate on time-varying sequences, the notion of "Lucid-time" is contained in operators such as **first** (which accesses the first element of the sequence), **next** (which yields the remainder of the sequence), **is current** (which temporarily freezes time with respect to a specified variable), **asa** (which selects a value *as soon as* a condition is met), and **fby** (which allows the construction of the time-varying sequences): and by built-in variables such as **index** (which is automatically incremented by one for each tick of Lucid-time). The operator net reveals the temporal relation between portions of the computation. Incorporation of timing constraints introduced by the use of Lucid-time constructs allows analysis of individual code blocks. Combining these analyses with information about data input arrival rates allows derivation of the potential parallelism in the Lucid program.

Fig. 1 shows two different Lucid code segments to compute the steady-state output of Eqn. (1). Given the notion of data streams (and lacking the notion of stored, randomly accessible vectors) and the capabilities of Lucid, it is easier to access "future" data items in the input or output streams than to access "past" items, so the computation has been reformulated as

$$y_{n+q} = \sum_{k=0}^{p} c_k\, x_{n+q-p+k} + \sum_{k=0}^{q-1} d_k\, y_{n+k} \text{ for } n \geqslant -q \quad (2)$$

where $c_i = a_{p-i}$ for $0 \leqslant i \leqslant p$ and $d_i = b_{q-i}$ for $0 \leqslant i < q$. Zero-padding of $x$ and $y$ ensures correct alignment of the two sums (see Fig. 1).

**Implementation Using Recursive Function Calls**

```
//input = x; zpad_x = x with P-1 zeros prepended
//output = y; zpad_y = y with Q zeros prepended

y where y = xsum(c,zpad_x,P) + ysum(d,zpad_y,Q)
   where
     xsum(coeff,z,P) =
       f(coeff,z,first P) fby xsum(coeff,next z,first P);
     ysum(coeff,z,Q) =
       f(coeff,z,first Q) fby ysum(coeff,next z,first Q);
     f(w,v,N) = sum asa index eq N
       where sum = 0 fby sum + w * v; end;
   end;
 end
```

**Non-Recursive Implementation**

```
//A from I is <A_I ,A_I+1,A_I+2, · · · >

y where
   y = g(c*X,P) + g(d*Y,Q)
     where
       I is current index;
       X = zpad_x from I;
       Y = zpad_y from I;
     end;
   g(s,N) = sum asa index eq N
     where sum = 0 fby sum + s; end;
 end
```

Fig. 1. Portions of two digital filter algorithms.

Fig. 2 shows the operator net for the *xsum* and *ysum* functions from the recursive version, where *f* can be specified by a subgraph representing the operator net for the function *f*. The operator net reveals the temporal relation between the two major portions of the computation. In terms of the dataflow model, the computation of *f* can begin immediately. After a delay of one time unit $\delta$ (corresponding to advancing one time unit in the $x$ or $y$ data stream) the *xsum* (or *ysum*) function will be enabled, allowing another *f* computation to begin. The operation of *f* is constrained by the sequential character of the input and by the **fby** construct to proceed linearly. (Thus, although the associative addition allowed accumulating the terms in any order in the theoretical analysis, the Lucid model imposes a linear order on the computation.) In general, then, $t_f = (N-1)\delta + t_{mac}$. Except for the case of $N$ small and $t_{mac} \gg \delta$, $t_f = \lambda N \delta$ for some $\lambda$. Combining the analysis of *f* with the enabling pattern of the *xsum* or *ysum* functions yields Fig. 3, where the length of the rectangles is $\lambda N \delta$. Once steady state has been achieved, taking a horizontal cross section across the graph shows the number of instances of *xsum* (or *ysum*) that can proceed simultaneously. Including the addition that combines *xsum* and *ysum* gives an approximate steady state potential parallelism of $\lambda(P+Q)+1$.

In the non-recursive implementation, the fact that the filter moves across the signal is contained completely in the implicit advancing of Lucid-time. At each time step $I$, time is frozen and an output $y$ value is computed using $X = zpad\_x_I, zpad\_x_{I+1}, \cdots$ and $Y = zpad\_y_I, zpad\_y_{I+1}, \cdots$. As in the recursive implementation, a new computation can begin upon arrival of a new $x$ value or completion of a new $y$ value, i.e., at each interval $\delta$. In function $g$, the **fby**

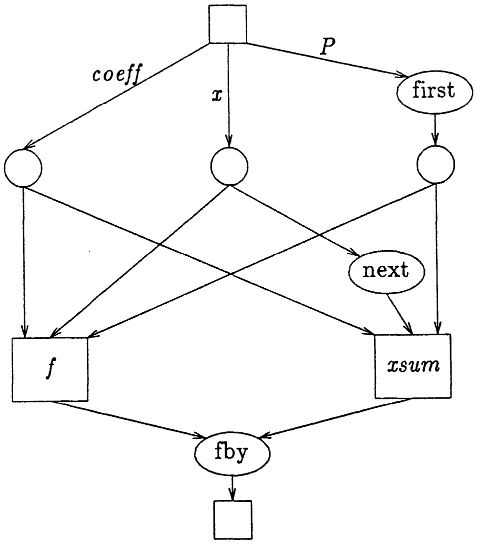


Fig. 2. Operator net for function *xsum*.

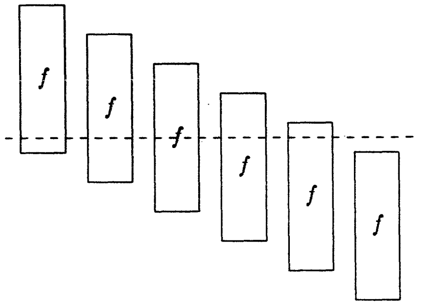


Fig. 3. Time sequence of *xsum* computations, shown for $\lambda N = 5$, in time units of $\delta$.

forces linear operation, with execution time proportional to $P$ (for $g(c^*X, P)$) or $Q$ (for $g(d^*Y, Q)$). The analysis is similar to that for $f$ above. Combining the implicit advancing of the computations with the analysis of $g$ yields the same execution characteristics as shown in Fig. 3 for the implementation using recursive function calls. In the non-recursive case, the basic operations are additions and multiplications rather than multiply-accumulates. The asymptotic behavior will be the same for the two cases; specific differences will depend on factors such as how the recursion and how the tagging requirements arising from the **is current** construct are implemented.

### Architecture Analysis

We consider analyses relating the dataflow architecture to the Lucid algorithm. The characteristics of the algorithm are used to derive requirements on the architecture in order to achieve the desired performance. This entails combining the constraints imposed by the input arrival rate, desired output rate, and the computational complexity characteristics obtained via the analysis of the Lucid program. The resulting model of the progression of the algorithm's execution can be combined with the architecture model to derive the processing rates required at each component of the architecture. For simplicity, we use a data-driven model, shown in Fig. 4. The computation can proceed essentially as outlined above. Assuming equal data arrival and output rates of $S$ samples per second, the minimum required processing rates shown in Table 1 can be derived. The "$S(P+Q+1)$" terms are due to the computation steps; the "$S$" terms account for the input. The fundamental operation is taken to be a 3-operand multiply-accumulate. If more basic arithmetic operations and 2-operand matches are used, the entries due to computation approximately double.

Table 1. Processing Rates for Dataflow Ring (sec$^{-1}$)

| | |
|---|---|
| Switch | S(P+Q+1)+S tokens |
| Token Queue | S(P+Q+1)+S tokens |
| Matching Store | S(P+Q+1)+S matches |
| Instruction Store | S(P+Q+1) accesses |
| Processors | S(P+Q+1) operations |

The operations count in the processors implies that in time $\delta$, $P+Q+1$ operations must be performed. The parallelism analysis has shown that this may be done in as many as $P+Q+1$ processors. Depending on $\delta$ and $t_{mac}$, fewer processors may suffice. The more stringent requirement is on the operation of the matching store. If $P$, $Q$, and $S$ are such that the matching store cannot function at the required rate, multiple submachines, such as those proposed for the Eazyflow Architecture [5] and the next generation Manchester system [3], could provide the needed capability.

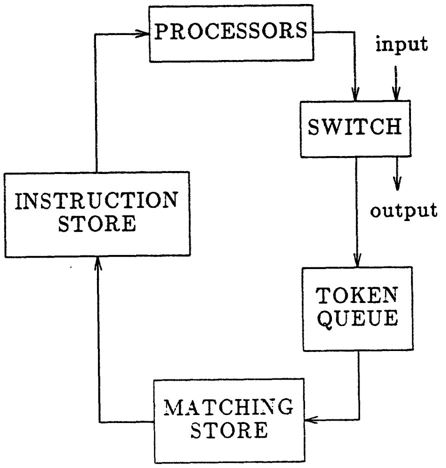


Fig. 4. Ring of a data-driven architecture [7].

### Summary

This paper addresses the problem of analyzing signal processing algorithms for dataflow architectures. The dataflow language Lucid differs significantly from traditional computer languages, just as the dataflow model of computation differs significantly from the von Neumann serial computer. Development of efficient signal processing algorithms for Lucid/dataflow execution therefore requires new algorithm approaches, programming strategies, and analysis techniques. We present some analysis tools which allow us to obtain design parameters for dataflow architectures targeted for signal processing applications.

*Acknowledgment: The authors thank G. B. Adams III for his help in preparing the paper.*

### References

[1] Arvind and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," in *Formal Description of Programming Languages,* IFIP Working Group 2.2, 1977.

[2] E. A. Ashcroft and J. Jagannathan, "Operator Nets," in *Fifth Generation Computer Architectures,* J. V. Woods, ed., North Holland, Amsterdam, 1986, pp. 177-201.

[3] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *CACM,* Vol. 28, Jan. 1985, pp. 34-52.

[4] B. M. Huey, A. A. Faustini, and E. A. Ashcroft, "An Eduction Engine for Lucid - A Highly Parallel Computer Architecture," *4th An. IEEE Phoenix Conf. Computers and Communications,* Mar. 1985, pp. 156-160.

[5] R. Jagannathan and E. A. Ashcroft, "Eazyflow: A Hybrid Model for Parallel Processing," *1984 Int. Conf. Parallel Processing,* Aug. 1984, pp. 514-523.

[6] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language,* Academic Press, London, 1985.

[7] I. Watson and J. Gurd, "A Practical Data Flow Computer," *Computer,* Feb. 1982, pp. 51-57.

# A MODEL OF QUANTITATIVE ANALYSIS FOR PERFORMANCE EVALUATION OF STATIC DATA FLOW COMPUTERS

Liu Gui-zhong and Ci Yun-gui
Department of Computer Science
Changsha Institute of Technology
Changsha, Hunan, P.R. China

Abstract — First, a model of static data flow computer and a model of data flow graph are proposed, then a model of system is presented for calculating practical parallelism degree with overhead of instruction execution on data flow computers as its parameter. From the computation, we can conclude that the maximum practical parallelism degree of a program running on a static data flow computer is determined with MP/OH (MP is the mean parallelism degree of a program, Oh is the overhead of instruction execution on the computer ). Therefore the overhead has great influence on the performance of a data flow computer.

## Introduction

The main purpose of research on data flow computers is exploiting parallelism in programs and speeding up their running. Instructions on data flow computers are driven by data, therefore data flow computers have great potential power in exploiting parallelism. However, progress in studying data flow computers is not great. In recent years, some people have criticized the approach of data flow. They point out that one of its problems is the great overhead, such as communication overhead in packet switched networks, extra accessions of memory, etc. While other people don't think the overhead is a problem. They consider a data flow computer as an asynchronous pipelined structure, and believe that if there are enough instructions available, each part of the computer can run efficiently even though the overhead is great.

In this paper, an approach is presented for studying the overhead. First, a model is given for a class of static data flow computers, then the relationship between the overhead of instruction execution and the parallelism degree on such a computer is established using the theory of stochastic process. The computed results show that for an user program, the practical maximum parallelism degree is only limited by the overhead of instruction execution.

## 1. MODEL OF STATIC DATA FLOW COMPUTERS AND MODEL OF DATA FLOW GRAPH

A typical structure of static data flow computers is shown in fig. 1. This is in fact a macro pipeline consisting of several units performing different operations. In such a system, many auxiliary operations are introduced, such as fetching instructions, sending them to processors through switched network, constructing result packets, sending packets back through network, storing packets into memory, and arbitrating whether instructions can be fired, etc. The so called EFFECTIVE OPERATION means the operation which directly do what the opcode of an instruction indicates. For example, the effective operations of arithmetic instructions are to perform arithmetic operations. The OVERHEAD of an instruction execution is all of its auxiliary operations. In this paper, we suppose the effective operation of each instruction takes one unit of time. The overhead of instruction execution is measured in units of time. In general, the overhead of instruction execution depends on what the instruction is and what environment the instruction is executed in. Here, the overhead practically refers to the average value of overheads of all instructions. In the following, the overhead of instruction execution is briefly called overhead, denoted by OH.

The model of pipelined structure for studying overhead is shown in fig.2. The N processors in the model, which represent N processors in a system, execute only effective operations of instructions in one unit of time. The n pipelines correspond to delays of auxiliary operations. The number of stages in each pipeline represents the corresponding value of overhead. Suppose the delay of one stage is one unit of time, then the number of stages of a pipeline is OH, $OH \in \{1,2,3,\ldots\}$. The memory in fig.2 is an idealized unit with unlimited size and accessing time 0. Its delay is considered as some of the stages of the pipelines.

A data flow graph is a directed cyclic graph. In fig.3, a data flow graph for computing N! is depicted. In order to describe the dynamic behaviour of a program, the data flow graph of the program should be unfolded to become an acyclic graph. That is, each function call should be replaced by its data flow graph and each loop should be unfolded. Therefore, no cycle exists in the graph obtained (the graph is an infinite one, if its program running will not stop). In such a graph, each node represents an instruction.

A pseudo-node, which is named start node and denoted by sn, and which emits all arcs to nodes that input tokens from outside of the graph, can be added to an unfolded graph, as shown in fig.5. Such a graph is called an extended data flow graph. In the flowing, we will only discuss this kind of graphs and call them data flow graphs(DFG) in brief. Suppose the length of each arc in such graphs is one.

DEF.1.1 Level of a node. The level of the start node is zero, the level of any other node is the length of the longest path to it from the start node.

DEF.1.2 Parallelism degree of level j of a program (j is a positive integer) $PD_j$ is the number of nodes at level j in its extended data flow graph. Parallelism degree of a program, PP is

$$PP \triangleq \begin{cases} \sum_{j=1}^{m} PD_j / m & \text{m is the maximum level of the graph} \\ \lim_{m \to +\infty} \sum_{j=1}^{m} PD_j / m & \text{if the graph is infinite} \end{cases}$$

DEF.1.3 Balanced data flow graph. If there are two or more paths from the start node sn to a node in a data flow graph, then their lengths

0190-3918/86/0000/0611 $01.00 © 1986 IEEE

are the same. Such a graph is a balanced data flow graph, otherwise it is an unbalanced one.

When a balanced data flow graph is used to process a set of data in a pipelined way, high efficiency can be achieved. Unfortunately, most data flow graphs are unbalanced ones. In order to transform them into balanced ones, some identity instructions (I), which only delay tokens by one unit of time, should be inserted in some short paths, thus increasing their lengths. An unbalanced DFG is shown in fig.5. We insert an identity instruction into it, so that it becomes a balanced one, as shown in fig.6.

On balanced DFG, the nodes at level i receive data tokens only from level i-1. In other words, instructions at level i are only driven by tokens from level i-1. So, we propose an assumption.

ASSUMPTION ON DRIVING LEVEL BY LEVEL. On extended DFG, instructions at level i are only driven by tokens from level i-1.

For any unbalanced DFG, it is possible that an instruction at level i receives tokens not only from level i-1, but also from other levels. Because the tokens from level i-1 goes through the longest way from the start node sn to the instruction, it is mostly possible that the instruction is fired by this token. Therefore, this assumption is reasonable in general.

Let Object_No$_{j-1}$ denote the total number of result packets generated by all instructions at level j-1, and Operand_No$_j$ the total number of operands required by all instructions at level j. According to the previous assumption, we have

$$\text{Object\_No}_{j-1} = \text{Operand\_No}_j \quad j \in \{2,3,4,\dots,m\}$$

A stochastic process PD$_j$, $j \in \{1,2,3,\dots\}$ is used for describing the dynamic behaviour of parallelism degree of a program. Suppose that all PD s are independent, identically distributed random variables with distribution $N(MP, \sigma^2)$, where MP and $\sigma^2$ are the mean and the variance of PD$_j$ respectively, PD$_j$ is the parallelism degree of level j.

## 2. MODEL OF SYSTEM FOR COMPUTING PARALLELISM DEGREE

First, we make some assumptions on this system.

Assumption 2.1. The system has n processors. In other words, the maximum parallelism degree of the system is n. Let $\mathcal{N} = \{0,1,2,\dots,n\}$, is a set of all possible practical parallelism degree of the system.

Assumption 2.2. The maximum parallelism degree of a program is $P_{max}$. $P_{max} = MAX(P_1, P_2, \dots, P_m)$, $P_j$ is the parallelism degree of level j, m is the maximum level. Let $\mathcal{P} = \{1,2,3,\dots, P_{max}\}$, is a set of all possible parallelism degrees of the levels.

Assumption 2.3. The numbers of result packets produced by instructions at each level are independent, identically distributed variables with revised Poisson distribution.

Suppose the mean number of result packets produce by an instruction at level j is $\lambda_j$, the revised Poisson distribution $P^+(\lambda_j)$ is

$$P^+(\lambda_j) = \frac{\lambda_j^{x-1}}{(x-1)!} e^{-\lambda_j} \quad \begin{array}{l} x = 1,2,3,\dots \\ \lambda_j \in \{1,2,3,\dots,\lambda_{max}\} \\ \quad = MAX(\lambda_1, \lambda_2, \cdots, \lambda_m) \end{array}$$

Assumption 2.4. Any result packet is transmitted to each instruction at next level with identical probability.

Assumption 2.5. The numbers of operands needed by instructions at each level of a program are independent, identically distributed variables with revised Poisson distribution $P^+(\beta_j)$, where $\beta_j$ is the mean number of operands required by an instruction at level j.

DEF.2.1. A description vector of level i of a program is a k-tuple sequence $M_j$, $M_j = (a_{j1}, a_{j2}, \dots, a_{jk})$, $a_{ji} \in \mathcal{P}$, $1 \leq i \leq k$, $k = \lambda_{max} * MP$. $a_{ji}$ is the number of instructions requiring i operands. At beginning, $PD_j = \sum_{i=1}^{k} a_{ji}$.

According to assumption 2.5.

$$\text{Let} \quad a_{ji} = \frac{\beta^{(i-1)}}{(i-1)!} e^{-\beta_j}$$

$$M_j \in \underbrace{\mathcal{P} * \mathcal{P} * \cdots * \mathcal{P}}_{k} = \mathcal{P}^k$$

Let $\mathcal{M} = \mathcal{P}^k$, called a set of description vectors.

When a result packet is produced and passed to the next level, the Probability with which the result is sent to a set of instructions requiring i operands is $a_{ji} / \sum_{i=1}^{k} a_{ji}$ according to assumption 2.4. After that, the number of instructions requiring i operands decrements, whereas the number of instructions requiring (i-1) operands increments. The description vector $M_j$ is transformed to $M_j'$.

$$M_j' = \begin{cases} (a_{j1}, a_{j2}, \dots, a_{ji-1}+1, a_{ji}-1, \dots, a_{jk}) & i>1 \\ (a_{j1}-1, a_{j2}, a_{j3}, \dots, a_{jk}) & i = 1 \end{cases}$$

When the result packet is transmitted to a set of instructions requiring only one operand (i=1).one instructions in the set can be fired. The number of instructions $a_{j1}$ decrements $(a_{j1}-1)$,the total number of instructions at level j also decrements, and the number of instructions in the buffer increments. In other cases, the instructions at level j does not decrease. Obviously, when $\sum_{i=1}^{k} i * a_{ji}$ result packets are received at level j, all instructions at this level can be fired.

From the assumption on driving level by level, $PD_{j-1} * \lambda_{j-1} = PD_j * \beta_j$ then $\beta_j = \lambda_{j-1} * PD_{j-1} / PD_j$

DEF.2.2. The state of processors is an integer PR, $0 \leq PR \leq n$, n is the number of processors in the system.

DEF.2.3. The state of the pipeline is a vector PI with OH elements, $PI = (n_1, n_2, \dots, n_{OH})$, $n_i \in \mathcal{N}$, $1 \leq i \leq OH$. $PI \in \underbrace{\mathcal{N} * \mathcal{N} * \cdots * \mathcal{N}}_{OH} \triangleq \mathcal{N}^{OH}$

DEF.2.4. The state of the buffer is an integer BU, $BU \in \{0,1,2,\dots, m*P_{max}\}$

DEF. 2.5. A descriptor of a program with m levels is a m-tuple sequence $PG_m$, $PG_m = (M_1, M_2, \dots, M_m)$, $M_j \in \mathcal{M}$, $1 \leq j \leq m$, $PG_m \in \mathcal{M} * \mathcal{M} * \cdots * \mathcal{M} \triangleq \mathcal{M}^m$.

The model of system for computing parallelism degree is shown in fig.7, in which PR is the state of processors, PI the state of pipeline, BU the state of the buffer. The value of BU represents the number of instructions, which can be fired. but are waiting for free processors. The memory of the system is composed of two parts. One is a half infinite linear storage with locations 1,2,3, .... Each location can store a description vector, and location 1 stores $M_1$, location 2 stores $M_2$,... location m stores $M_m$, that is, m description vectors are stored in order from location 1 to m. The second part of the memory is a pointer Point.which points to a location that can be read or written currently. Point can stay at the original place or move one location to the right. Its initial is at the first location. Except the m locations storing $PG_m$, the locations in the memory all store blank vectors B, B = (b,b,b,...,b), b is a blank character.

DEF.2.6. The state of the system s for computing parallelism degree is a triplet (PR, PI, BU), in which PR, PI, and PU are the state of processors, the state of the pipeline, and the state of the buffer respectively (as def. 2.2,2.3,2.4).

Let $\mathcal{PS}$ be a set of system states,

$\mathcal{PS} = N*N^{OH} * \{0,1,2,\ldots,m*P_{max}\}$, $s \in \mathcal{PS}$.

DEF.2.7. The model of system for computing parallelism degree is a 5-tuple ($\mathcal{PS}$, $s_o$, $\mathcal{M}$, B, $\delta$), in which $\mathcal{PS}$ is the set of system states, $s_o$ is the initial system state, $s_o = (0, (0,\ldots,0), 0)$, $\mathcal{M}$ is the set of description vectors of a program, B is the blank vector, and $\delta$ is a map

$$\delta : \mathcal{PS} * \mathcal{M} \longrightarrow \mathcal{PS} * \mathcal{M} * \{L, A\}$$

L means that the Point does not move

A means that the Point advances one location to the right

$\delta$ is called the transform rule of the system state. We shall describe it with the algorithm TRANS as follows.

Supposition 1. $Poisson^+(\lambda)$ is a random number generator with revised Poisson distribution. $\lambda$ is the mean of the random numbers.

Supposition 2. $U(M_j)$ is also a random number generator, which give a positive integer i, $1 \leq i \leq k$, according to assumption 2.4.

```
Algorithm TRANS
if $M_j$ = B then stop else
  if $M_j = (0,0,\ldots,0)$ then j:=j+1 else
    begin
    RN:=PR; PR:=PI ;
    for i :=2 to OH-1 do $PI_i := PI_{i+1}$ ;
    if BU >= n then [$PI_{OH}$:= n; BU:=BU - n]
     else [$PI_{OH}$:=BU; BU:=0];
    Packet := 0;
    while RN>0 do (Packet:=Packet + $Poisson^+$
      ($\lambda_{j-1}$); RN:=RN-1] ;
    While (Packet>0 and $M_j \neq (0,0,\ldots,0)$) do
      begin
      i:=U($M_j$); $a_{ji} := a_{ji} - 1$;
      if i>1 then $a_{ji-1} := a_{ji-1} + 1$ else
        BU:=BU+1
      end
    end;
```

Suppose the system is executing instructions at level (j-1) in a data flow graph. After a unit of time, instructions in n processors of the system are all finished, each of which produces some result packets randomly. The total number of these result packets are stored into a variable Packet. At the same time instructions at the first stage of the pipeline PI is sent into the processor PR, and some instructions are fecthed into the last stage of the pipeline PI from the buffer BU. The result packets in Packet are passed to the instructions at level j with identical probability. The random number generator $U(M_j)$ produces a number i, then one result packet is transmitted into one of the instructions at level j, which needs i operands. If i >1, then the description vector $M_j = (a_1, a_2, \ldots, a_{i-1} + 1, a_i - 1, \ldots, a_K)$; If i=1, then one of instructions can be fired, and is passed to the buffer BU. After all the result packets (in Packet) are passed to the memory one by one, $M_j$ is changed to $M_j'$, BU to BU', and system goes into a new state.

When the model is running, the values of PR and the number of steps of the system proceeded are accumulated, and then the practical parallel degree can be calculated as $\sum$PR/steps.

Before running the model, the mean of parallelism degree of a program, its varience, and the overhead of instruction execution are set as parameters. When the system finishs, the relation among the overhead, parallelism degree, and the number of processors can be determined.

## 3. RESULTS COMPUTED AND DISCUSSION

According to the model, some results computed for various parameters have been obtained. And two sets of curves are depicted in fig.8,9.

In fig.8, we suppose the communication overhead is in proportion to log N, so OH = $C_1 + C_2 \log N$, where $C_1, C_2$ are integers ($C_1, C_2 > 0$). Since packets pass through the networks two times, $C_2$ equals to two.

Each curve in fig.8 has a maximum value. After the maximum value is reached, the practical parallelism degree falls, the efficiency of processors decreases, although the number of processors N in system increases.

The table 1. shows the relationship between the maximum practical parallelism degree and other variables.

In fig.9, a set of curves of relations between practical parallelism degrees and parallelism degrees of systems with OH as parameter are shown. where on each curve there is also a maximum value of practical parallelism degrees.

From the model of the system, we can see that in order to obtain high efficiency, the pipeline representing overhead must be filled up with OH*N enabled instructions. If MP<OH*N, all instructions at one level are not enough for filling the pipeline. As a result, the practical parallelism degree decreases.

When MP of a program and OH of a system are given, the maximum practical parallelism degree is equal to or less than MP/OH. In the following, we give the explanation on this.

(1) Suppose MP<OH*N.

Since MP instructions are executed during OH steps, practical parallelism degree is MP/OH;

(2) Suppose MP$\geq$OH*N

Obviously, the practical parallelism degree is the possible maximum parallelism degree of a system, N. Since MP>=OH*N, then N<=(MP/OH).

Therefore, the practical parallelism degree is not greater thanMP/OH. Consequently, when parallelism degree of a program is a constant, its practical maximum parallelism degree is only determined by the average overhead of instruction execution on a system and have nothing to with the number of processors in the system. If we hope to speed up the running of a program on a system, only by means of increasing the number of processors, even at the cost of increasing the overhead, then the result may be countrary to our hope. The increasing of the overhead may cause the practical parallelism to be decreased, and the running time of programs increased.

The practical parallelism degrees computed from the above model is much less than MP/OH, because the execution of instructions enabled on data flow computers is in random order, which is not suitable for pipelines.

From the above computation, we can see that overhead has great influence on the performance of a data flow computer. If we want to increase the parallelism degree of data flow computers, we must be sure that the overhead does not increase quickly.

Obviously, although the conclution is obtained from static data flow computers, it is also useful for other types of multi-processor systems.

## REFERENCES

(1) Liu Gui-zhong and Ci Yun-gui, "Architecture of the Synchronous Dataflow System SDS-1," Computer Science and Technology No. 1 1986, China

(2) Dennis, J.B., and Misuna, D.P., "A Preliminary Architecture for a Basic Data Flow Processor," Proc. Second Ann. Symp. on Computer Architecture, IEEE, Jan. 1975, pp. 126-132.

(3) Dennis, J.B., "First Version of Data Flow Procedure Language," in Lecture Notes in Computer Science, 19, Springer-Verlag, Berlin, 1974, pp. 362-376.

(4) Dennis, J.B. and Gao, G.R., "Maximum Pipelining of Array Operations on Static Dataflow Machine," Proc. 1983 Int'l. Conf. Parallel Proc., August 1983.

(5) Gajski, D.D., Panda., D.A., Kuck, D.J., and Kuhn, R.H., "A Second Opinion on Dataflow Machines and Languages," IEEE Comp., Feb.1982, pp. 58-70.

(6) Arvind, Kathail, V., and Pingali, K., "A Data Flow Architecture with Tagged Tokens," Technical Memo 174, Lab. for Computer Science, MIT, Sept., 1980.

(7) Kai Hwang and Faye A. Briggs, Computer Architecture and Parallel Processing. McGraw-Hill Book Comp. 1984.

(8) Stone, H., "Optimal Partitioning of Randomly-Generated Distributed Programs," Tutorial at Changsha Institute of Technology, China, 1984.

(9) Kai Hwang, "Multiprocessor Supercomputers for Scientific/Engineering Applications," IEEE Comp., June, 1985, pp. 57-73.

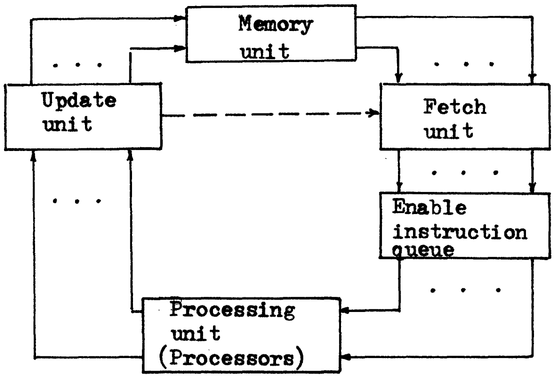


Fig. 1

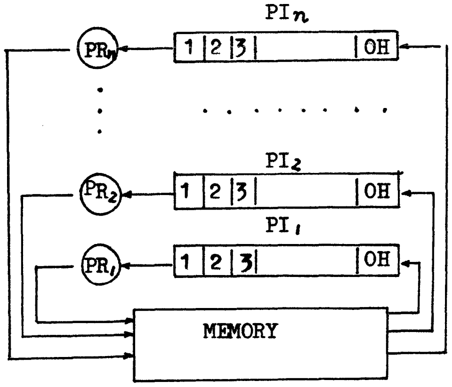


Fig. 2

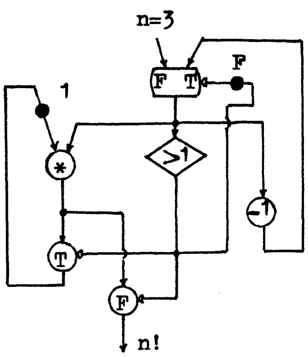


Fig. 3 data flow graph

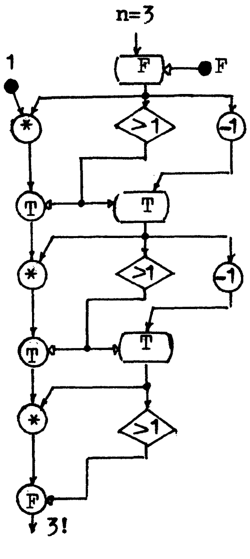
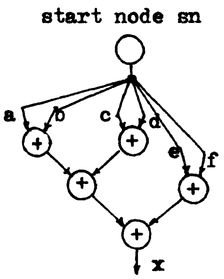


Fig. 4 unfolded DFG



x=a+b+c+d+e+f

Fig. 5 extended DFG

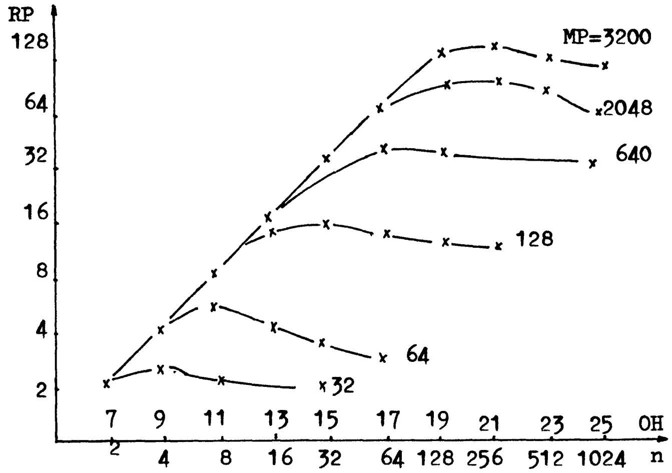
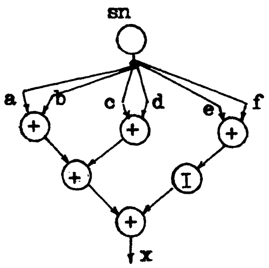


Fig. 8



I:(identity instruction

Fig. 6 balanced DFG

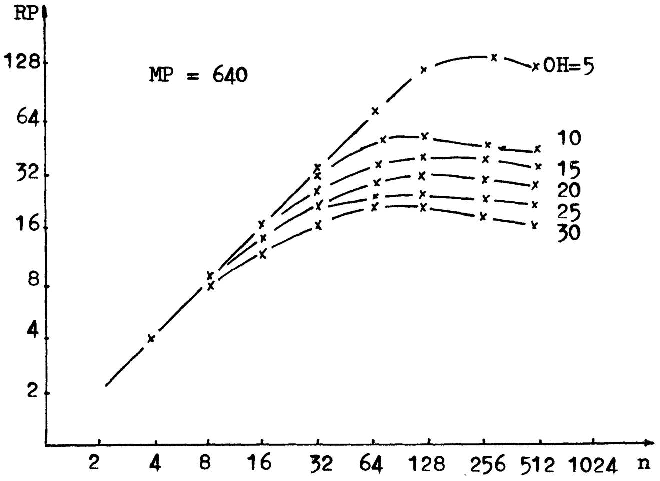


Fig. 9

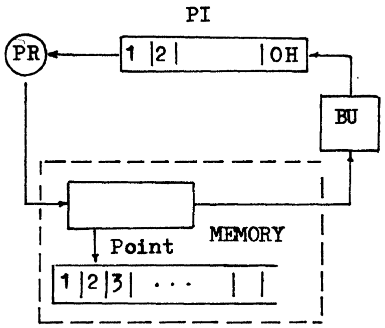


Fig. 7 model of system

Table 1

| parallelism degrees of programs | 32 | 64 | 128 | 256 | 512 | 640 | 1024 | 2048 | 3200 |
|---|---|---|---|---|---|---|---|---|---|
| OH | 9 | 11 | 13 | 15 | 15 | 17 | 19 | 19 | 21 |
| parallelism degress of systems : N | 4 | 8 | 16 | 32 | 32 | 64 | 128 | 128 | 256 |
| maximum parallelism degrees | 2.5 | 4.8 | 6 | 14.6 | 26 | 36 | 45 | 83 | 128 |

# USING FACTS FOR IMPROVING THE PARALLEL EXECUTION OF FUNCTIONAL PROGRAMS

Alberto Pettorossi
IASI-CNR
Viale Manzoni 30
00185 Roma (Italy)

Andrzej Skowron
Institute of Mathematics
Warsaw University
PKiN IX p.907
00-901 Warsaw (Poland)

University of North
Carolina at Charlotte
Department of Computer
Science
Charlotte, NC 28223

Abstract -- We address the problem of improving the efficiency of the parallel evaluation of functional programs. We assume that those programs are evaluated by a set of concurrent agents which communicate with each other and cooperate together while the computation progresses. Communications among agents improve the performance, because properties or facts about the functions to be computed are exploited. In particular we show that those communications may avoid redundant computations of intermediate results.

## 1.Introduction

Functional languages have been advocated as a formalism for expressing algorithms which allow to overcome the limitations imposed by the von Neumann computer architecture [1]. Applicative expressions in fact, can be evaluated in a parallel way by a set of concurrent computing agents, because there is a unique value associated with any subexpression, independently from the context in which it occurs (by the referential transparency property). That context independence property allows us to compute the various subexpressions in a parallel way by assigning each of them to an individual computing agent.

In order to fix our ideas and to introduce our Hope-like notations [3], let us consider the following simple program for computing binomial coefficients:

$$\text{BO:}\quad \begin{cases} \text{bin}(n,0)=1 \\ \text{bin}(n,n)=1 \\ \text{bin}(n,m)=0 \quad \underline{\text{if}}\ n<m \\ \text{bin}(n+1,m+1) = \text{bin}(n,m)+\text{bin}(n,m+1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \underline{\text{if}}\ n> m \end{cases}$$

The computation of the function bin(n,m) can be informally described as a rewriting of sets of agents.(The formal definition of a generic computation will be given later). For the time being,an agent can be thought of as a pair consisting of a string, i.e.,the name of the agent, and an expression, i.e.,the expression which the agent has to evaluate.

We may assume that initially there is the agent ε::bin(n,m), which can be rewritten into the set of agents:

{ε:: .0+.1, 0::bin(n-1,m-1), 1::bin(n-1,m)}
where n>m≥1.

In the rewriting we see that:i) new agents are generated by the recursive calls of the program equations, ii) the left and right sons of the initial agent ε have names 0 and 1 respectively, and iii) .k denotes in any given expression the result of the computation of the agent whose name is k for k=0 or 1.

We assume that the convention for giving names to the agents is the following: the initial agent has name ε, and if the agent x has k+1 sons, i.e., it uses for the rewriting a recursive equation with k+1 recursive calls, the names of its sons are x0, x1, ..., xk, respectively, according to the left-to-right order of those recursive calls.

For our binomial function, in the second step of the computation the agents 0 and 1 generated by the agent ε, can be rewritten independently from each other, so that the computation of bin(n,m) can be more precisely viewed as a nondeterministic and parallel rewriting of sets of agents into new sets of agents (see figure 1). The nondeterminism consists in choosing one (or more) agents to be rewritten, and parallelism consists in rewriting all agents which have been chosen, according to the recursive equations of the program.

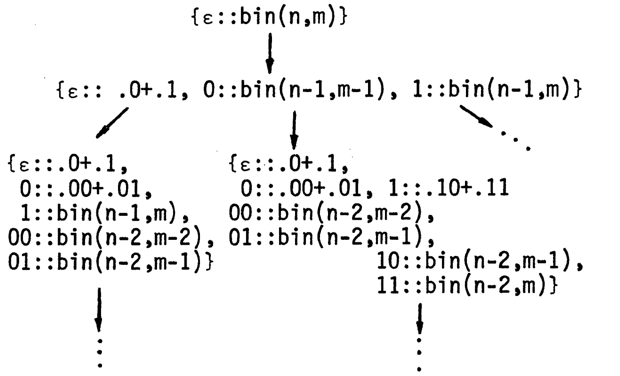


figure 1. Possible rewritings of the parallel computation of bin(n,m) when n>m≥2.

Notice that, according to the naming convention, each set of agents can be viewed as a tree-structured set, and therefore we may say that the computation progresses as a nondeterministic and parallel rewritings of trees of agents.

In this paper we will be concerned with the problem of improving performance of the trees of agents for the parallel evaluation of functional programs.

We will propose a method based on the discovery or knowledge of some "facts" concerning the functions to be evaluated. Those facts will be implemented as suitable communications among computing agents and the occurrence of those communications will realize the expected efficiency improvements.

Suppose, for instance, that we know the following fact about the binomial coefficients function: if n>m≥2 then the left son of the right son of bin(n,m) is equal to the right son of the left son of bin(n,m). In the language of facts which we will formally introduce later, we write:

bin(n,m)]01 = bin(n,m)]10

(see figure 2).

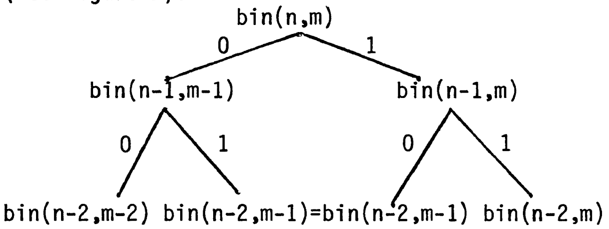


figure 2. A fact for the binomial coefficients function: bin(n,m)]01=bin(n,m)]10 where n>m≥2.

In the rewriting of the tree-structured set of agents we may then avoid the expansion of the agent 10 and allow the expansion of the agent 01 only. Once the value of bin(n-2,m-1) has been obtained, the agent 01 may send it to the agent 10, and save its computation. We will see in the sequel how that communication takes place and how it can be implemented by associating a local memory with each agent.

Avoiding the expansion of the agent 10 reduces the number of agents we need for performing the required computation of bin(n,m) and in most cases that saving is crucial,because in practice the number of available computing agents cannot be considered as unbounded.

There is another reason why the knowledge of relevant facts about recursive functions may drastically improve the efficiency. Suppose that during the computation of a given function we have used all the agents at our disposal. At that point we have to backtrack by deallocating the tasks from some agents which are leaves of the current tree of agents. The deallocation will make some agents available for the rewriting of other leaf agents and then more computation steps can be performed. It is desirable that backtracking steps are done in the most effective way, so to avoid,if it is possible, subsequent backtracking steps and at the same time, allow the maximum utilization of the agents at our disposal. Unfortunately the deallocation-allocation procedure in general cannot be optimized, if we do not know enough properties about the dynamical expansion of the trees of agents during the computation of the particular function we have to evaluate. In the binomial coefficients case, for instance, when computing bin(n,m) it may be better to expand the tree of agents so that we activate as soon as possible bin(m,m) or bin(n-m,0) because no extra agents are needed when computing those function calls.

In the Section 2 we will present a method for implementing facts about recursive functions and we will see how they can be translated into communications among agents.

In the subsequent Sections we will give a more formal presentation of the ideas introduced in Sections 1 and 2, and we will study some properties of the communications among agents.

## 2. The general scenario and a preliminary example

We consider the functional programs written as a set of recursive equations in a language L0 (to be defined later), and we assume that the programmer discovers or knows some useful facts about the functions to be computed. Those facts, written in the language of facts LF, are first submitted to a calculus which may or may not accept them. The role of the calculus is twofold: i) it makes sure that the facts are correct with respect to the given programs, and ii) it validates the efficiency improvements they determine, once implemented as communications among agents. Those two aspects of the calculus are very important, because otherwise we may derive, as we will see, erroneous or inefficient programs.

Facts which are accepted by the calculus are then translated by a translation algorithm, which produces a new set of recursive equations, written in a language L1, where it is possible to denote communications among function calls.

The general scenario of our approach is the one depicted in figure 3.

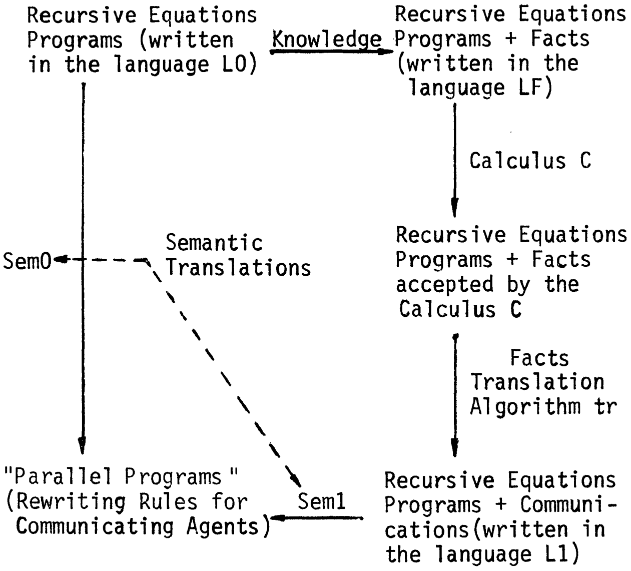

Recursive Equations Programs (written in the language L0)
Knowledge
Recursive Equations Programs + Facts (written in the language LF)
Calculus C
Recursive Equations Programs + Facts accepted by the Calculus C
Facts Translation Algorithm tr
Recursive Equations Programs + Communications(written in the language L1)
Sem0
Semantic Translations
Sem1
"Parallel Programs" (Rewriting Rules for Communicating Agents)

figure 3. The general scenario for improving the parallel execution of functional programs.

The scenario we presented can be considered as an extension of the ideas of the transformation system of Burstall and Darlington [2]. However in our approach the knowledge the programmer has about the functions to be computed, is expressed in a language LF of facts (not as "eureka steps" [2]). Moreover, the presence of the calculus which deals with the facts and ensures their correctness and their usefulness for improving efficiency, avoids the problems inherent to the Burstall-Darlington approach, where it may be that during the transformation process correctness is not preserved [8] and efficiency is not improved.

Notice that we explicitly deal with functional programs executed in a parallel way, so that the facts discovered by the programmer may refer to the behaviour of the associated sets of computing agents, but we allow only facts which can be written in a high level language LF. In this language it is possible to describe properties of sequences of expressions generated by the standard rewriting technique, so that the programmer is not involved in all the peculiarities connected with the computing agents behaviour. The system can accept facts if they are "true" facts and "efficiency improving" facts. Accepted facts about functional programs are used for increasing the efficiency of their parallel execution.

The semantics Sem0 and Sem1 are identified as 2 translations. Given a program P in L0 (or L1) Sem0 (or Sem1) produces a corresponding parallel program P'. Indeed, Sem0(or Sem1) defines a set of agents for evaluating the functions defined in P, and a set of rewriting rules which specify the concurrent behaviour of those agents.

The meaning of P' is given by a transition relation which defines for any given finite set of computing agents, i.e., a configuration, all possible future configurations. That transition relation is able to capture the nondeterministic and parallel execution of the program P' by constructing a tree of configurations as indicated in figure 1.

We will show that the translation Sem0 from a program P to P' (or the translation Sem1∘tr from P with some additional facts to P') can be done automatically for some classes of programs and facts.

The system is able to improve efficiency by automatically introducing suitable communications among computing agents. Those communications are derived on the basis of the given facts only.

It will be shown that the introduction of the intermediate language L1 in which functional programs with communications are written, allows us to have a simple calculus for checking the correctness of the communications among computing agents.

The language of parallel programs should be considered as a lower level language, which is understood by the system which realizes the parallel execution of functional programs via a set of computing agents.

The following example (given at schema level and taken from [4]) will clarify the main ideas we have presented so far.

Let us consider the program schema P:

$$\text{P:}\begin{cases}\text{f(x) = a(x)} & \underline{\text{if}}\ \text{p(x)=true}\\ \text{f(x) = b(x,f(c(x)),f(d(x)))} & \underline{\text{otherwise}}\end{cases}$$

Let us suppose that we discovered that for any x s.t. p(x)=false we have f(x)]01 = f(x)]10. That fact can be expressed as

f(c(d(x)))=f(d(c(x)))

and it means that the left-son call of the right-son call of f(x) is equal to the right-son call of the left-son call of f(x). That fact is then submitted to the calculus (whose definition will be given later). If it is accepted as a "true" fact and an "efficiency improving fact", it will be incorporated into the program P by the translation algorithm which produces the following program P1:

$$\text{P1:}\begin{cases}\text{f(x)=a(x)} & \underline{\text{if}}\ \text{p(x)=true}\\ \text{f(x)=b(x,f(c(x))(1}\ \underline{\text{comm}}\ \text{l),} & \\ \qquad\text{f(d(x))(0}\ \underline{\text{comm}}\ \text{l))}\underline{\text{decl}}\ \text{l} & \underline{\text{otherwise}}\end{cases}$$

The informal meaning of the expression "e decl l" is that a temporary memory location l, is kept during the evaluation of the expression e. That location is discarded when the evaluation of e is completed (see figure 4).

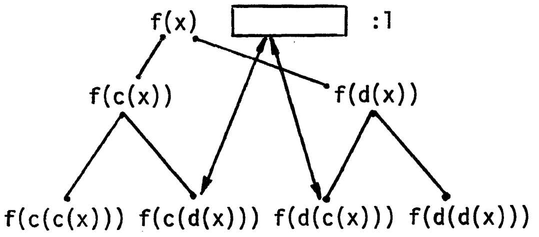


figure 4. The program for f(x) with communications via the location l.

The informal meaning of f(c(x))(1 comm l) (and analogously for f(d(x))(0 comm l)) is that the value of the right son of f(c(x)), i.e., f(c(d(x))) is stored in the location l. To be more precise, f(c(x))(1 comm l) means that i) during the evaluation of f(x), the agent which has to compute f(c(d(x))) may look at the content of the location l to get the result of its computation, and ii) at the end of the computation of f(c(d(x))), that value is stored in l. If a computing agent looks at the content of a location l and there is no value stored in it, its computation continues by activation of recursive calls.

From the informal description we have given, it is clear that efficiency improvements may take place, because before the end of the computation of f(d(c(x))), the corresponding agent may look at the location l and get the value it has to compute.

In particular, that agent may simply wait for the result of its computation to be written in l by the companion agent, which has to compute f(c(d(x))). If it does so, the generation of redundant agents is avoided, but we have to make sure that the agents do not wait for each other and no deadlock occurs.

Notice that the identification of a descendant call of a function f(x) is done via a string in $\{0,1,\ldots,k\}^*$. f(x)]ε denotes the function call f(x) itself, and f(x)]s.j recursively denotes the j-th son call of f(x)]s, for $s\in\{0,1,\ldots,k\}^*$ and $0\leq j\leq k$.

The calculus used in our approach is based on the symbolic evaluation of functions and it makes sure that for the given functions c and d, we have indeed: f(c(d(x)) = f(d(c(x))). The efficiency improvements due to the realization of the facts, are guaranted by the property that when a communication occurs, it is like making many rewritings of trees of agents in one step only.

The following new fact, called periodic redundancy in [4], can be discovered about the function f(x):

$$f(x)]0^{n+1} = f(x)]1^{m+1} \quad \text{for some m and } n\geq 0,$$

where $j^n$ denotes the string of n j's.

The above fact can be implemented as new additional communications between computing agents, and we get the following program P2:

```
    ┌ f(x)=a(x)                          if p(x)=true
    │
P2: ┤ f(x)=b(x,f(c(x))(1 comm l1)(0^n comm l2),
    │
    │          f(d(x))(0 comm l1)(1^m comm l2)))
    └                  decl l1,l2   otherwise
```

Notice that the programmer, when he discovers new facts, has only to suggest them to the system, and their translation into new recursive equations with communications is done automatically by the translation algorithm (at least for the facts we considered here, i.e., the ones which can be expressed as equality of functions calls). Indeed it is not difficult to see that one can derive the communication expressions of the form:

s comm l and decl l

starting from the strings denoting function calls in the language of facts.

The goal of our project is to build a system for recognizing different classes of facts concerning functional programs, and exploiting those facts for improving efficiency.

## 3. Parallel programs

In this section we introduce the notion of parallel programs [8-10].

AExp, MExp, Exp are sets of terms constructed in the standard way from constants, variables and function symbols.(The sets of constants, variables and function symbols are fixed in our considerations.) The elements of the sets AExp, MExp, Exp are called agent-name expressions, message expressions and expressions, respectively. If T is a set of terms then by CT we denote the subset of T s.t. t∈CT iff t is a term without variables.

An agent expression is a triple

< agn,msg >:: e

where agn∈AExp, msg∈MExp, e∈Exp.

A computing agent (or simply an agent) is an agent expression without variables.

Configurations are finite sets of agents. By CON we denote the set of all configurations.

A rule (or recursive equation) is an expression of the form:

lh <= rh if cond

where lh,rh are finite sets of agent expressions and cond is a boolean expression.

By r(x1,...,xk) we denote the rule r in which x1,...,xk are the only variable occurrences. If a1,...,ak are constants (of suitable types) then by r(a1,...,ak) (or r or lh <= rh if cond) we denote a concrete instance of r which we derive by substituting a1,...,ak for x1,...,xk in r.

A parallel program is any non-empty finite set of rules.

Let c,c'∈CON and let r be a concrete instance of the rule r and let us assume that:

$c \xrightarrow{r} c'$ holds iff cond is true and lh ⊆ c and c'=(c-lh)∪ rh .

$\xrightarrow{r}$ is called the transition relation of r.

The transition relation corresponding to a given sequence s= r1,...,rk of instances of rules is defined as the composition of the transition relations:

$\xrightarrow{r1}$ ,..., $\xrightarrow{rk}$ and it is denoted by $\xrightarrow{s}$.

The transition relation of a program P (denoted by $\xrightarrow{P}$) is defined as follows:

$c \xrightarrow{P} c'$ holds iff there is a non-empty finite sequence s of instances of rules in P (derived by the same substitution)s.t. for an arbitrary permutation s' of s we have:

$$c \xrightarrow{s'} c'.$$

As an example, let us consider the parallel program BO' for computing the binomial coefficients.BO' is a set of rewriting rules which implements BO.

```
     ┌{<x,E>::bin(n+1,m+1)}<={<x,E>::.x0+.x1,
     │                         <x0,E>::bin(n,m),
     │                         <x1,E>::bin(n,m+1)}
     │                                if n>m
     │{<x,E>::.x0+e, <x0,E>::n}<={<x,E>::n+e}
BO': ┤{<x,E>::e+.x1, <x1,E>::n}<={<x,E>::e+n}
     │{<x,E>::bin(n,0)}<={<x,E>::1}
     │{<x,E>::bin(n,n)}<={<x,E>::1}
     │{<x,E>::bin(n,m)}<={<x,E>::0} if n<m
     │{<x,E>::n+m}<={<x,E>::k} where k=n+m
     └{<x,E>::n-m}<={<x,E>::k} where k=n-m
```

where E is the empty message.The messages will play significant role in others examples, here they do not.

We see the above rules in action in the following computation of bin(3,2), where we write x::exp instead of <x,E>::exp and by ——> we denote the transition relation of BO'.

```
{ε::bin(3,2)}——> {ε::.0+.1,0::bin(2,1),
                          1::bin(2,2)}——>
{ε::.0+.1,0::.00+.01,00::bin(1,0),01::bin(1,1),
                                  1::1}——>
{ε::.0+.1,0::.00+.01,00::1,01::1}——>
{ε::.0+1,0::1+1}——>{ε::.0+1,0::2}——>
{ε::2+1}——>{ε::3}.
```

## 4. The translation functions Sem0 and Sem1.

This section is a technical section where we formally define the language L0 in which preliminary program versions are written, the language LF in which facts as equality of terms are written, and the language L1 in which recursive programs and facts are translated. We also present methods for translating programs in L0 (or L1) into parallel programs.

We will define a <calculus,fact-translation> pair for which facts accepted by the calculus can be correctly translated into communications among recursive calls. A correct fact-translation tr is the one which makes the diagram of fig.3 commute.

We will give an explanation of the various notions by discussing the problem of computing the binomial coefficients.

The language L0 in which the functional program for computing binomial coefficients is written is a Hope-like language [3]. An expression e of L0 is defined by

```
    e::= n|x|g(e,...)|f(e,...)
```

where n∊Constants, x∊Variables, g∊Basic_Functions and f∊Recursive_Functions.

A program P in L0 is a set of recursive equations each of which is of the form:

```
    f(e,...)=n (base case)
or
    f(e,...)=e' with f occuring in e' (recursive
                                        case).
```

For L0 expressions we assume a parallel and distributed evaluation using computing agents [9]. The translation Sem0 of functional programs in L0 into parallel programs is defined by the following list of rules:

1. Generation of sons.

```
   f(e0,...,ek)=g(...,f(e,...),...,f(e',...),...)
produces:          0            p
{<x,E>::f(e0,...,ek)}<={<x,E>::g(...,.x0,...,.xp,...),
                        <x0,E>::f(e,...),...,
                        <xp,E>::f(e',...)}
```

2. Base configurations.

```
   f(e0,...,ek)=n produces:
{<x,E>::f(e0,...,ek)}<={<x,E>::n}
```

3. Values to fathers.

```
{<x,E>::g(...,.xj,...),<xj,E>::n}<=
                       {<x,E> ::g(...,n,...)}
```

4. Basic functions evaluation.

```
{<x,E>::g(n1,...)}<={<x,E>::m} if m=g(n1,...)
```

Initial agent.For computing f(n1,...) the initial configuration {<ε,E>::f(n1,...)} is generated.

E is the empty message. It will play a significant role in Sem1, here it does not.

One can check that our parallel program B0' is the result of the above translation applied to B0.

In order to produce an improved version of the program B0 we may use our system by supplying it with "facts", i.e., properties of the computation of bin. We will restrict our attention to facts which can be expressed as equalities of agent expressions. They can be discovered by symbolic evaluation.

The syntax of the language LF of facts is as follows:

```
          e::=...(as in L0)| e]s  with s∊{0,1,...,k}*
          fact::=f(e,...)]s1=f(e,...)]s2.
```

A fact is checked by a Calculus and, once accepted, a Translation algorithm produces from it an improved version of the given program.

The Calculus for checking facts is given by the rewriting rules, which we will present for any program P0 in L1 with one recursive case only. Let P0 be :

```
 {f(e,...)=n1,...,f(e',...)=nk  (base cases)
 {f(e,...)=e' (recursive case).
```

The rules for the Calculus are:

```
 1. e]ε⊢—>e
 2. n]s⊢—>error  if s≠ε
 3. x]s⊢—>error  if s≠ε
 4. g(e0,...,ek)]js⊢—> if 0≤j≤k then ej]s
                                else error
 5. f(e0,...,ek)]s⊢—>if f(e0,...,ek)=e is an
                     instance of the recursive
                     case then e]s else error.
```

We write:

```
             e]s = e']s'
```

iff it is possible to reduce e]s and e']s' to the same expression (different from error) after applying to them a finite number of times the rules 1-5 and the rules of the Basic_Functions algebra.

For instance, in the program B0 we have for n≥2 and m≥2:

```
bin(n,m)]01⊢—>bin(n-1,m-1)]1⊢—>bin(n-2,m-1)
bin(n,m)]10⊢—>bin(n-1,m)]0⊢—>bin(n-2,m-1),and
   bin(n,m)]01=bin(n,m)]10.
```

The program produced by the translation is written in the following language L1. The syntax of L1 is like that of L0, with the following additions:

```
       e::=...|e(s comm l) with s∊{0,1,...,k}*,
                                l∊Locations.
```

The r.h.s. of an equation can be of the form: e or e decl l.

The parallel programs which we will get after the translation of programs in L1 have a special form of messages.

Messages are either E, i.e.the empty message or em≈L, where:

i) em is the empty elementary message ϕ or a constant elementary message n∊Constants,

ii) L is a set of names of son agents which may read or write the associated elementary message. L is represented as the list [s1,..., sn],where each si∊{0,1,...,k}*.

For instance, if the left son of an agent has to read/write the message of its father, that message will initially be: ϕ≈[0].

Now we are ready to define the translation of the set of programs in L1 into the set of parallel programs.

A program P in L1 is translated as a program in L0 (i.e.,rules 2,4 and the initial agent rule for Sem1 are like those for Sem0) with the following additions and changes:

1'. Generation of sons with communications.

```
f(e0,...,ek)=g(...,f(e,...),...,
                   0
                   f(e',...)(s comm l),...)decl l
produces:          j
{<x,E>::f(e0,...,ek)}<=
    {<x,ϕ≈L>::g(...,.x0,...,.xj,...),
     <x0,E>::f(e,...),...,<xj,E>::f(e',...),...}
where L={js| s comm l occurring in the j-th
                recursive call}.
```

3. Values to fathers.

```
{<x,m>::g(...,.xj,...), <xj,m'>::n}<=
{<x,m>::g(...,n,...),<xj,m'>::n}.
```

Notice that contrary to the previous case,after reporting to its father a son agent remains present. It may need to communicate its value to a message location.

5.Writing a message location.

```
{<x,ϕ≃L>::e,<xs,m>::n}<={<x,n≃L-{s}>::e,<xs,m>::n}
                                             if s∊L
```

Notice that after writing a location a son agent remains present because afterwards it may have to return its value to its father.

6. Reading a message location ("reading communications")

```
{<x,n≃L>::e,<xs,m>::e1}<=
      {<x,n≃L-{s}>::e,<xs,m>::n}  if s∊L.
```

The Calculus and the Translation algorithm enjoy the necessary properties with respect to correctness and efficiency of the derived programs, as the following results show.

Let *P0* be the class of all programs in L0 with 1 recursive definition only and let *P1* be the class of all programs in L1. We consider the class TR of translations from *P0* into *P1* such that if tr∊TR then tr adds s comm 1 and decl 1 annotations only.

Correctness Theorem for Communications. If for every program P0∊*P0* and s comm 1, s'comm 1 occurring in tr(P0) in the recursive call at position j and j' respectively, we have in our Calculus:

f(...)]js = f(...)]j's'

then tr is correct, i.e. for every P0∊*P0* the parallel programs Sem0(P0) and Sem1(tr(P0)) compute the same function. □

Remark. Sem1(tr(P0)) is more efficient than Sem0(P0) if some reading communications take place.

Let us consider the following program B1 in L1:

```
    ⎧bin(n,0)=1
    ⎪bin(n,n)=1
    ⎪bin(n,m)=0   if n<m
B1: ⎨bin(n+1,m+1)=bin(n,m)(1 comm 1) +
    ⎪             +bin(n,m+1)(0 comm 1)decl 1
    ⎪                          if n>m≥1
    ⎩bin(n,1)=n   if n>1
```

The program B1 is equivalent to B0, i.e., the parallel programs Sem0(B0) and Sem1(B1) compute the same function bin.This follows from the Theorem above and from the fact:

bin(n,m)]01 = bin(n,m)]10

accepted by the Calculus when n>m≥2.

The following parallel program Sem1(B1) is the result of translating B1:

```
{<x,E>::bin(n+1,m+1)}<={<x,ϕ≃[01,10]>::.x0+.x1,
                        <x0,E>::bin(n,m),
                        <x1,E>::bin(n,m+1)}
                                   if n>m≥1
{<x,mes>::bin(n,0)}<={<x,E>::1}
{<x,mes>::bin(n,n)}<={<x,E>::1}
{<x,mes>::bin(n,1)}<={<x,E>::n}  if n>1
{<x,mes>::bin(n,m)}<={<x,E>::0}  if n<m
{<x,mes>::n±m}<={<x,E>::k}       if k=n±m
{<x,mes>::e+.x1,<x1,mes1>::n}<=
                 {<x,mes>::e+n,<x1,mes1>::n}
{<x,mes>::.x0+e,<x0,mes1>::n}<=
                 {<x,mes>::n+e,<x0,mes1>::n}
{<x,ϕ≃[01,10]>::e,  <x01,mes>::n}<=
           {<x,n≃[10]>::e,<x01,mes>::n}
{<x,ϕ≃[01,10]>::e,  <x10,mes>::n}<=
           {<x,n≃[01]>::e,<x10,mes>::n}
{<x,n≃[01]>::e, <x01,mes>::e1}<=
           {<x,n≃[]>::e,<x01,mes>::n}
{<x,n≃[10]>::e, <x10,mes>::e1}<=
           {<x,n≃[]>::e,<x10,mes>::n}.
```

Let us consider an example of computation of the program Sem1(B1) with the initial configuration {<ε,E>::bin(5,3)}:

```
{<ε,E>::bin(5,3)}——>

{<ε,ϕ≃[01,10]::.0+.1,

 <0,E>::bin(4,2),     <1,E>::bin(4,3)}——> ...——>

          {<ε,ϕ≃[01,10]>::.0+.1

<0,ϕ≃[01,10]>::3+.01           <1,ϕ≃[01,10]>::.10+1

              <01,ϕ≃[01,10]>::3   <10,E>::bin(3,2)}
 A
——>       {<ε,3≃[10]>::.0+.1

<0,ϕ≃[01,10]>::3+.01           <1,ϕ≃[01,10]>::.10+1

              <01,ϕ≃[01,10]>::3   <10,E>::bin(3,2)}
 B
——>
          { <ε,3≃[]>::.0+.1

<0,ϕ≃[01,10]>::3+3             <1,ϕ≃[01,10]>::.10+1

                                  <10,E>::3 }
...
```

In the transition *A* the value computed by the agent 01 is communicated to the agent ε and in *B* this value is communicated to the agent 10 (by the agent ε). That communication avoids the computation of bin(3,2) by the agent 10.

Let us consider now a program FIB in L0:

```
     ⎧fib(0)=1
FIB: ⎨fib(1)=1
     ⎩fib(n+2)=fib(n+1)+fib(n)
```

and a program FIB1 derived from FIB and the fact

fib(n)]01 = fib(n)]10  (n>1)

```
      ⎧fib(0)=1
FIB1: ⎨fib(1)=1
      ⎪fib(n+2)= (fib(n+1)(0 comm 1)+
      ⎩               fib(n)W(ε comm 1))decl 1
```

where we use the annotation *W* to tell the system that the newly generated computing agent with the task fib(n) is forced to wait until the corresponding value appears in the location 1.

The reader can easily modify our language L1 and the fact-translation algorithm for coping with the new kind of annotations.

Fact. Let FIB1' be the result of application of that new translation algorithm to FIB1.The parallel program FIB1' uses a linear number of computing agents (with respect to n) in the computation with the initial configuration

{<ε,E>::fib(n)}.

The length of this computation is a linear function of n. □

In the <calculus,fact-translation> pair we use, it is possible to express facts as equality of terms, so that repeated evaluations of common subexpressions are avoided. The calculus is based on the unfolding rule [2] and the symbolic evaluation technique. The associated translation realizes communications among concurrent agents and improves the efficiency of their computations.

## 5. Another class of facts.

In this Section we will consider a new class of facts which turn out to be very useful for deriving efficient versions of parallel programs. As an example we will derive a program for finding the connected components of undirected graphs.

Some facts can be viewed as making "jumps into the future". They improve efficiency by identifying several subexpressions which can be evaluated in a parallel way, and by allowing us to know in advance the results of some concurrent computations.

Other facts have more sophisticated structure, but all of them can be discovered by analysing the behaviour of the given functional programs, maybe using a little induction.

We think that by developing more techniques similar to those presented in this paper it will be possible to construct advanced systems which (semi)automatically derive parallel programs with high performances (at least for some specific classes of problems).

In this Section we will not give all the technical details. We leave them to the reader.

We will suggest the construction of another <calculus,fact-translation> pair for the facts of the form "jumps into the future".The calculus is based on symbolic evaluation and induction, while the translation algorithm in most cases reduces to a straightforward inclusion of the discovered facts as new rewriting rules for the computing agents (see for instance, the following facts F1 and F2).

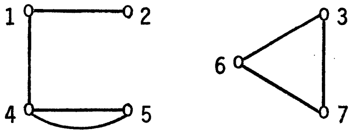
Let us derive the program for computing the connected components of a given graph.We are given the following functions:

```
dec succ: nodes ——> set nodes
    succ(n) computes {n}∪s, where s is the set of
    nodes adjacent to n.
dec f: set nodes——> set nodes
    f is the extension of succ to a set of nodes.
    f(nilset)=nilset
    f({n}∪̇s)=succ(n)∪f(s)  where ∪̇ denotes disjoint
                                      union.
```

Given a graph G, that is, a particular function succ, we may use the following function F to compute the connected components of G, by taking as input for F the set of singletons of the nodes in G.

```
dec F: set set nodes ——> set set nodes
    F(nilset)=nilset
    F({a}∪̇X)=F({f(a)}∪X)  if f(a)≠a
    F({a}∪̇X)={a}∪F(X)     if f(a)=a
    F({a,b}∪̇X)=F({a∪b}∪X) if a∩b≠nilset.
```

For instance, given the graph:



where the nodes are denoted by natural numbers, we get:

```
F({{1},...,{7}})=...
=F({{1,2,4},{2,1},{3,6,7},{4,5,1},{5,4},{6,7,3},
   {7,6,3}})=...
={{1,2,4,5},{3,6,7}}.
```

The reader can observe that if he translates the functional program defining F into a parallel program by applying the algorithm Sem0, he will not get an efficient implementation.
It is easy to see that the following fact holds:

(F1) F({a,b}∪̇X) = F({fa∪fb}∪X)  if fa∩fb≠nilset.

By using that fact we can simplify the rewriting of the expression:

F({{1},...,{7}}).

The system can implement that fact as a special rule of the form:

```
{x::a1...a...ak, y::b1...b...bm}<=
        {x::a1...fa∪fb...ak, y::b1...b̸...bm}
                                  if  fa∩fb≠ nilset
```

where we omitted some set brackets and b̸ denotes the erasing of b.

The computing agents with names x and y can communicate if fa∩fb≠nilset, and if this communication takes place, the result can be computed much faster. Also the following fact F2 holds:
(F2) F({a1,...ak}) = F({f̄a1,..,f̄ak})

where f̄ is equal to f or it is the identity.

The translation of that fact can improve the efficiency, because it allows the computations of the f̄ai's in a parallel way. However, in order to avoid the repeated evaluations of common subexpressions it is advisable to activate as often as possible the communications which derive from the fact F1. In a sense a good implementation of the program for F should balance the requirements from facts F1 and F2.

That objective can be achieved by implementing the following fact F3 [13].

Let us define first the following operation:

$$g(X,Y) = \{a \cup b \mid fa \cap fb \neq \text{nilset and } a \in X, b \in Y\}$$

$$\cup \{a \in X \mid fa \cap fb = \text{nilset for all } b \in Y\}$$

$$\cup \{b \in Y \mid fa \cap fb = \text{nilset for all } a \in X\}$$

where X,Y are families of finite sets of nodes.
We define also the operation C on finite sequences of families of finite sets:

$$C(X1,X2,X3,X4,\ldots)=[g(X1,X2),g(X3,X4),\ldots].$$

Finally we can formulate the following fact F3:

$$\text{(F3)} \quad F(\{\{n1\},\ldots,\{nk\}\}) = C^{p}(\{\{n1\}\},\ldots,\{\{nk\}\})$$

where $p=\lceil \log_2 k \rceil$.

Now we are ready to present the parallel program in which facts F1, F2 and F3 are implemented. As usual we will write it as a set of rewriting rules for computing agents.

Connected Components Program

```
1. {x0 :: a1...a...ak, x1 :: b1...b...bm}<=

   {x0 :: a1...fa∪fb...ak, x1 :: b1...b̸...bm}

                                     if fa∩fb ≠ nilset

2. {x :: a1...a...b...ak}<=

        {x :: a1...a∪b...b̸...ak}

                                     if a∩b≠nilset

3. {x0 :: a1...ak, x1 :: b1...bm, x:: .x0∪.x1}<=

        {x :: a1...ak b1...bm}

                  if rules 1. and 2. cannot be applied

4. {x :: F({a1,...,ak})}<={x :: .x0∪.x1,

                          x0 :: F({a1,...,a⌈k/2⌉}),

                          x1 :: F({a⌈k/2⌉+1,...,ak})}

5. {x :: F({a})}<={x ::{a}}
```

where instead of <x,m>:: we write only x:: because messages are not used. a,b,a1,...,ak,b1,...,bm are finite sets of numbers (i.e.,nodes of graphs).

## 6. Conclusions

Through some examples we presented various kinds of facts for improving the efficiency of the parallel evaluation of functional programs. We also presented the structure of a system which is essentially a <Calculus,Fact-translation> pair for making those improvements in an automatic way.Facts are to be accepted by the Calculus and then they are used by the translation algorithm to produce efficient parallel implementations of the functional programs to which they refer. We think that more work should be done in the direction we indicated here. In particular it would be important to develop a theory for the automatic improvements of functional programs and their implementations.

In a sense we follow the program derivation technique à la Burstall-Darlington, and we provide a framework for overcoming some difficulties encountered by those authors.In particular we propose an intermediate "language of facts",used by the programmer for writing some properties or facts of the program to be improved. Facts accepted by the calculus are indeed bound to increase the efficiency, and in this way we can make sure, contrary to the original Burstall-Darlington approach, that efficiency is improved when new versions are derived. Moreover the language of facts gives to the programmer freedom for expressing intuitions or discoveries, while the "eureka steps" à la Burstall-Darlington need to be expressed into the recursive equation language used for programs. Similarly, in our approach, the language of the new versions of the program may be be different from the language of the old versions. Therefore, efficiency improvements may also derive from better compilers one can construct for the language of the new versions. Our approach allows also for the incremental discovery of the facts to be incorporated into old program versions. Most of those facts can be interpreted as establishing some communications among computing agents, so that they work in a cooperative way while the computation progresses.

## References

[1] J. Backus, "Can Programming be Liberated from the von-Neumann Style ? A Functional Style and Its Algebra of Programs", J. A.C.M. Vol.21, No.8 (1978) pp.613-641.

[2] R.M. Burstall, J. Darlington, "A transformation system for developing recursive programs",J. A.C.M. Vol.24, No.1 (1977) pp.44-67.

[3] R.M. Burstall, D.B. MacQueen, D.T. Sannella, "HOPE: An Experimental Applicative Language", Proc. LISP Confer. Stanford Univ. (1980).

[4] N.H. Cohen, "Eliminating Redundant Recursive Calls", ACM Transactions on Programming Languages and Systems 5 (1983) pp.265-299.

[5] J.Darlington, M. Reeve, "Alice: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages",Proc. ACM/MIT Conference on Functional Programming Languages and Computer Architecture (1981).

[6] M.S. Feather, "A system for developing programs by transformations", Ph.D. Thesis, Univ. of Edinburgh, Scotland (1979).

[7] L. Kott, "About transformation system: A theoretical study", 3ème Colloque International sur la Programmation, Dunod, Paris (1978) pp. 232-247.

[8] A. Pettorossi, A. Skowron, "Complete Modal Theories for Verifying Communicating Agents Behaviour", Computer Science Dpt. CSR-128-83 Edinburgh Univ. (1983).

[9] A. Pettorossi, A. Skowron, "A Methodology for Improving Parallel Programs by Adding Communications", LNCS 208 Springer-Verlag (1985) pp. 228-250.

[10] A. Pettorossi, A. Skowron, "Theories for Veryfying Communicating Agents Behaviour in Recursive Equations Programs", Proc. of 20-th Annual Conference on Information Sciences and Systems, Princeton, March 19-21 (1986).

[11] W.L. Scherlis, "Expression Procedures and Program Derivation", Ph.D. Thesis, Stanford Univ. Computer Science Report STAN-CS-80-818 (1980).

[12] W.L. Scherlis, D. Scott, "First steps towards inferential programming", IFIP83, R.E.A.Mason (ed.) Elsevier Science Publ. North Holland (1983).

[13] U. Vishkin, "An Optimal Parallel Connectivity Algorithm", Research Report RC9149 (#40043) IBM (1981).

[14] B. Wegbreit, "Goal-directed program transformation", IEEE Trans. Software Eng. SE-2 (1976) pp.69-79.

[15] N. Wirth, "Program Development by Stepwise Refinement",C.A.C.M. Vol.14 (1971) pp.221-227.

# A NEW STRUCTURING MECHANISM FOR SUPPORT OF SPATIALLY REDUNDANT DISTRIBUTED COMPUTATION

R.S. Ramanujan
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, Minnesota 55418

J.G. Kuhl
Department of Electrical and
Computer Engineering
University of Iowa
Iowa City, Iowa 52242

Abstract--Spatial redundancy denotes computation characterized by simultaneous execution of redundant program elements on separate computing channels. This paper presents a mechanism called molecular activity for the support of spatially redundant distributed computation. It is shown that molecular activities extend the domain of existing software fault-tolerance techniques to include applications constructed as sets of communicating processes and involving access to global data. Molecular activities also allow exploitation of high-level parallelism in a large class of applications, particularly in the area of artificial intelligence.

## I. Introduction

Spatial redundancy is a term used to describe computation characterized by the simultaneous (or concurrent) execution of redundant program elements on separate computing channels. Some mechanism for reaching agreement among the redundant elements is employed to facilitate decisions regarding the results produced by the computation.

The concept of spatial redundancy has been employed to achieve tolerance of software failures through a notion known as a design diversity and a technique called n-version programming [1]. In a broader sense, spatial redundancy could also be used to exploit parallelism in the solution of certain important types of problems. Many computationally difficult problems are characterized by the existence of alternative algorithms or heuristics for obtaining a desired solution. Examples include certain planning problems in the field of artificial intelligence, as well as various VLSI design problems involving heuristic search of a large solution space. Although it is known that one algorithm or heuristic may perform significantly better than another for a particular instance of the problem (in terms of execution time or solution optimality), there is typically no way to determine a priori which one will perform best. Spatial redundancy could potentially be employed to carry out alternative strategies in parallel, with some mechanism being used to select the desired result (and perhaps preempt other still-executing alternatives).

In a distributed system environment, spatially redundant computation should ideally allow a redundant program element to be structured as a set of communicating processes residing at various nodes of the system. This would allow for exploitation of concurrency and parallelism within the individual program elements.

Thus, the implementation of spatial redundancy could lead to the existence of multiple distinct sets of communicating processes, each set representing one of the redundant elements. If it is allowed that several different distributed computations (each employing spatial redundancy) might exist simultaneously in a distributed computing system, the situation may become quite complex. Clearly, some technique for structuring the interactions between these computations is necessary.

In this paper we discuss mechanisms for structuring spatially redundant distributed computations. It will be shown that the conventional mechanism for enforcing data consistency among concurrent computations, the atomic action, is insufficient for supporting general spatially redundant distributed computation. New mechanisms known as atomic activities and molecular activities are proposed for this purpose. It is argued that these new mechanisms are necessary in order to extend the usefulness of existing techniques for structuring redundant computations, such as n-version programming, to cover a wide range of distributed applications.

## II. The Need for a New Structuring Mechanism

Distributed computation is characterized by programmed actions on data objects that may be located at various nodes of a distributed system. Data objects accessed by a distributed computation can be classified into two categories:

1. Local Data Objects--These are objects declared in, and local to, the program that specifies the computation. Their lifetime coincides with that of the computation.

0190-3918/86/0000/0624 $01.00 © 1986 IEEE

2. External (global) Data Objects--The lifetime of these objects supercedes that of the computation. They may represent permanent or semi-permanent information that exists independently of the various computations which access and manipulate them. The collection of data objects in a distributed database provides an example of external data objects.

One of the fundamental differences between internal and external data objects lies in the ability to statically determine which objects will be accessed and/or modified by a computation. Since local objects are declared within, and belong to, a computation, it is possible to enumerate all of the local objects that will be accessed by computation at the time when the program specifying the computation is written. However, the external data objects to be accessed, and the nature of that access, may be determined, at least in part, during the execution of the computation.

For instance, consider an application concerned with routing certain connections during the design of a VLSI circuit. This application must access a large database representing the layout of the circuit. However, the specific portions of the layout data to be traversed during an instantiation of the routing application, and the portions to be modified, can be determined only during the execution of the router.

The basic mechanism for maintaining the consistency of external data objects operated upon by concurrent computations is the atomic action [2]. An atomic action is some program-specified computation that reads or modifies the state of one or more data objects and appears as an indivisible operation from the point of view of computations outside the atomic action.

Built-in atomic actions greatly simplify a programmer's task in coping with unplanned concurrency and failures. The programmer of an atomic action can write his software without bothering about other computations and failures that may affect the data that his program manipulates. In recent years, a number of distributed systems supporting atomic actions have been implemented. However, the intoduction of spatial redundancy into distributed computation introduces new problems in consistency maintenance that cannot be addressed by the simple use of atomic actions.

Consider an example of two distinct computations, C1 and C2, in a distributed system that may access and modify common external data objects. Let us suppose that those two computations are entirely logically independent and that they may reach execution concurrently (perhaps on different nodes of the system). For example, in the VLSI design example mentioned earlier, they might represent different design operations being carried out upon the database representing a partially completed circuit. Structuring each of these computations as an atomic action would allow them to execute in a correct and serializable fashion. However, suppose that design diversity--for instance, in the form of n-version programming--was used to implement each of the computations in order to enhance software reliability.

Now, consider the atomic action representing the spatially redundant computation C1. Note that the external data objects used by each of the n-versions must be copied into a local data space for that version, before execution and modifications are made to the local copy. This is done because the versions must execute in isolation, and hence must not independently modify the actual external environment. At the conclusion of the execution of the n-versions, a vote is taken on the modifications to (copies of) external objects, and based upon the outcome, the actual modifications are made to external data objects.

There are a number of problematic aspects to this scheme, including:

1. It may not be possible to determine a priori which external objects will be accessed by a version of the computation. Even if it is possible, the number may be too large to effectively or efficiently copy into n different local data spaces. For example, duplicating the huge state space of a large artificial intelligence problem might not be practical. (Note that other schemes, such as the use of logging in each of the versions in lieu of copying external state space into local states, are possible, but these suffer from similar problems.)

2. Since the n versions may be executed concurrently on distinct processing nodes, and since external data objects may also be distributed among nodes, a run-time structure must be constructed to coordinate and synchronize the copying of external objects to the local data spaces, the final voting upon results, and the copying of results back to the external data space of the global atomic action for $C_1$. This structure is over and above that required for the support of atomic actions.

## III. Atomic and Molecular Activities

In this section we propose a new structuring mechanism, called a molecular activity, intended to deal with problems of the type illustrated in the above example. This mechanism provides a general way of structuring spatially redundant distributed computations that access external data objects. As such, it is applicable not only to n-version programming problems of the sort described above, but also to supporting other constructs for design diversity, such as the parallel execution of recovery blocks or conversations [3-5]. It is also appropriate for supporting the parallel execution of alternative strategies in order to speed up the solution of computationally difficult problems. The mechanism automatically manages the consistency of external objects accessed by spatially redundant computations without requiring that the objects be identified in advance. In addition, the mechanism provides the support for committing final results of the spatially redundant computation.

We define an atomic activity as a program-specified computation whose computational steps are executed such that the intermediate state of data objects manipulated by an atomic activity are not visible to any computation outside the atomic activity. An atomic activity that constitutes a molecular activity may be specified by a sequential or concurrent program, and hence, the computation of an atomic activity may be represented by either a single sequential process or a set of interacting processes.

A molecular activity is defined as a collection of one or more atomic activities that may execute concurrently such that, from the point of view of computations outside a molecular activity, it appears as an indivisible operation whose effect on the state of the data objects in the system reflects the actions of at most one of its component atomic activities. The indivisibility of molecular activities implies that the concurrent execution of any two molecular activities is equivalent to the sequential execution of the two molecular activities in some order. Thus, during the execution of a molecular activity, objects whose values are read by steps in the atomic activities constituting a molecular activity cannot be modified by any computation outside the molecular activity. Also, intermediate states of data objects that arise during the execution of a molecular activity will never be observed by computations outside the molecular activity. In fact, even for the computations within the molecular activity, modifications made to the data objects by one atomic activity will not be visible to other atomic activities constituting the molecular activity.

Let $S_M$ denote the set of data objects accessed by the computations in the molecular activity M. Let A = $A_1$, $A_2$,..., $A_n$ denote the set of atomic activities defining the molecular activity M. The initial state I of M is defined by the values of the data objects in $S_M$ before they were modified by any computation within M. The final state F of M is defined by the values of the elements in $S_M$ after the completion of the molecular activity M.

From the definition above, it is clear that the final state F of M is determined by the transformation on I performed by the computation of at most one of the atomic activities in A. Now, the criterion for determining which, if any, of the atomic in A is to be chosen to reflect the execution of the molecular activity M is application dependent. It is specified by a set of user-defined procedures encapsulated within a construct called a decision module, which is associated with each atomic activity of the molecular activity M.

The informal definition of a molecular activity given above provides only an abstract description in terms of the effect of its execution on the state of the system and ignores details of the internal structure of a molecular activity. A more formal model of atomic activities and molecular activities is presented in [6].

## IV. Molecular Activities--Implementation Issues

The implementation of a system supporting molecular activities must satisfy the following requirements.

R1: Let $M_I$ denote the initial state of a molecular activity M that is composed of atomic activities $A_1$, $A_2$, ..., $A_n$. Then, the concurrent execution of $A_1$, $A_2$, ..., $A_n$ should be such that each atomic activity $A_i$ in M is oblivious to the transformations on $M_I$ made by any other atomic activity of M. We call this the isolation requirement.

R2: After the completion of the molecular activity M, the final state $M_F$ of M should reflect the transformations on $M_I$ made by at most one of the atomic activities of M. We call this the global data consistency requirement.

R3: The transformation from the initial state, $M_I$, to the final state, $M_F$, of the molecular activity M appears as an indivisible and primitive step to all computations outside M. Requirement R3 can be refined as follows:

R3a: During the execution of the molecular activity M, the intermediate states of objects accessed by computations in M should not be visible to computations outside M.

R3b: If $A_1$, $A_2$, ..., $A_n$ represent atomic activities belonging to different molecular activities, the concurrent execution of $A_1$, $A_2$, ..., $A_n$ must be equivalent to the sequential execution of $A_1$, $A_2$, ..., $A_n$ taken in some permutation.

To satisfy requirements R1 and R3 we need a mechanism that controls access to data objects in the system. We call this mechanism the synchronization mechanism. To ensure requirement R2 we need another mechanism, which we call the committment mechanism. The implementation of these mechanisms in a distributed system is presented in [6].

## V. Concluding Remarks

In this work we have described a mechanism, called a molecular activity, for the support of spatially redundant distributed computation. This mechanism allows each redundant element to be constructed as a set of distributed communicating processes known as an atomic activity. Multiple molecular activities can execute simultaneously in a distributed system, and they can access and modify a common global data space without loss of consistency.

Molecular activities allow the domain of application of existing constructs for achieving software fault tolerance to be extended to include applications constructed as sets of communicating processes and involving access to global data. No methods have previously existed for allowing n-version programs to share global data, and no explicit work has addressed the viability of versions constructed as sets of communicating processes in a distributed environment. With respect to other proposed constructs for design diversity [4,5], molecular activities can also be used to support parallel execution of the interacting sessions of a conversion. Details are given in [6].

The potential use of spatial redundancy to allow simultaneous application of alternative algorithms or heuristics for solving a computationally difficult problem is largely unexplored. An extended example of this approach applied to a planning problem in artificial intelligence is given in [6]. The example demonstrates the viability of this technique. Spatial redundancy employed in this fashion exploits parallelism at a very high level. The molecular activity construct allows each activity to consist of a set of distributed processes. Hence, the use of spatial redundancy does not mitigate the exploitation of parallelism within individual versions of the algorithm or heuristic.

## References

[1] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," Computer, Vol. 17, No. 8, (August 1984), pp. 67-80.

[2] T. Anderson and P.A. Lee, Fault Tolerance, Principles and Practice, Englewood Cliffs, Prentice-Hall, 1981.

[3] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," in Lecture Notes in Computer Science, Vol. 16, Berlin: Springer, 1984, pp. 177-193.

[4] B. Randell, "System Structure for Software Fault-Tolerance," IEEE Trans. on Software Eng., Vol. SE-1 (June 1975), pp. 220-232.

[5] K.H. Kim, "Software Fault Tolerance," in Handbook of Software Engineering, C.R. Vick and C.V. Ramamoorthy (Eds.), New York: Van Nostrand Reinhold Company, 1984, pp. 437-455.

[6] R.S. Ramanujan, "Molecular Activities: A Structuring Mechanism for Distributed Computation," Ph.D. Thesis, Dept. of Electrical and Computer Engineering, University of Iowa, December 1985.

# POKER ON THE COSMIC CUBE: THE FIRST RETARGETABLE PARALLEL PROGRAMMING LANGUAGE AND ENVIRONMENT[†]

Lawrence Snyder
David Socha

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

**Abstract** – This paper describes a technique for retargetting Poker, the first complete parallel programming environment, to new parallel architectures. The specifics are illustrated by describing the retarget of Poker to CalTech's Cosmic Cube. Poker requires only three features from the target architecture: MIMD operation, message passing inter-process communication, and a sequential language (e.g. C) for the processor elements. In return Poker gives the new architecture a complete parallel programming environment which will compile Poker parallel programs, without modification, into efficient object code for the new architecture.

## Introduction

Software portability for sequential computers became an issue in the early sixties as higher-level languages began to supplant the machine specific assembly languages and as machine varieties proliferated; it was a much harder problem than originally supposed and it remains a serious problem today. By comparison, portability of a parallel program should be substantially more difficult because:

- architectural differences among parallel computers are much more fundamental than among sequential machines, and
- the characteristic that makes portability difficult – the dependence of programs on machine specific features – arises often in parallel computation in order to get good performance.

We say "should be more difficult" because to date there is little experience: There is little production parallel software and there are few truly parallel languages, and few parallel machines. But in spite of the potential problems, there is some reason for optimism.

The Poker[1] language has been retargetted from the Pringle Parallel Computer [2] to the CalTech Cosmic Cube [3]. Thus, programs written for the CHiP [4] family of computers can run on one of the cube family of architectures *without modification.* This is possible because

- the Poker language uses a reasonably universal program abstraction,
- Poker programs have a (unique) structure that is both visible and simple, and
- the Poker language and environment is structured to make retargetting simple.

There is no impediment to porting the Poker language to other parallel computers as this paper will explain.

The benefit of portable parallel software is obvious: Programs can be written without regard for the underlying architecture. However, portability only guarantees that programs will *function.* With Poker, we are making a stronger claim: Poker programs will run with an efficiency that is comparable to that of programs which were specifically written for that architecture. This leads to a key point about the retargetable Poker parallel programming environment:

> Poker requires a small set of system functions of the host architecture and can thus serve as the definition of the basic software support required of a new parallel computer.

Simply by creating the few basic interface systems that Poker requires, an architecture automatically inherits the available Poker programs and a complete software environment. This vastly reduces the software development efforts for new parallel machines.

## Overview

Before explaining the details of the retarget, some familiarity with the Poker system must be acquired. Towards this end we present first a high level view of the Poker system pointing out some of the key components and their interactions. Later, we discuss the relevant software and hardware pieces, devoting a section each to: the Poker programming environment, the Cosmic Cube, and the new cross-compiler. Finally, we connect all the pieces and discuss the effects of the new extended system.

Figure 1 shows the relationship between the Poker Parallel Programming Environment and the parallel computers which can be programmed with it. Poker is a sequential system that makes extensive use of a relational database to represent programs. It is written in C[5] and built on top of UNIX[(a)].

To see how Poker works, focus on two components of the system: the cross-compiler and the debugging environment. The cross-compiler accepts a Poker program as input and produces an object version suitable for execution on one of three machines: the Pringle, the Cosmic Cube, or a simulator/emulator of a parallel machine that runs on whatever sequential machine is hosting Poker itself. The compiled version is then down-loaded to the target parallel computer and executed. During execution, the run-time support of the machine sends tracing information back to Poker's debugging environment so that the programmer can view the execution of the program within the Poker environment.

Thus, one writes and runs programs from an environment that provides flexible interactive graphic support and a view of the program consistent with its definition. During the program debugging activity there is communication between the front-end processor and the back-end parallel machine to facilitate program tracing. When the program is debugged, no tracing is requested and Poker serves as the operating system for the back-end machine, running the program "flat out."

Thus, Poker is both a language and an environment: A sequential system from which to compile, execute, and debug programs on parallel computers. A "port" of the Poker language entails retargetting the cross-compiler and constructing the run-time software to support the communication between the Poker environment and the parallel computer. Only the Poker programs and run-time system get ported to the new machine; the Poker environment runs on a sequential computer.[(b)] This paper will detail the activities required in crossing the vertical line of Figure 1.

## The Poker Programming Environment

The Poker programming environment is built around a programming abstraction that is common to non-shared memory parallel algorithms, as described in the next section. However, Poker's interface to the abstraction is quite non-standard. The section on Concrete Poker Programs outlines the structure and semantics of Poker programs. As we shall see, this formulation of parallel algorithms lends itself to easy retargetting for new parallel architectures.

### Abstract Poker Programs.

A non-shared memory parallel algorithm is conceptualized as a finite graph. The graph's vertices are labeled with process names corre-

[†] Supported in part by National Science Foundation Grant DCR-8416878 and by Office of Naval Research Contract No. N00014-85-K-0328.

[(a)] UNIX is a trademark of AT&T Bell Laboratories

[(b)] Currently, the Poker environment runs on a number of computers, including Vaxes, Vax-stations, Sun workstations, IBM PC/RTs, and HP-9000s.

0190-3918/86/0000/0628 $01.00 © 1986 IEEE

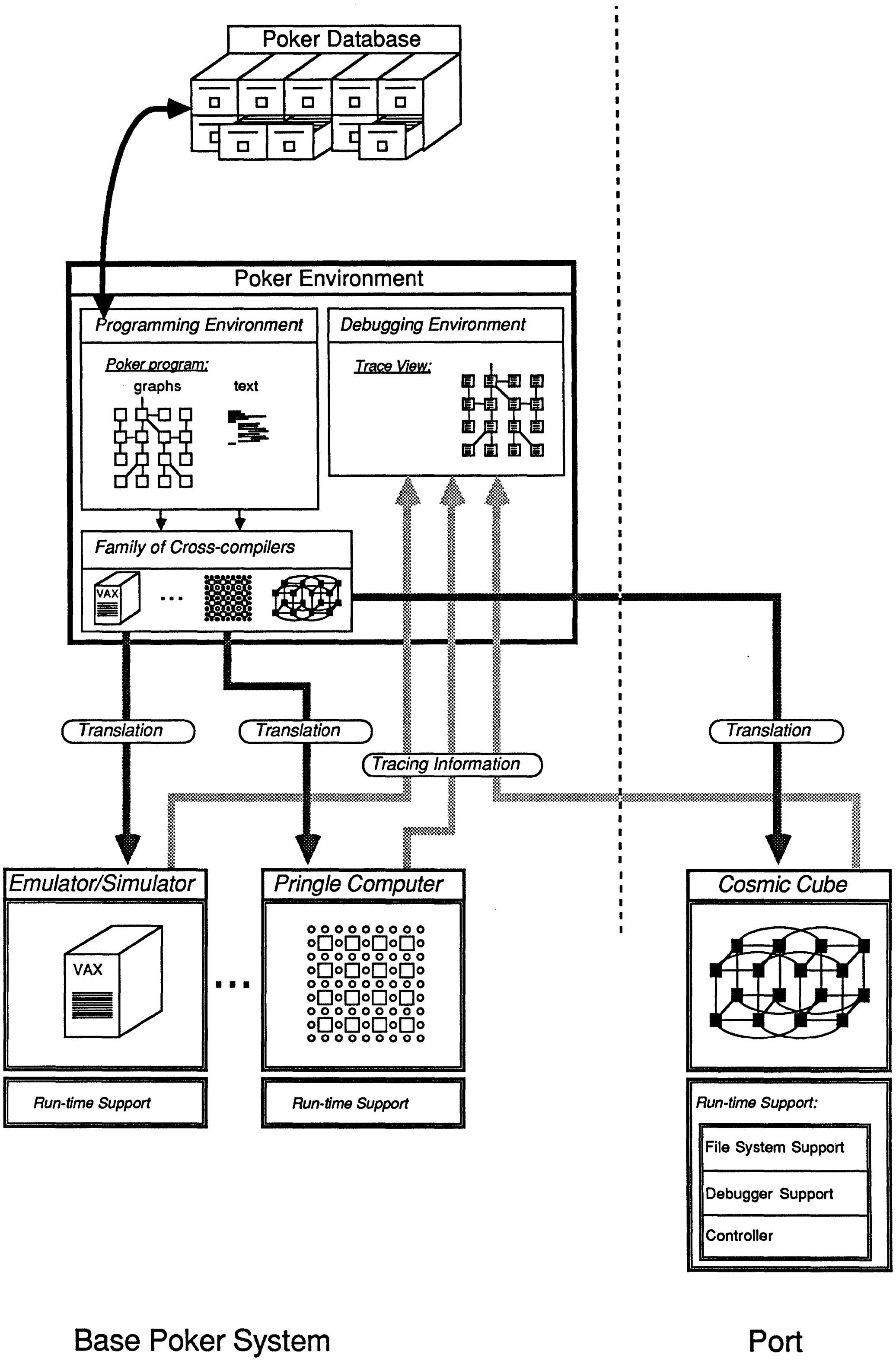
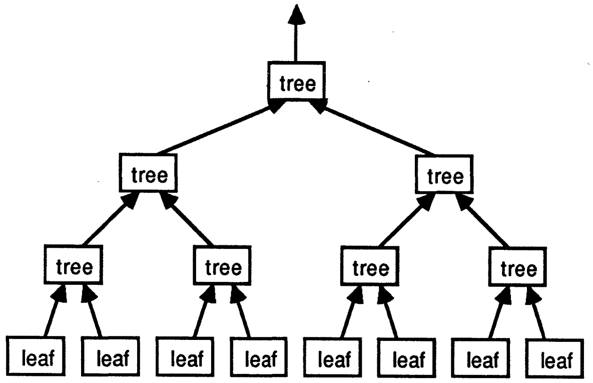


Figure 1: An Overview of the Poker System.



tree: parent <- max ( myValue, leftValue, rightValue)
leaf: parent <- myValue

Figure 2: The Maximum Algorithm.

sponding to sequential programs. The graph's edges are of two types: Edges between vertices are communication paths between processes, and edges "dangling" off of the graph are channels through which streams of data pass to or from the graph. (Technically, graphs cannot have dangling edges, of course, but it is convenient to abuse the definition.)

For example, Figure 2 shows the maximum algorithm. The graph is a binary tree. The vertices are labeled with one of: the name of the `leaf` process, which passes its local value to its parent, or the name of the `tree` process, which receives two values from its children, finds the maximum of these values and its local value, and passes the result to its parent. The dangling edge is a stream containing a single output value produced at the root.

A problem is not generally solved by a single algorithm of this form. Rather this form is usually one "phase" of a computation. The problem is partitioned into a series of phases, each possibly with a different graph structure. Inter-phase communication is permitted only among those that share the same "location" as described by a one-to-one mapping function. Values from previous phases may be inherited by the vertex executing in the corresponding vertex in the next phase.

## Concrete Poker Programs.

Unlike a C or Pascal program, a Poker *phase program* is not a monolithic piece of text. Rather, it is composed of five components that correspond closely to the abstraction just presented (Figure 3 shows the five components of the maximum algorithm encoded as a Poker phase program).

- *Communication Graph:* A finite graph with dangling edges. The boxes correspond to processes and thus will be the vertices of the graph. The circles, which are switches for the CHiP computer [4], can be ignored for the present discussion.

- *Process Definition:* A (usually small) set of processes written in a sequential language, in this case a slightly modified version of C[5]. These can be thought of as standard procedures with formal parameters called at the beginning of a phase.

- *Process Assignment:* A labeling of the vertices of the graph with the names of the processes and actual parameters, if any, to be executed at that processing site.

- *Port Name Assignment:* A labeling of the edges of the graph, but from the point of view of the vertex, *i.e.* each edge has two names, one for each end, or "port".

- *Stream Name Assignment:* A labeling of the dangling edges of the graph giving names to the input and output streams; these names will subsequently be bound to files.

The correspondence of the five Poker program components to the phase abstraction given above should be clear.

A Poker program is composed of a finite set of phase programs together with an execution scheduler that describes the sequence in which they are to be invoked. Those vertices of different phases which occupy the same location in the Switch Setting View may communicate across phases through the use of inter-phase variables.

Inter-phase variables live for the duration of the Poker program and exist separately from the variables declared local to a phase. The local process codes may access the values of inter-phase variables by the `import` and `export` statements:

`import` *local* `from` *inter-phase*
`export` *local* `to` *inter-phase*

`Import` copies the value of the inter-phase variable into the local variable. `Export` copies the value of the local expression into the inter-phase variable. This is the only way to pass information between phases. It provides a well-defined interface to inter-phase communication and lets process codes load from and store to inter-phase memory during the course of computation.

Two other features of Poker C are the trace list and inter-process communication. The optional trace list found in the declaration section of each routine specifies the variables whose values will be traced in the debugging environment, if a traced execution of the program is requested. Trace "variables" are not restricted to the conventional "variables" of Poker C. They may include variables, labels, and other items such as the current procedure and depth of recursion.

Process input and output, respectively, are given by expressions of the form:

*variable* <- *port*
*port* <- *variable*

The values transmitted are simple scalar values from the language, and arrays and structures not containing pointers. Messages are tagged with their type so that process I/O may be type checked using structural type equivalence.

## The Programming Environment.

The Poker system is the set of facilities that assist the programmer in writing and running Poker programs. Poker uses two displays: One is a bit-mapped display having the general form shown in Figure 4 and used for interactive graphical programming of the parallel aspects of the program. The second terminal is used with a standard editor to write the sequential C process text.

Poker stores programs as a database, displaying the program in one of several views [6]. Figure 4 shows the display for the Switch Setting View; the other views are analogous, with the appropriate Poker program constituent displayed in the lower half of the screen. The seven views are

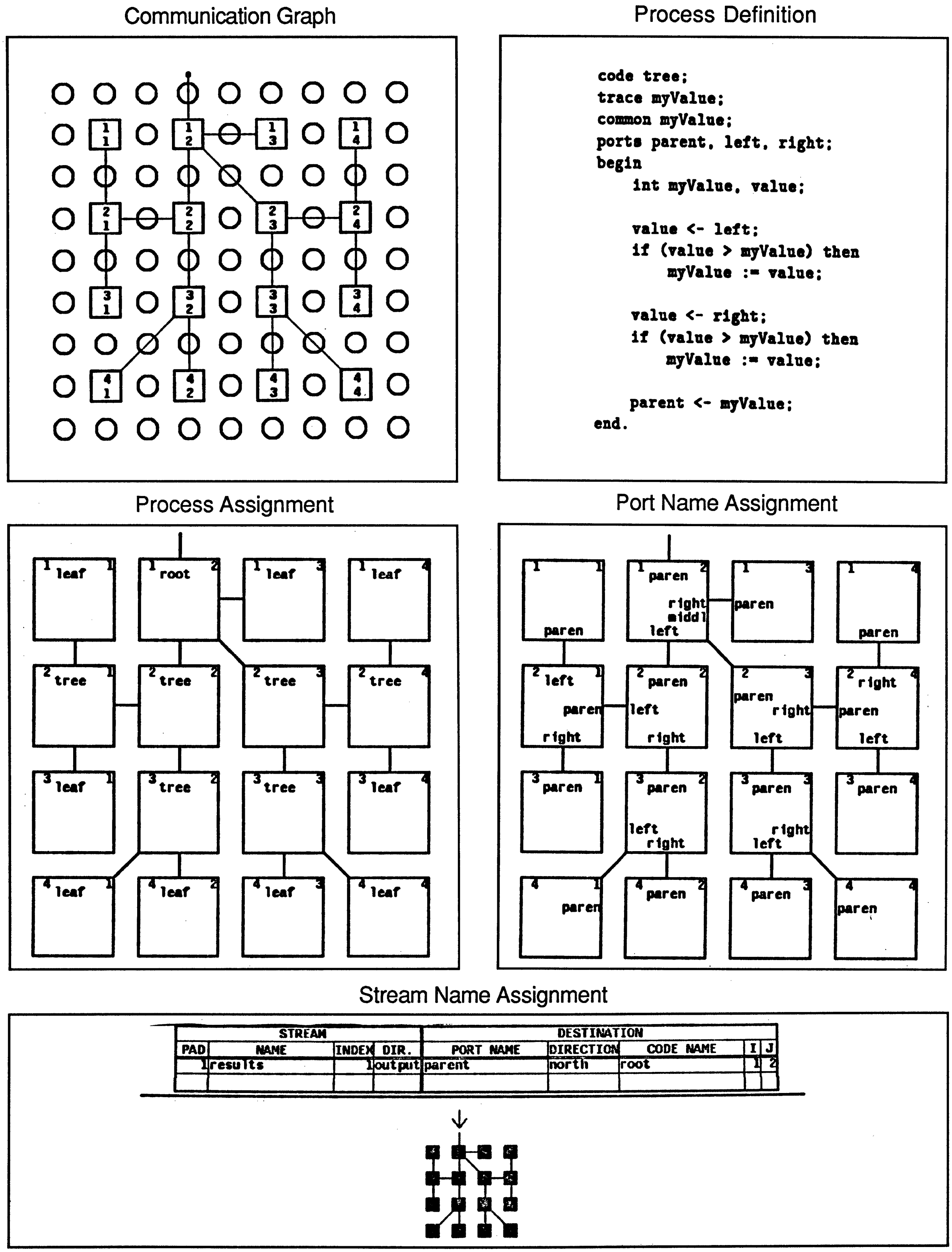
- *Switch Setting View:* Used to define the Communication Graph (Figure 3); the programmer uses a mouse or keypad to draw a picture of the graph by connecting the boxes with lines.

- *Code Names View:* Used to define the Process Assignment (Figure 3); the programmer moves from box to box entering the names of the process and actual parameters, if any.

- *Port Names View:* Used to define the Port Name Assignment (Figure 3); the programmer moves from box to box entering the names of the ports.

- *IO Names View:* Used to define the Stream Names Assignment (Figure 3); the programmer enters the names of the streams and their directions.
- *Command Request View:* Used to compile, assemble, link, load, etc. Poker programs. The bottom of the display shows the progress of the computation.

- *Trace View:* Used to display the execution of the Poker program; the programmer can start, stop, and single step the program while watching the successive changes of the traced variables.


Communication Graph

Process Definition

```
code tree;
trace myValue;
common myValue;
ports parent, left, right;
begin
    int myValue, value;

    value <- left;
    if (value > myValue) then
        myValue := value;

    value <- right;
    if (value > myValue) then
        myValue := value;

    parent <- myValue;
end.
```

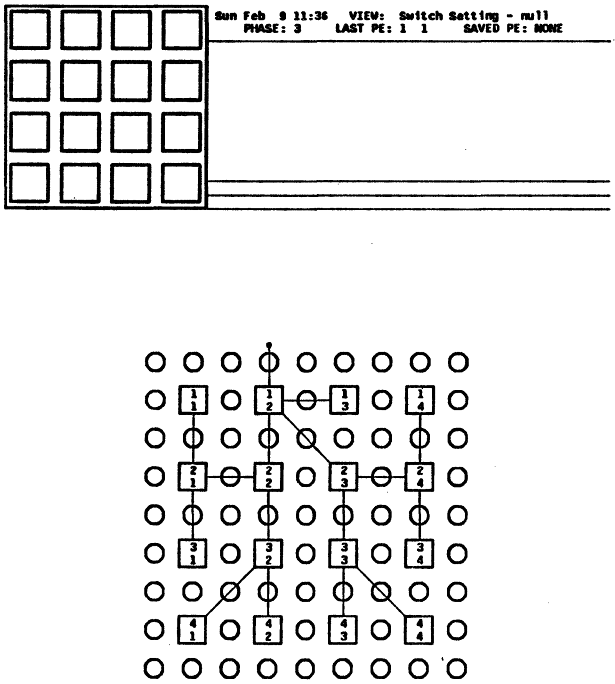
Process Assignment

Port Name Assignment

Stream Name Assignment

| STREAM | | | | DESTINATION | | | | |
|---|---|---|---|---|---|---|---|---|
| PAD | NAME | INDEX | DIR. | PORT NAME | DIRECTION | CODE NAME | I | J |
| 1 | results | 1 | output | parent | north | root | 1 | 2 |
| | | | | | | | | |

Figure 3: The five parts of a Poker program.



Sun Feb 9 11:36 VIEW: Switch Setting - null
PHASE: 3 LAST PE: 1 1 SAVED PE: NONE

Figure 4: The Poker Display showing the Switch Setting View.

- *CHiP Parameters View:* Used to describe the logical CHiP architecture being programmed. This includes parameters such as the number of processors in the processor grid (16 in Figure 3).

Notice that the display of most of the program constituents includes information defined in other program views; two examples (see Figure 3) are the graph shown in the Code Names View (Process Assignment) which is derived from the graph defined in the Switch Settings View (Communication Graph), and the material in the right half of the table of the IO Names View (Stream Name Assignment) which is gathered from the other views.

## The Cosmic Cube

The Cosmic Cube [3] provides flexible processing facilities that easily adapt to hosting the more structured Poker abstraction. Each Cosmic Cube program is also a graph: a set of processes, with non-shared memory, communicating through logical links mapped onto the underlying hardware [7]. However, instead of the static graphs of the phase program found in Poker, the Cosmic Cube programs consist of a single, perhaps dynamically changing, graph. Processes may create new communication links (edges), if the creator knows the address[c] of the other process, or destroy old links in order to build the best graph for the moment. This gives the Cosmic Cube programs more flexibility than we need.

This abstraction is implemented on a MIMD non-shared memory computer with a binary n-cube interconnection; the processors and their local memory, or "nodes", sit at the corners of the n-dimensional cube while the edges of the n-cube are formed from physical wires connecting the node processors. Each node may host zero or more processes, living in separate address spaces, communicating by message-passing over the physical wires connecting adjacent nodes. The operating system automatically forwards messages between non-adjacent nodes, preserving the message order between the sending and receiving processes.

A Cosmic Cube program starts as a single process on the host machine[d] connected to one corner of the Cosmic Cube. This host process "spawns" the processes in the initial program graph, placing them on the nodes of the cube and establishing the graph edges by forwarding process addresses to the processes on the cube. These processes may then spawn more processes of their own or create new communication links by sending known process addresses to other processes.

Typically, process codes are written in Cosmic Cube C, a version of the C programming language [5] extended with calls to routines in the Cosmic Cube's operating system, or one of the other extended sequential languages supported for the Cosmic Cube.

## The Poker to Cosmic Cube Cross-Compiler

Retargetting the Poker language and porting the run-time system to the Cosmic Cube changed nothing in the existing Poker environment; neither the Poker programs nor the Poker environment needed modification (process codes written in *XX* [2] are preprocessed into Poker C). Instead we only needed to (1) retarget the cross-compiler to translate Poker C process codes into a single Cosmic Cube C program, using the information in the database to determine the configuration of the resultant graph, and (2) extend the run-time software on the Cosmic Cube to include a few routines interfacing to, and controlling, the Cosmic Cube program. This Cosmic Cube C program from the translator is then treated as any other Cosmic Cube C program, compiling, loading, and executing it using the facilities provided for the Cosmic Cube. The difference is that the extra interface routines know about the Poker system, so that the user of Poker need not know which underlying architecture is executing the user program – no matter which target architecture is used, the creation and execution of the program will be the same. This retarget and run-time system port is the topic of the remainder of this paper.

### Converting Poker C to Cosmic Cube C.

The first task of the cross-compiler is to convert the Poker C process codes into a form acceptable for the processors of the target architecture. There are at least three ways to support a sequential language on the processors of a new architecture. One is to directly generate object code for the target machine's processor elements. A second method is to provide a kernel that runs on the processors of the new architecture and interprets the sequential language (or an intermediary language derived from it). This method could easily support a number of interpretive languages such as LISP [8] and Prolog [9]. The third approach is to translate the source process codes into a language already supported on the new architecture. This last approach works best when the languages are similar in structure, as is the case with Poker C and Cosmic Cube C.

Translating one sequential language into another, source-to-source translation, is well understood. In our case, the only major translation problems come from tracing the variables in the trace list and from reducing all of the Poker phase graphs into one more general and less structured Cosmic Cube graph.

Trace variables are handled by the first part of the cross-compiler which uses a Yacc [10] based C-to-C compiler to insert calls to a run-time trace routine after every instance of a traced variable. The C-to-C compiler does not look for aliases; instead it only inserts a trace command after each assignment whose left-hand-side is an instance of a literal in the trace list. If aliases are a concern, Poker will, at the user's discretion, insert a special trace call after each suspect assignment, exhaustively checking for any changes in the traced variables. Exhaustive traces are expensive at run time but, presumably, a great benefit for debugging.

The C-to-C compiler also replaces instances of the Poker's interprocess communication statements

*variable* <- *port*
*port* <- *expression*

with calls to the Cosmic Cube `send()` and `receive()` routines.

[c] On the Cosmic Cube a process' address consists of two numbers: the number of the physical node in which it resides, and its process number on that node.

[d] A typical host machine is a Sun Microsystems workstation.

The C-to-C compiler is more complex than a simple pre-processor such as UNIX's cpp. To see why, consider tracing a variable that is modified twice in a single expression. We want to trace both changes, but to capture both values of the traced variable we have to insert trace calls *into* the expression. We cannot simply append a trace call onto the expression since the value of the expression may be needed for an assignment or conditional test. Instead, we have to store the expression value in a temporary variable, trace the changed trace variables, and then append the temporary variable to return the expression value.

For instance, given the variables

```
float f;
int i, j;,
```

and a request to trace f and j, the C-to-C compiler converts the conditional expression

```
(((f += 4.3) < 10) ?
f++ + j++, i-- :
f--)
```

into

```
(((_tempint = ((f += 4.3) < 10)), _Trace(&f, FLOAT),
      _tempint) ?
f++ + j++, _Trace(&f, FLOAT), _Trace(&j, INT), i-- :
((_tempfloat = f--), _Trace(&f, FLOAT), _tempfloat))
```

where _Trace is a trace routine taking a pointer to a value and a constant telling it the type of the value and _tempint and _tempfloat are reserved variables declared in the enclosing routine.

In the first line, _tempint determines which of the next two expressions to execute: the comma expression before the colon, or the simple expression after the colon. We have to insert a trace of f immediately after the < expression, since if we waited to trace the value of f until after executing the entire conditional expression, we would miss the first value of f.

For the same reason we needed _tempint, the value of the expression was used outside of the expression, we also need to use _tempfloat to save the value of the conditional expression before tracing f's new value.

Clearly, this is not a simple textual substitution. Before we insert a temporary variable, we have to know the type of the expression. This requires keeping a symbol table, to hold the types of the variables, and using a parse tree to calculate the types of expressions. In other words, the C-to-C compiler is truly a compiler even though the languages it consumes and generates are nearly identical.

The end result of the C-to-C compiler is two code segments from each of the Poker C process codes. One segment contains information about the inter-phase variables imported to, and exported from, that process code. The other segment contains the routines defined for that process code. These code segments are used by the cross-compiler, as discussed in the next Section.

### Compiling the Poker Database.

The second part of the cross-compiler combines the pieces of the newly translated code with the rest of the information in the Poker program's database to create a single Cosmic Cube C program. The first problem here is to figure out how to collapse the phases of the Poker program into one graph, while maintaining efficient inter-phase communication$^{(e)}$. This is easily achieved by the technique suggested in Figure 5. When the phases are stacked as in Figure 5 the processes above one another are exactly those processes that share inter-phase variables. Combining them into one "aggregate" process keeps the Poker processes in a single process space. Inter-phase variables are globally declared for that process so that inter-phase communication requires no extra computation.

Building these aggregate processes is done as follows. The cross-compiler computes the inter-phase data space of each aggregate process from the inter-phase variables required by each of the Poker processes being placed into the aggregate process. The main() routine of each aggregate process is simply an infinite loop containing a call to the controlling host process asking for the number of the phase to execute, followed by a switch statement to call the main routine of the appropriate phase (see Figure 6). Since Poker C does not allow externally declared variables, the only potential source of name conflicts among different phases come from the routines, typedef's, and externally defined structures and unions. Prefixing these names with a unique phase ID eliminates any possible name conflicts. Linking the main() code with the routines from the phases and routines supporting the calls to receive(), send(), and the like, results in the Cosmic Cube C code for the aggregate process.

The last problem with the aggregate processes is mapping the edges of the different phases onto one graph. Since the Cosmic Cube does not have any restriction on the degree of the vertices (processes) in its program graph, we can project all of the phase graphs onto one graph. In actuality, each aggregate process gets a different logical interconnection map for each phase. The map holds the eight pairs of (process address, port number) that correspond to the terminal ends of the wires connected to the ports of the process for that phase. Since process addresses are known only after the processes are placed on the Cosmic Cube nodes, these tables are initialized at run time.

This packing results in extremely efficient inter-phase communication and phase change. One of the advantages of Poker programs is that the phase graphs are known at compile time so that the processors do not have to expend any run-time effort constructing or modifying the program graph or dynamically allocating more processes.

After spitting out the aggregate vertex codes, the cross-compiler runs them through the Cosmic Cube C compiler to get executable versions ready to be placed on the Cosmic Cube.

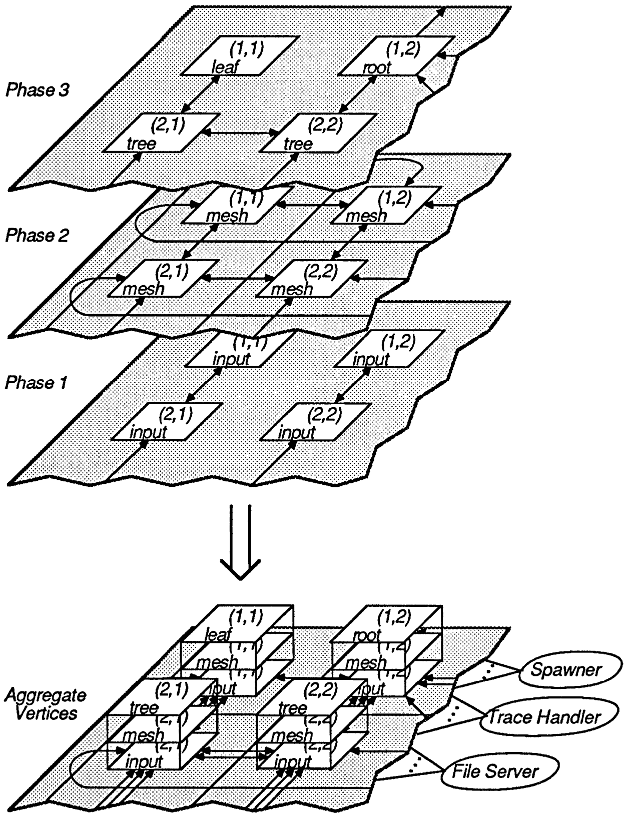


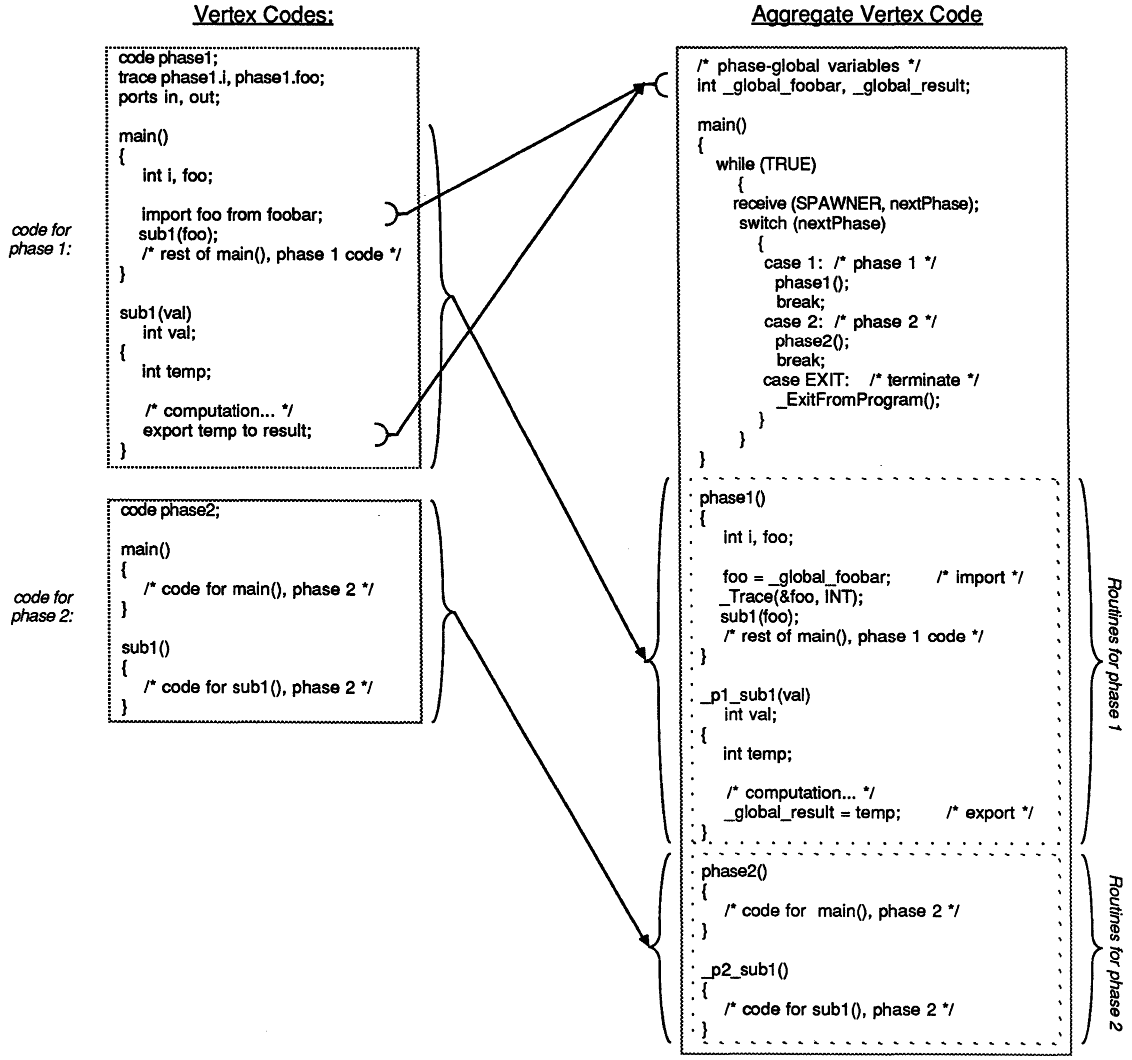
Figure 5: Collapsing the phases of a Poker program to create the aggregate processes.

[e] Keeping efficient inter-*process* communication is a much more complex issue, discussed in the Results.

Extending the Cosmic Cube's Run-time System.

The third task of the cross-compiler is to augment the operating system of the target computer with special processes implementing the features we need for the source language. In the case of Poker, we have three such special processes: the File Server providing file system support, the Spawner acting as the controller, and the Trace Handler providing debugger support. All three of these special processes live on the host computer attached to the Cosmic Cube.

- *Spawner:* The Spawner has two tasks:
  - *Initializing a Poker program onto the Cosmic Cube.* This includes:
    - *Mapping aggregate processes to Cosmic Cube nodes.* Currently, the aggregate processes are mapped naively onto Cosmic Cube nodes, with no attention to the interconnection between the processes. Better allocations are often possible, but the realization of one of these is a complex matter discussed in the Results.
    - *Placing the aggregate processes on the Cosmic Cube nodes.* The Spawner, takes the executable codes for the aggregate processes and places ("spawns") them on the Cosmic Cube nodes.
    - *Initializing the edge maps of the "aggregate" processes.* After the Spawner places all of the aggregate processes on the Cosmic Cube the Spawner sends each aggregate process a message containing the addresses of the processes on the other end of its edges, so that the aggregate processes can initialize their edge mapping tables. The Cosmic Cube operating system provides these addresses as a result of the Spawner placing the aggregate processes.
  - *Controlling execution order of the phases.* The Spawner is the Master Control Program that sends messages to all of the aggregate processes telling them which phase to execute. When each aggregate process finishes its phase code, it sends a "Done!" message to the Spawner. The Spawner waits to hear the "Done!" messages from all of the processes before sending the next "Execute phase *n*!" message to all of the aggregate processes.



```
code phase1;
trace phase1.i, phase1.foo;
ports in, out;

main()
{
    int i, foo;

    import foo from foobar;
    sub1(foo);
    /* rest of main(), phase 1 code */
}

sub1(val)
    int val;
{
    int temp;

    /* computation... */
    export temp to result;
}
```

```
code phase2;

main()
{
    /* code for main(), phase 2 */
}

sub1()
{
    /* code for sub1(), phase 2 */
}
```

```
/* phase-global variables */
int _global_foobar, _global_result;

main()
{
    while (TRUE)
      {
      receive (SPAWNER, nextPhase);
       switch (nextPhase)
         {
          case 1:  /* phase 1 */
            phase1();
            break;
          case 2:  /* phase 2 */
            phase2();
            break;
          case EXIT:    /* terminate */
            _ExitFromProgram();
         }
      }
}
```

```
phase1()
{
    int i, foo;

    foo = _global_foobar;        /* import */
    _Trace(&foo, INT);
    sub1(foo);
    /* rest of main(), phase 1 code */
}

_p1_sub1(val)
    int val;
{
    int temp;

    /* computation... */
    _global_result = temp;       /* export */
}
```

```
phase2()
{
    /* code for  main(), phase 2 */
}

_p2_sub1()
{
    /* code for sub1(), phase 2 */
}
```

Figure 6: Packaging the Poker C process codes into a Cosmic Cube C aggregate process.

- *File Server:* Processes access Poker's file system through dangling edges. The File Server keeps "input" edges filled with values from files and stores values from "output" edges into files. All messages on dangling edges route through this File Server.[f]
- *Trace Handler:* Trace variables are variables whose values are always known to the outside world where a programmer may view them in a special Trace View within Poker. The Trace Handler collects messages from the processes indicating new values for the traced values and sends them to Poker's debugging environment. These values come from `send()` statements that the C-to-C compiler inserts after each assignment to a traced variable.

The Cosmic Cube Poker Program.

In summary, the Cosmic Cube program corresponding to a Poker program consists of the aggregate processes on the Cosmic Cube nodes and three special processes running on the host computer: the Spawner, the File Server, and the Trace Handler. These processes implement the run-time support, while Poker provides the programming environment, traces program execution via the Trace Handler, and cross-compiles the Poker program to create the Spawner, the File Server, and the aggregate processes that run on the Cosmic Cube.

## Results

Machine dependencies. Most of the Poker system is machine *in*dependent. The machine dependent pieces fall into three parts:

- *The Cross-compiler.* The C-to-C compiler inserts calls to runtime routines for inter-process communication, inter-phase communication, tracing, and execution control. Changing these calls requires slight changes to the C-to-C compiler. If the target processors support C, the C-to-C compiler should not need any changes. However, if the target language is not C, the C-to-C compiler may need major reworking.
- *The Run-time system.* The routines supporting the interface between the Poker environment and operating system of the target processors are highly machine dependent. Large parts of these may have to be written from scratch.
- *The mapper from logical to physical interconnections.* Not only is this the least efficiently implemented feature of the system, as described below, but it also is extremely machine dependent.

Changing these three parts is sufficient to retarget the Poker language to a new architecture.

Mapping interconnections. When the programmer defines a communication graph in Poker and it is run on the Pringle (or, one day, the CHiP Computer), the graph is "directly implemented" in the sense that the figures produced in the Switch Setting View will be quite literally compiled into the object code of the machine. This is because the machines are configurable: The switches route messages according to the interconnection drawn in the Switch Set View. For any fixed-interconnection machine, such as the Cosmic Cube, the situation is different: The communication graph, which will be considered the logical communication structure, must be mapped onto the physical interconnection structure of the architecture. Of course, programmers *implicitly* perform this mapping when programming in other parallel languages.

As described in the section on extending the Cosmic Cube's run-time system, the mapping is done by the Spawner using an arbitrary allocation. If the programmer defined a logical cube graph, the system might not, as things now stand, allocate it so that logically adjacent vertices are physically adjacent. In general, the best allocation is not so obvious, yet the system should try to minimize the distance between communicating processes. The problem of automatically finding an optimal allocation instead of our *ad hoc* allocation remains unsolved. Moreover achieving optimality is complicated by multiple phases with the one-to-one correspondence of vertices between phases; a good mapping for one phase may be quite poor for another phase of the same algorithm. We are hopeful that the work of Berman and colleagues [11] will lead to an automated solution, though the retargetting in no way depends on Berman's software; it only improves the quality of the end result. Presently, we advocate a programmer assisted mapping, but we have not yet provided the software for it. No matter what enhancement is chosen, there is an entry point in Poker for the appropriate software.

## Conclusion

We have shown that it is easy to port Poker to a new parallel architecture, the Cosmic Cube, in such a way as to produce object code as efficient as any written for that architecture. Furthermore, Poker can easily be ported to *any* parallel computer simply by providing three basic features for the new architecture:

- MIMD operation,
- message passing facilities, and
- a compiler for the process codes.

These are very modest requirements considering that a small amount of work would then provide the new architecture with a complete, easy to use programming, debugging, and cross-compiling environment, as well as all of the existing Poker parallel programs. It is for this reason that we claim that Poker can be the definition of the basic software support required by a new parallel computer.

## References

[1] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[2] Alejandro Kapauan, Ko-Yang Wang, Dennis Gannon, Janice Cuny, and Lawrence Snyder. The Pringle: an experimental system for parallel algorithm and software testing. *Proceedings of the International Conference on Parallel Processing, IEEE*, 1–6, 1984.

[3] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[4] Lawrence Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[5] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language.* Academic Press, New York, 1978.

[6] Lawrence Snyder. *Poker (4.0) Programmer's Reference Guide.* Technical Report 86–05–04, Computer Science Department, University of Washington, May 1986.

[7] Wen-King Su, Reese Faucette, and Chuck Seitz. *C Programmer's Guide to the Cosmic Cube.* Technical Report 5203:TR:85, Computer Science Department, California Institute of Technology, September 1985.

[8] P. H. Winston and B. K. P. Horn. *Lisp.* Addison-Wesley, 1984 (second edition).

[9] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[10] Stephen C. Johnson. *Yacc – Yet Another Compiler Compiler.* Technical Report Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[11] Francine Berman, Michael Goodrich, Charles Koelbel, III W. J. Robison, and Karen Showell. Prep-P: a mapping preprocessor for CHiP architectures. *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[f] This is, of course, a potential bottleneck. Multiple File Servers could be used, but the architecture of the Cosmic Cube still requires all messages between the Cosmic Cube and the host machine, and thus the file system, to pass through the single physical edge connecting the host computer to the Cosmic Cube. Without additional hardware or a different basic IO design, Cosmic Cube programs risk becoming IO bound even with "reasonable" IO overhead.

# Performance of the Direct Binary *n*-Cube Network for Multiprocessors

Seth Abraham
Department of Computer Science
University Of Illinois
Urbana, Illinois 61801

Krishnan Padmanabhan
Digital Architectures Research Department
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

## Abstract

*A mathematical model for predicting the performance of the direct binary n-cube interconnection scheme in a packet switched environment is presented. These predictions are checked against simulations of a comparable system. The results for the network are compared to known results for indirect interconnection schemes like the crossbar and indirect binary n-cube networks. Several modes of operation of the network are analyzed and, it is shown that the direct cube performs better than its indirect counterpart for reasonable sizes of switches in the latter.*

## 1. Introduction

In the current climate of increased focus on large concurrent computer systems, it is recognized that one critical segment of such systems is the communication network used to connect the processing elements together. Various network schemes have been proposed for this purpose, including crossbars, indirect binary *n*-cube networks (also referred to as omega networks [Lawr75] [Peas77]), and direct binary *n*-cube networks [Seit85] (also referred to as the hypercube connection scheme). These interconnection schemes can be classified as *direct* or *indirect*. A direct interconnection scheme is one in which nodes (processing elements) are connected together by point-to-point links. In an indirect interconnection scheme on the other hand, the network is a separate entity with inputs and outputs; processors and memories are connected to the inputs and outputs respectively (most commonly in a shared memory architecture), or processors are connected to both inputs and outputs of the system (message passing architecture). Examples of the direct interconnection scheme are the ring, mesh, and binary *n*-cube architectures. Examples of the indirect interconnection scheme, other than the crossbar, are various multistage interconnection networks of the omega type, listed above.

Indirect binary *n*-cube type networks have been analyzed extensively in the literature ([DiJu81], [Pate81], [KrSn83]) for various aspects of their performance under different operating conditions. In this paper we analyze the performance of an important direct connection scheme, the direct binary *n*-cube network, in a packet switched mode of operation. We develop several mathematical models for the network, provide simulation results, and compare its performance with that of the indirect binary *n*-cube network.

## 2. Principles of Operation

The *binary d-cube* [Peas77] consists of $N = 2^d$ nodes connected together using dedicated edges. If the nodes are numbered 0 to $2^d-1$, then the *d*-cube connection is defined by the statement that two nodes whose binary representations differ in exactly one bit position are connected together. The graph of such a svstem has degree *d*. See [Peas77] and [Seit85] for detailed discussions of the binary *d*-cube interconnection scheme.

Each node in the *d*-cube consists of a processing element (PE) and a $(d+1)\times(d+1)$ crossbar switch as shown in Figure 1a. An input-output line pair constitutes an edge in the binary *d*-cube and the switch is connected to *d* of its neighbors using *d* such pairs. (See Figure 1b for the implementation of a binary 2-cube.) When a switch receives a message, the message's destination address is compared to the current node address. If there is a match, then the switch will attempt to route the message to the node's PE. Otherwise, the message will be routed to one of the node's neighbors, chosen at random from the bit positions that differ between the message and node addresses. When the switch attempts to route multiple packets to a single output line at the same time, either the multiple messages must be buffered at the output or all but one will be lost. We will analyze the performance of the network with and without buffers at the switch outputs.

Two versions of the switch will be considered for analysis: the *single-accepting PE scheme,* (Figure 1a) where one message can be accepted by the PE each cycle, and the *multiple-accepting PE scheme,* where multiple (up to *d*) messages can be accepted by the PE in a single cycle. The latter is more realistic in a scheme where the routing is done in software [SEIT 85] and the PE takes on the switching functions.

For simplicity of analysis, all nodes will be assumed to be identical. Furthermore, in Sections 3 and 4 each node will be assumed to generate requests for other nodes in an egalitarian fashion. As the traffic is being generated by identical nodes, for uniform destinations, it is reasonable to conclude that at each node the traffic received from any neighbor is identical to that received from any other neighbor. (In Section 5, we generalize the results to arbitrary distributions of message destinations.)

Let us define $m_g$ to be the message generation rate at each PE. (It is also equal to the probability of receiving a message from the PE in each cycle.) The rate at which messages arrive from a particular neighboring node is defined to be $m$. ($m$ is also the probability of receiving a message from a particular neighbor in a cycle.) Also, let the rate of messages going to the PE be defined as $m_a$. (If the single-accepting PE model is used, then $m_a$ is also the probability of a message being accepted by the PE in a cycle.) Furthermore, let $P_t$ (the probability of termination) be the probability that a message received from a neighboring node is for the PE.

## 3. Analysis of Unbuffered Networks

The performance measures we are interested in are the probability of acceptance of a generated message, and the bandwidth of the network as a whole. When multiple messages for the same output line are received at a switch in an unbuffered network, one randomly selected message "wins", and the rest of the messages are blocked. All blocked messages are lost from the system.

Let $P_A$ be the probability that a generated message is ultimately accepted. The total number of messages generated in the cube in one cycle is $Nm_g$. The total number of messages accepted in the cube in one cycle is $Nm_a$. Thus $P_A = Nm_a/Nm_g = m_a/m_g$. Network bandwidth is simply $Nm_a$. $m_a$ is computed in the following two sections.

### 3.1. Calculating the termination probability, $P_t$

We define the *distance* between two addresses on a cube to be the number of cycles needed to travel from one node to the other. This is exactly the number of differing bit positions in the two addresses. Hence the distance between any two node addresses is an integer between 0 and *d*. Given a node address, the number of node addresses that differ by exactly *k* bits is $\binom{d}{k}$. Let the distance between the message destination and the current node be defined as the *hopcount* of the message.

Starting with an empty network, on cycle 1 each PE can generate a message that will have hopcount *i* with probability $\binom{d}{i}/(N-1)$. At all subsequent cycles, this will also be the distribution of messages coming from each PE. $1/d$ of these messages will be sent to each of the neighboring nodes where it will be combined with the messages from $d-1$ other nodes. Let us define $P_a$, the probability that a (non-terminating) message

0190-3918/86/0000/0636 $01.00 © 1986 IEEE

received from a node is successfully passed on to another node, to be

$$P_a = \frac{\text{Messages going out to the nodes}}{\text{Messages trying to go out to the nodes}} = \frac{dm}{dm(1 - P_t)+m_g} .$$

(Recall that $m$ is the rate at which messages arrive at an input to the switch.) At the end of cycle 2 the distribution of messages sent to the neighboring nodes will be as follows:

$$\begin{array}{lccccc} \text{Hops:} & 0 & \dots & i-1 & \dots & d-1 \\ \text{Prob:} & \dfrac{P_a\binom{d}{1}+P_a^{\,2}\binom{d}{2}}{A} & \dots & \dfrac{P_a\binom{d}{i}+P_a^{\,2}\binom{d}{i+1}}{A} & \dots & \dfrac{P_a\binom{d}{d}}{A} \end{array}$$

where $A = P_a^{\,2}(N-1-d)+P_a(N-1)$. The first term in each probability is contributed by the PE on cycle 2, and the second term is contributed by the $d$ neighboring nodes. In general, at the end of cycle $i$, messages from node distance $i-1$ away $(0<i-1<d)$, can have hopcounts from 0 to $d-i$. (This is because a message can travel no further than distance $d$.) After $d$ cycles, the distribution reaches steady state. This distribution is given by

$$\Pr[\text{hopcount} = i-1] = \{\sum_{j=i}^{d} \binom{d}{j} P_a^{\,j-i+1}\} / \{\sum_{k=1}^{d}\sum_{j=k}^{d} \binom{d}{j} P_a^{\,j-k+1}\}$$

Thus the probability of receiving a message with hopcount = 0 (i.e., the termination probability) can be derived to be

$$P_t = \{(P_a-1)((1 + P_a)^d - 1)\} / \{P_a((1 + P_a)^d - N)\} .$$

### 3.2. Calculating message rates $m$ and $m_a$

Given a message from some other node, it terminates at this node with probability $P_t$. Otherwise, it is passed to one of $d-1$ other nodes as it is never passed back to the node it came from. Therefore, with probability $(1-P_t)/(d-1)$ a message is routed to one particular output line. (Actually, the message is routed to one of $k$ outputs, where $k$ is the number of differing bit positions between node and message addresses. However, this is a reasonable assumption, as comparisons with simulations will show.) The probability of getting a message from the PE is simply $m_g$. This message is routed to one particular output line with probability $1/d$.

Considering one particular output, the probability that no message arrives at this output is $(1-m_g/d)(1-m(1-P_t)/(d-1))^{d-1}$. Since the output line of this node is the input line for some other identical node, the rate at which messages leave the node must be equal to the rate messages arrive at the node. Thus,

$$m = 1 - (1 - m_g/d)(1 - m(1-P_t)/(d-1))^{d-1} .$$

Using the single-accepting PE model and employing a similar argument,

$$m_a = 1 - (1-mP_t)^d .$$

When the multiple-accepting PE model is used we obtain:

$$m_a = dmP_t .$$

### 3.3. Unbuffered network results and discussion

These results, along with corresponding simulations are presented in Figures 2 and 3. Figure 2 plots the probability of acceptance of a message in a direct binary cube as a function of the message generation rate for various network sizes. (A multiple-accepting PE model is assumed.) The model and the simulations agree to within 5% in all cases. The performance of the direct cube compared with that of the crossbar and indirect binary cube networks is presented in Figure 3. (Fairly standard models are considered for the crossbar and indirect binary cube networks, as for instance [KrSn83] and [Pate81].) Since the crossbar and indirect cube networks are by definition single accepting networks, a single-accepting PE model is assumed for the direct network figures in these graphs. Indirect networks can be constructed out of switches of different sizes and the choice of switch size affects both cost and performance. Figure 3 considers indirect binary $n$-cube networks constructed using switches of size 2, 4, and 8. We notice that it takes a switch size of about 8 in the indirect network to equal the performance of the direct network.

Various measures of cost for these two types of networks are shown in the table below (where the symbol "k" means multiplied by 1024).

| | Direct binary *n*-cube | | | Indirect binary *n*-cube | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Network Size | 256 | 1024 | 4096 | 256 | | 1024 | | 4096 | | |
| Switch Size | 9 | 11 | 13 | 2 | 4 | 2 | 4 | 2 | 4 | 8 |
| Switches | 256 | 1k | 4k | 1k | 256 | 5k | 1280 | 24k | 6k | 2k |
| Lines | 2304 | 11k | 52k | 2304 | 1280 | 11k | 6k | 52k | 28k | 20k |
| Crosspoints | 20736 | 121k | 676k | 4k | 4k | 20k | 20k | 96k | 96k | 128k |

The ability of the direct binary cube to outperform its indirect counterpart comes at the expense of increased hardware (using any of the cost measures), as the table illustrates. However, the hardware is not used in the most effective manner, as a comparison with the indirect cube using 8x8 switches shows. (The two networks have similar performance.) Thus it would seem that given a certain amount of hardware (essentially a number of switches and PEs), it would be preferable to connect them in the form of an indirect binary cube. It should be pointed out that this observation applies to the equiprobable reference model where a PE wishes to talk to any of the others with equal probability. To our knowledge, this is the mode in which existing implementations of the system software for direct cube architectures operate (for example, the Intel Hypercube). A more complex allocation of tasks to PEs, utilizing knowledge about locality of communication, would improve the performance of the direct binary cube.

## 4. Analysis of Buffered Networks

In a buffered network, multiple messages heading for the same output in a cycle are saved up in a queue at the output side of the switch. The line going to the PE needs a buffer only in the single-accepting PE model; the multiple-accepting PE model assumes some mechanism of giving up to $d$ messages to the PE at once. In the first part of this analysis, the queue lengths are assumed to be infinite; finite length queues are analyzed in Section 4.5.

When infinite buffers are assumed, no messages are lost. ($P_A = 1$, and bandwidth $= Nm_g$.) A good measure of network performance in this case is $T$, the mean message delay time. This time is defined to be the average time a message spends in the system before reaching the destination PE.

### 4.1. Calculating the termination probability, $P_t$, and message rate, $m$

By following an analysis similar to that for unbuffered networks (setting $P_a$ to 1) we obtain $P_t = (2N - 2)/(dN)$.

To derive expressions for $m$ and $m_a$, we observe that with infinite length queues, no messages can be lost. Hence, $m_a = m_g$. If we allow multiple messages to be accepted by the PE, then $m_a = dmP_t$. Therefore,

$$m = m_g/(dP_t) = Nm_g/(2N-2).$$

If the PE accepts only single messages in each cycle, then the above expression for $m$ is still valid if we consider $dmP_t$ to be the rate at which messages are sent to the PE's buffer.

### 4.2. Multiple-accepting model

In order to determine $T$ for the multiple-accepting model, we first must determine the length of the buffers at the switch outputs of each node. Consider a single output queue and associated output line. We say this system is in state $i$ (with probability $b_i$) when there are $i$ messages in the system. Examining first only the messages received from outside the node, the probability of receiving $i$ messages at a particular output line of the queue is given by

$$q(i) = \binom{d-1}{i}(1 - m(1-P_t)/(d-1))^{d-1-i}(m(1-P_t)/(d-1))^i$$

for $0 \le i < d$. (We assume that a message does not go back to the node it came from.) Considering generated messages also, the probability of getting $i$ messages at an output line is given by:

$$a_i = \begin{cases} (1 - m_g/d)q(0) & i=0 \\ (1 - m_g/d)q(i)+(m_g/d)q(i-1) & 0<i<d \\ (m_g/d)q(d-1) & i=d \\ 0 & i>d \end{cases}$$

Given these arrival rates at an output queue, the state distribution for the queue are

$$b_i = \sum_{j=0}^{i} a_j b_{i-j+1} + a_i b_0 \ .$$

Then the mean number of messages in the system can be shown to be

$$\bar{b} = m + \{m^2(d(1 - P_t^2) - 2(1 - P_t))\} / \{2(d - 1)(1 - m)\}.$$

Using Little's identity, the mean time a message spends at an intermediate node is given by $\bar{b}/m$. As an average message goes through $\frac{1}{N-1}\sum_{j=1}^{d} j \binom{d}{j} = \frac{1}{P_t}$ nodes, the delay to reach the destination node is $\bar{b}/(mP_t)$. Including an additional cycle for the message to be transferred to the PE from the switch, the total delay is obtained as

$$T = 1 + \bar{b}/(mP_t).$$

**4.3. Single-accepting model**

Now we consider the case where only one message can be accepted at the PE at a time. We need to determine the length of the PE's buffer and this can be done in the same manner as before. For the PE's queue, the arrival rates are given by

$$a_i' = \binom{d}{i}(1 - mP_t)^{d-i}(mP_t)^i$$

for $0 \le i \le d$, and $a_i = 0$ otherwise. The probability of being in state $i$ is $b_i'$. Using this and following the same procedure as before the mean message delay can be derived as:

$$T = \bar{b}/(mP_t) + m_g(d-1)/(2d(1-m_g)) + 1.$$

**4.4. Buffered network results and discussion**

Figure 4 presents the results in the infinite buffered model and compares them with the same model for indirect binary cube networks. Figure 4a verifies the accuracy of the analytical model vis a vis simulations. We notice that with the multiple-accepting model, about a 30% increase in delay results from a fully saturated network. With a single-accepting PE model however, (Figure 4b), the receiving queue at each PE does blow up in size with increasing generation rates just as in the case of the indirect network. However, at high request rates, the difference between the direct and indirect networks is substantial. As in the case of the unbuffered networks, we notice that a switch size of at least 8 is needed before the indirect networks can exceed the performance of the direct network. The cost-performance discussion in Section 3.3 for unbuffered networks holds for buffered networks also.

**4.5. Finite buffers**

We now consider buffered systems where the queue lengths at the outputs of switches are finite. We define $Q$ to be the length of the queue. Recalling our definition of state from section 4.2, such a queue and associated output line has $Q + 1$ states. The expressions for $a_i$ and $b_i$ in section 4.2 still apply ($b_i = 0$ for $i > Q$); however, the rest of the analysis in that section is particular to the infinite buffer case. For the finite buffer case, we still must derive an expression for $m$ (and $P_t$). Since the associated output line emits a message whenever the system is not in state 0, $m = 1 - b_0$. $b_0$ may be calculated by solving the system of linear equations described by the state transitions. (When $Q = 0$, $m = 1 - a_0$ which corresponds to section 3.2.) Given this value for $m$, $P_t$ and $P_a$ may be calculated using the expressions of section 3.1.

For the multiple-accepting PE model, $m_a = dmP_t$ just as in section 3.2. When using the single-accepting PE model, $m_a = 1 - b_0'$, where $\{b_i'\}$ is the state distribution for the PE's queue. (As before, when $Q = 0$, $m_a = 1 - a_0'$ which corresponds to section 3.2.)

Figure 5 displays the results for the finite buffered network for varying values of $Q$. Note that fairly small buffer sizes yield more than 90% of infinite buffer performance. The performance difference between the single-accepting and multiple-accepting models is due to a bottleneck at the node to PE connection in the single-accepting model. This bottleneck is most constraining when the message generation rate approaches 1 and message delays increase (refer to Figure 4b). The behavior of the finite buffered direct cube is similar to that of the indirect binary cube network, as presented in [DiJu81]: small buffers capture the effective performance of the infinite buffers, in all but the saturated mode of network operation.

**4.6. Saturation value of message generation rate, $m_g$**

In Section 3.3 we observed that while a direct binary $n$-cube network had more hardware then an indirect network, there was not a corresponding performance gain. If we let $m_g = 1$, we obtain $m = N/(2N - 2)$ (Section 4.1). For maximum utilization of the network links, $m$ should approach 1, and this happens when $m_g$ approaches $2(N - 1)/N$ [AbPa86]. Thus, a multiple-accepting PE requires a *multiple-generating* PE to allow full utilization of all the network links.

## 5. General Distributions

It is possible to generalize our expressions in the previous sections to arbitrary reference patterns [AbPa86]. Let $h_i$ be the probablility that a message generated at a node will have hop-count $i$. Figure 6 shows the message delay for three distributions of hopcount. $h_i = 1/d$ corresponds to a uniform distribution of hopcounts, while the harmonic reference pattern is given by $h_i = 1/\alpha i$, where $\alpha = \sum_{i=1}^{d} 1/i$. Clearly, exploiting the locality of reference in a program results in substantial performance increases.

## 6. Summary and Conclusions

We have presented in this paper an analytical model and simulation results for the performance of the direct binary $n$-cube interconnection scheme for multiprocessors. The results show that with the equiprobable reference model for interprocessor communication, the direct cube performs better than the indirect cube networks constructed using switches of size less than 8 for all modes of operation that we have considered. The indirect networks require fewer switches, especially as the switch sizes begin to increase. However, it is possible in the direct case, for each processing element to take on some or all of the switching function at each node. In the latter case, the direct binary $n$-cube performs especially well in a mode called the multiple-accepting PE mode of operation. Furthermore, this multiple-accepting PE mode, coupled with a multiple message generating PE will fully utilize all the network bandwidth; this bandwidth is greater than that of the indirect cube networks, due to the presence of more links in the direct cube.

Results regarding buffering of packets at nodes generally are similar to known results in the indirect network case: not buffering packets leads to a significant loss of packets in the network, and a small buffer size (1 or 2) dramatically reduces loss of packets. When processor allocation is done intelligently to exploit locality of communication, our analysis shows that significantly better performance can be realized. This will have to be done for the performance/cost ratio to be better in the direct cubes than in their indirect counterparts.

## References

[AbPa86]S. Abraham and K. Padmanabhan, "Performance of the Direct Binary $n$-Cube Network for Multiprocessors," submitted for publication.

[DiJu81] D. M. Dias and J. R. Jump, "Packet Switching Interconnection Networks for Modular Systems," *Computer*, Vol. 14, No 12, pp. 43-53, Dec 1981.

[KrSn83] C. P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Transactions on Computers*, Vol. C-32, No. 12, pp. 1091-1098, Dec 1983.

[Lawr75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, Vol. C-24, pp. 1145-1155, Dec 1975.

[Pate81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers*, Vol. C-30, No. 10, pp. 771-780, Oct 1981.

[Peas77] M. C. Pease, III, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Transactions on Computers*, Vol. C-26, No. 5, pp. 458-473, May 1977.

[Seit85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, pp. 22-33, Jan 1985.

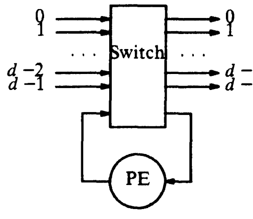


Figure 1a:

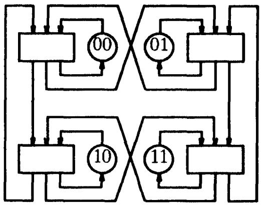


Figure 1b:

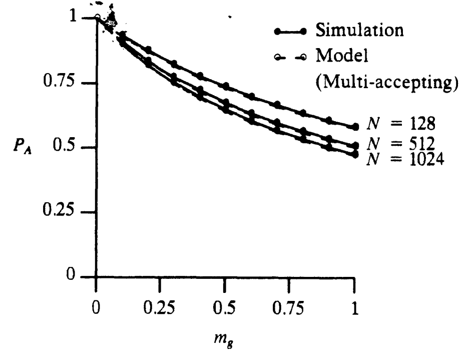


Figure 2:

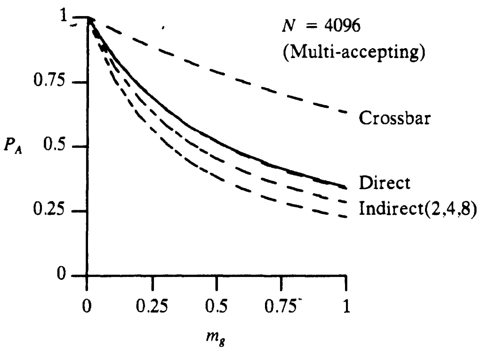


Figure 3a:

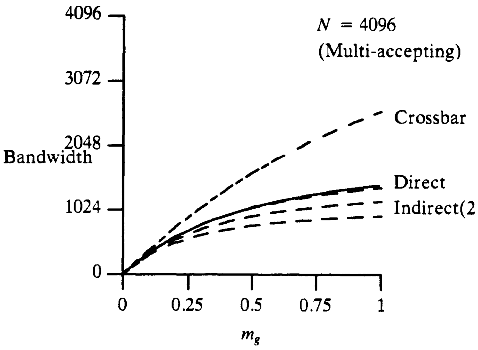


Figure 3b:

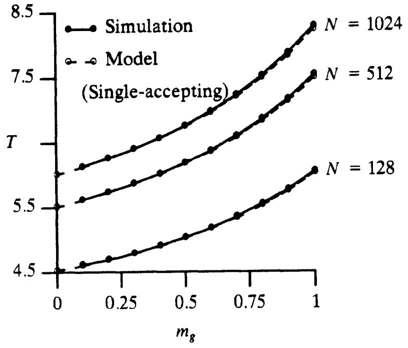


Figure 4a:

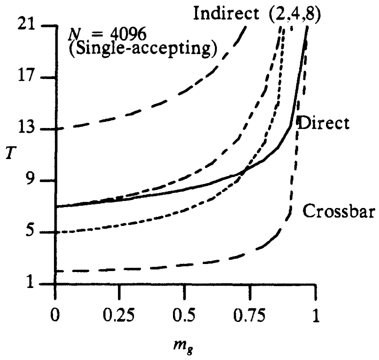


Figure 4b:

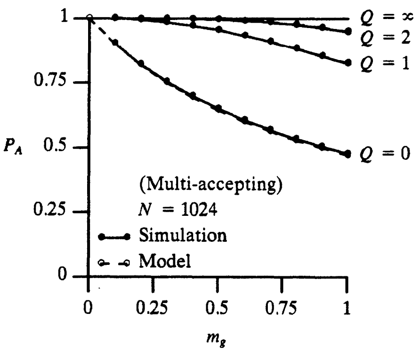


Figure 5a:

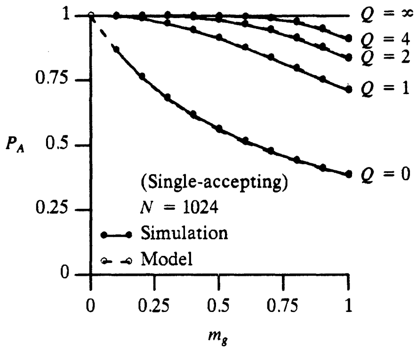


Figure 5b:

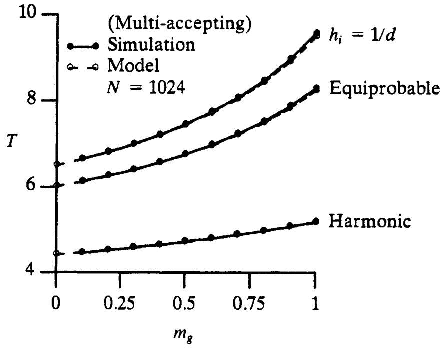


Figure 6:

# Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes

Ching-Tien Ho and S. Lennart Johnsson

Department of Computer Science
Yale University
New Haven, CT 06520

## Abstract

High communication bandwidth in standard technologies is more expensive to realize than a high rate of arithmetic or logic operations. The effective utilization of communication resources is crucial for good overall performance in highly concurrent systems. In this paper we address two different communication problems in Boolean $n$-cube configured multiprocessors: 1) broadcasting, i.e., distribution of common data from a single source to all other nodes, and 2) sending personalized data from a single source to all other nodes. The well known spanning tree algorithm obtained by bit-wise complementation of leading zeroes (referred to as the SBT algorithm for *Spanning Binomial Tree*) is compared with an algorithm using *multiple spanning binomial trees* (MSBT). The MSBT algorithm offers a potential speed-up over the SBT algorithm by a factor of $\log_2 N$. We also present a *balanced spanning tree* algorithm (BST) that offers a lower complexity than the SBT algorithm for Case 2. The potential improvement is by a factor of $\frac{1}{2}\log_2 N$. The analysis takes into account the size of the data sets, the communication bandwidth, and the overhead in communication. We also provide some experimental data for the Intel *iPSC/d7*.

## 1. Introduction

Broadcasting of data from a single source to all other nodes in a multiprocessor system is an important operation. It is used in many parallel algorithms, for instance, in matrix multiplication, the solution of irreducible linear systems, and forming transitive closure. Examples of a variety of algorithms using specific forms of communication are contained in [6, 12, 11]. The reverse operation, reduction, occurs, for example, in computing inner products, solving linear recurrences, [14], and parallel prefix computation. A different situation occurs if the source node distributes personalized information to all other nodes. In this case no replication of information takes place during distribution (or reduction in the reverse operation). The collection of data to a single node and distribution of personalized messages to all other nodes is a useful operation for the solution of tridiagonal systems under certain combinations of start-up times for communication, communication bandwidth, and problem sizes [13]. Matrix transposition is another example of personalized communication in that every node sends different data to every other node [11].

Data communication in Boolean cubes has received significant interest recently due to the success of the Caltech Cosmic Cube project [20] and the availability of Boolean cube configured concurrent processors from Intel Scientific Computers, NCUBE, Ametek, Floating-Point Systems and Thinking Machines Corp. [7]. The embedding of complete binary trees is treated in [22, 11, 18, 3, 2]. Wu also discusses the embedding of $k$-ary trees, and Bhatt the embedding of arbitrary binary trees. Efficient routing using randomization for arbitrary permutations has been suggested by Valiant [21] . Broadcasting of data from a single source to all other nodes is studied in [18]. We propose a lower bound algorithm that offers a speed-up of a factor of $\log N$[1] over the algorithm in [18]. We also present lower bound algorithms for personalized communication. We give routing algorithms, and analyze the complexity in detail. The analysis is compared with experimental data.

A Boolean $n$-cube has $N = 2^n$ nodes, diameter $\log N$, $\binom{\log N}{i}$ nodes at distance $i$ from a given node, and $\log N$ disjoint paths between any pair of nodes. The paths are either of the same length as the Hamming distance between the end points of the paths, or the Hamming distance plus two [19]. The fanout of every node is $\log N$, and the total number of communication links is $\frac{1}{2}N\log N$. Any spanning tree can be used to broadcast data from a single source to all other nodes.[2] A node replicates the data as many times as correspond to the out-degree of the node in the spanning tree. In broadcasting one element (or packet), the minimum number of routing steps is $\log N$. Any spanning tree with height $\log N$ can achieve this lower bound, if each node can send out data through all the links connected to it during one step. In case each node can send or receive data through only one link during one step, then only the class of *spanning binomial trees* can attain the lower bound, since after each broadcasting step the number of nodes that own the desired data is at most twice that of the previous step. The $\log N$ lower bound is attained only if the number of nodes that own the desired data doubles at each step. This is exactly the definition of a binomial tree. A 0-level binomial tree has only 1 node. An $n$-level binomial tree is constructed out of two $(n-1)$-level binomial trees by adding one edge between the roots of the two trees, and by making either root the new root, [1, 4]. It follows from this recursive construction that:

1. An $n$-level binomial tree has $\binom{n}{i}$ nodes at level $i$.
2. The $n$-level binomial tree is composed of $n$ subtrees[3] each of which is a binomial tree of $0, 1, \ldots, n-1$ levels respectively. The $k$-level subtree has $2^k$ nodes.
3. An $n$-level binomial tree can be obtained from a $k$-level binomial tree, $k < n$, by replacing each node of the $k$-level binomial tree by an $(n-k)$-level binomial tree. The child nodes of a node in the $k$-level tree become children of the root of the replacing $(n-k)$-level binomial tree.

Since an $n$-level binomial tree can be embedded in an $n$-cube as a spanning tree, we call it a *Spanning Binomial Tree* (SBT). Note that the number of nodes at each level $i$ of the binomial tree is equal to the number of nodes at distance $i$ from a node in an $n$-cube.

In broadcasting $M$ elements using a packet size of $B$ elements, and by pipelining the communication from the root towards the leaves along any $\log N$ height spanning tree, the number of routing steps becomes $\lceil \frac{M}{B} \rceil + \log N - 1$, which is not optimal. Since each node has a fanout of $\log N$, a lower

[1] $\log N = \log_2 N$ throughout this paper.

[2] In particular, a Hamiltonian Path is also a spanning tree.

[3] In this paper "subtree" refers to "subtree of the root" unless stated otherwise.

bound for the number of routing steps is $\lceil \frac{M}{B \log N} \rceil + \log N - 1$. In order to achieve this lower bound, the data set has to be split into $\log N$ subsets, each of which is communicated over a distinct communications link from the source node. It follows that the nodes adjacent to the source node must be roots of subtrees spanning all but one node of the cube (the source node). The depth of the subtrees is $\log N$, and a tight lower bound for the number of routing steps, assuming concurrent bi-directional communication, is $\lceil \frac{M}{B \log N} \rceil + \log N$.

In sending personalized information from a single source to all other nodes, no replication of information takes place and the total number of packets that the source must send is $\lceil \frac{(N-1)M}{B} \rceil$ for $M$ elements per destination node. The number of routing steps for the *Spanning Binomial Tree* algorithm (SBT) is $\log N$ if the maximum packet size is sufficiently large $(NM/2)$; even so, the communication bandwidth is poorly utilized since the transfer time is at least proportional to $NM/2$. In the SBT, half of the nodes belong to one subtree, one quarter to another subtree, etc. A *Balanced Spanning Tree* (BST) is defined in that each subtree has approximately $\frac{N}{\log N}$ nodes. The data transfer on any link is limited to approximately $\frac{N}{\log N}M$. The BST algorithm offers a potential $\log N$ speed-up over the SBT algorithm in sending personalized information from a single source to all other nodes. In fact, lower bound algorithms for broadcasting from every node to every other node and sending personalized data from every node to every other node on a Boolean cube can be attained by using $N$ BST's rooted at each node concurrently. See [8] for details.

In section 2 we introduce the notation and some definitions used throughout the paper. Section 3 considers broadcasting from a single source to all other nodes, and section 4, personalized communication from a single source to all other nodes. Experimental results on the Intel *iPSC/d7* are presented in section 5.

## 2. Notation and Definitions

In the following, $N$ denotes the number of nodes in the Boolean $n$-cube and $n = \log_2 N$ the dimension of the cube. Nodes in the cube are assigned binary addresses such that adjacent nodes differ in precisely one bit. Address bits are numbered from 0 through $n-1$ with the lowest order bit being *bit 0*. Node $i$ is the node that has a binary address equal to $i$, i.e., $i = (a_{n-1}a_{n-2}\ldots a_0)$. Let $\oplus$ be the bit-wise exclusive-or operation. The $j^{th}$ port of a node $i$ connects to the node $k$ that differs from $i$ in the $j^{th}$ bit, i.e., $i \oplus k = (00...01_j0...0)$. There is a port for each address bit, and ports are numbered from 0 through $n-1$. Let $|i|$ denote the number of bits with value one in the binary number $i$; hence $|i \oplus j|$ denotes the Hamming distance between the binary numbers $i$ and $j$.

Let $i = (a_{n-1}a_{n-2}\ldots a_0)$. Define $R$ to be the right rotation function, i.e., $R(i) = (a_0a_{n-1}a_{n-2}\ldots a_1)$, and $R^j = R^{j-1} \circ R$ to mean a right rotation of $j$ steps. The *rotation* of a graph with binary node addresses is accomplished by applying the same rotation function to all its addresses. This is similarly the case for the *translation* of a graph. Clearly, adjacency is preserved under rotation and translation. The period of a binary number $i$, $P_i$, is the least $j$ such that $i = R^j(i)$. For example, the period of (011011) is 3. A binary number is *cyclic* if its period is less than its length; otherwise it is *non-cyclic*. A *relative address* of node $i$ in a spanning tree rooted at node $s$ is $i \oplus s$. A *cyclic node* is a node with cyclic relative address.[4] If one binary number can be derived by rotating another binary number, then they are in the same *generator set* $G$ (or *necklace* [15]). For example, (001001), (010010) and (100100) are in the same generator set. The number of elements in the generator set $G_i$ of $i$ is $P_i$.

In the graph model of the Boolean cube there is a node (vertex) for each node (processor with local memory) of the cube, and a pair of directed edges for each pair of nodes that differ in precisely one bit. The directed edges between a pair of nodes form a communication link. A *source node*, or a *root node*, is a node that only has edges directed away from it. A *sink node*, or a *leaf node*, only has edges directed to it. Nodes that are neither source (root) nor sink (leaf) nodes are *internal nodes*. The root of a tree is at *level 0* and traversing the edge away from the root increases the level by one. The height of a tree is equal to the label of the last level. The MSBT and BST are constructed out of $n$ subtrees labelled 0 through $n-1$ (from left to right in the Figures of this paper).

The communication is assumed to be packet switched. $M$ denotes the number of elements to be received by a node, $t_c$ the transfer time for an element, and $\tau$ the start-up time for the communication of a packet of maximum $B$ elements. With concurrent bi-directional communication we assume that a pair of adjacent nodes can exchange a pair of messages during the same communication step, or cycle. In a *port-oriented* routing algorithm, all information to be communicated over a port is sent before any communication is performed on any other port. In *packet-oriented* communication, a piece of information corresponding to a packet is communicated on all ports before a second packet is sent on any port.

## 3. Broadcasting

In this section, we will describe and compare routing algorithms for broadcasting based on a Spanning Binomial Tree (SBT) and Multiple Spanning Binomial Trees (MSBT). We first define the SBT and MSBT topologies, then state the communication complexity of the routing algorithms for the two topologies, assuming communication on only one port at a time (which, effectively, is the case with the Intel *iPSC*), and on all $\log N$ ports concurrently.

### 3.1. Spanning Binomial Trees

The familiar spanning tree rooted at node 0 of a Boolean $n$-cube contains the edges that connect a node $i$ with the subset of its neighbors having addresses obtained by complementing any bit of leading zeroes of the binary encoding of $i$, [5, 11, 16, 18, 19]. For an arbitrary source node $s$ the spanning tree is simply translated by a bit-wise exclusive-or operation on all addresses with the address of the source node; i.e., $c = i \oplus s$ is formed. Complementation of those bits of $i$ that correspond to the leading zeroes of $c$ defines the edges of the translated spanning tree. More precisely, let $s = (s_{n-1}s_{n-2}...s_0)$, $i = (a_{n-1}a_{n-2}...a_0)$, and $c = (c_{n-1}c_{n-2}...c_0)$, where $c_m = s_m \oplus a_m$. Let $c_k = 1$ and $c_m = 0, \forall m > k$ with $k = -1$ for $c = 0$, i.e., $k$ is the highest order bit of $c$ that is 1. Let $children(i, s)$ be the set of child nodes of node $i$ in the SBT rooted at node $s$ and $\mathcal{M}_{SBT}(i \oplus s) = \{k+1, \ldots, n-1\}$. Then,

$$children_{SBT}(i,s) = \{(a_{n-1}a_{n-2}\ldots\overline{a}_m\ldots a_0)\}, \quad \forall m \in \mathcal{M}_{SBT}(i \oplus s)$$

In implementing the routing algorithm for the SBT topology it is also convenient to introduce the inverse function, i.e., a

[4] A cyclic node is defined in terms of a spanning tree.

function that for each node defines its parent. Let $parent(i,s)$ be the parent of node $i$ in the spanning tree rooted at node $s$. Then

$$parent_{SBT}(i,s) = \begin{cases} \phi, & i = s; \\ (a_{n-1}a_{n-2}\ldots\overline{a}_k\ldots a_0), & i \neq s. \end{cases}$$

It is easy to verify that the parent and children functions are consistent. Figure 1 shows a spanning tree generated by the children (or parent) function for the root located at node 0 in a 4-cube.

### 3.2. Multiple Spanning Binomial Trees

The Multiple Spanning Binomial Trees (MSBT) graph can be viewed as being composed of $\log N$ SBT's with one tree rooted at each of the nodes adjacent to the source node. The SBT's are rotated such that the source node of the MSBT graph is in the smallest subtree of each SBT. The MSBT graph is then obtained by reversing the edges from the roots of the SBT's to the source node. After the edge reversal each SBT becomes an ERSBT (Edge Reversed Spanning Binomial Tree). The MSBT graph is not a tree. The diameter of the MSBT graph is $\log N + 1$, since the source node is adjacent to all the roots of the SBT's used in the definition of the MSBT graph, and each SBT is of height $\log N$. The total number of edges in the $\log N$ SBT's is $(N-1)\log N$, which is $\log N$ less than the total number of directed edges in the cube. Hence, if the $\log N$ SBT's are edge-disjoint, then all edges are used, except the edges directed from the roots of the SBT's towards the source node. The SBT's used for the construction of the MSBT graph can be obtained by translation and rotation of the SBT defined before. We refer to the SBT rooted at node $(00\ldots01_j0\ldots0)$ as the $j^{th}$ SBT of the MSBT graph. The $j^{th}$ ERSBT is obtained from the $j^{th}$ SBT by reversing the edge directed to node 0 (the source).

Let $i = (a_{n-1}a_{n-2}\ldots a_0)$ and $k$ be such that $a_k = 1$, and $a_m = 0, \forall m \in \mathcal{M}_{MSBT}(i,j)$ where $\mathcal{M}_{MSBT}(i,j) = \{(k+1) \bmod n, (k+2) \bmod n, \ldots, (j-1) \bmod n\}$. Hence, $k$ is the first bit to the right of bit $j$, cyclically, which is equal to one, if $k \neq j$. For the special case of $i = 0$ we define $k = -1$. For the $j^{th}$ ERSBT of the MSBT graph with source node 0, the set of child nodes, and the parent node, of node $i$ are defined as follows:

$$children_{MSBT}(i,j,0) = \begin{cases} (a_{n-1}a_{n-2}...\overline{a}_j...a_0), & \text{if } k = -1; \\ \{(a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i,j) \bigcup \{j\}, & \text{if } a_j = 1,\ k \neq j; \\ \{(a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i,j), & \text{if } a_j = 1,\ k = j; \\ \phi, & \text{if } a_j = 0, k \neq -1. \end{cases}$$

$$parent_{MSBT}(i,j,0) = \begin{cases} \phi, & \text{if } k = -1; \\ (a_{n-1}a_{n-2}...\overline{a}_j...a_0), & \text{if } a_j = 0, k \neq -1; \\ (a_{n-1}a_{n-2}...\overline{a}_k...a_0), & \text{if } a_j = 1. \end{cases}$$

All nodes with bit $j$ equal to zero are leaf nodes of the $j^{th}$ ERSBT, except node 0. Conversely, all nodes with $a_j = 1$ are internal nodes of the $j^{th}$ ERSBT. Figure 2 shows an MSBT graph with source node 0 in a 3-cube.

It can be shown that the $\log N$ directed ERSBT's are edge-disjoint and the height of the MSBT graph is minimal among all possible configurations of $\log N$ edge-disjoint spanning trees[8].

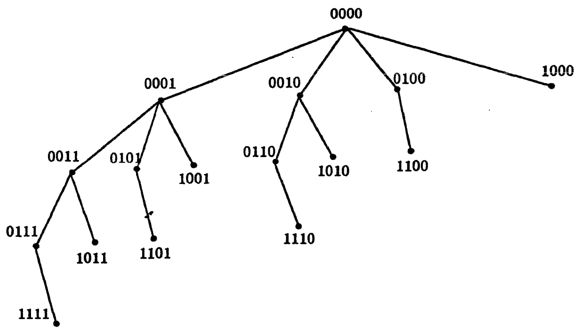


**Figure 1:** A spanning tree in a 4-cube.

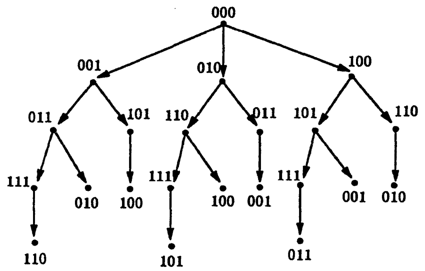


**Figure 2:** Three edge-disjoint directed spanning trees in a 3-cube.

For an arbitrary source node $s$ an MSBT graph is defined by translating the MSBT graph rooted at node 0. The only difference in the definition of the *parent* and *children* functions is that $k$ is determined from $c = i \oplus s$. Hence, for a source node $s$, $k$ is such that $c_k = 1$, and $c_m = 0, \forall m \in \mathcal{M}_{MSBT}(i \oplus s, j)$. For the special case of $c = 0$, $k = -1$.

$$children_{MSBT}(i,j,s) = \begin{cases} (a_{n-1}a_{n-2}...\overline{a}_j...a_0), & \text{if } k = -1; \\ \{(a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i \oplus s,j) \bigcup \{j\}, & \text{if } c_j = 1,\ k \neq j; \\ \{(a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i \oplus s,j), & \text{if } c_j = 1,\ k = j; \\ \phi, & \text{if } c_j = 0, k \neq -1. \end{cases}$$

$$parent_{MSBT}(i,j,s) = \begin{cases} \phi, & \text{if } k = -1; \\ (a_{n-1}a_{n-2}...\overline{a}_j...a_0), & \text{if } c_j = 0, k \neq -1; \\ (a_{n-1}a_{n-2}...\overline{a}_k...a_0), & \text{if } c_j = 1. \end{cases}$$

### 3.3. Communication Complexity of SBT- and MSBT-based Broadcasting

#### 3.3.1. Spanning Binomial Trees

With the communication restricted to one port at a time the data is first sent to the node that is the root of the largest subtree. Since the binomial tree is composed of two $(n-1)$-level binomial trees, the broadcasting operation is now reduced to the broadcasting of data in two same-sized, disjoint, subtrees of the cube. The process is repeated $\log N$ times and the complexity is $T = \lceil \frac{M}{B} \rceil (\tau + Bt_c) \log N$. Clearly $B_{opt} = M$ and $T_{min} = \log N(\tau + Mt_c)$. The data transfer time is independent of the packet size, but the number of start-ups decreases.

With a capability of concurrent communication on all ports, pipelining can be employed extensively. The propagation time to the node farthest away from the source is at least $\log N(\tau + Bt_c)$. When this node has received all packets the broadcasting is terminated. Hence, $T = (\lceil \frac{M}{B} \rceil + \log N - 1)(\tau + Bt_c)$, $B_{opt} = \sqrt{\frac{M\tau}{t_c(\log N - 1)}}$, and $T_{min} = (\sqrt{Mt_c} + \sqrt{\tau(\log N - 1)})^2$. The communication complexity estimates for the SBT are also given in [18] and are included here for easy reference.

### 3.3.2. Multiple Spanning Binomial Trees

We consider the cases with communication restricted to one send *or* one receive operation at a time, one send *and* one receive operation concurrently, and concurrent communication on all ports. The minimum number of routing steps to broadcast $\log N$ packets is $2\log N$ with communication restricted to one send *and* one receive operation at a time. To realize this lower bound it is required that a routing algorithm be found that allows concurrent communication within all subtrees without violating the constraint on concurrent communication. We describe such a routing algorithm in terms of labelling the MSBT graph with the least label being 0. A valid labelling for the restriction of one receive and one send operation concurrently (per node) and allowing pipelining every $\log N$ cycles, requires that the following three conditions be satisfied:

1. For any node of each subtree the least label on the output edges is greater than the label on the input edge.
2. For any cube node the labels on its input edges are distinct modulo $\log N$. (If there is more than one packet per subtree then the root can send out a new packet to every subtree every $\log N$ cycles.)
3. For any cube node the labels on the output edges are distinct modulo $\log N$.

Let $i = (a_{n-1}a_{n-2}...a_0)$ and $f(i,j)$ be the label of the input edge of node i in the $j^{th}$ subtree for an MSBT graph with source node $s$. Let $c = i \oplus s$, $c_k = 1$ and $c_m = 0, \forall m \in \mathcal{M}_{MSBT}(i \oplus s, j)$. If $c = 0$ then $k = -1$. Define

$$f(i,j) = \begin{cases} \phi, & \text{if } k = -1; \\ j + n, & \text{if } c_j = 0, k \neq -1; \\ k, & \text{if } c_j = 1, k \geq j; \\ k + n, & \text{if } c_j = 1, k < j. \end{cases}$$

It can be proved that function $f$ satisfies these three conditions[8]. From the labelling scheme, the largest label of all the input edges is $2n - 1$, i.e., broadcasting the first $\log N$ packets (one packet per subtree) can be done in $2\log N$ steps. The MSBT graph allows $M$ elements to be broadcast in $\lceil \frac{M}{B} \rceil + \log N$ routing steps under the constraint of one receive operation concurrent with one send operation. This is a strict lower bound for $\frac{M}{B} > 1$[8]. Figure 3 shows an MSBT graph for a 3-cube labelled by the function $f$ defined above.

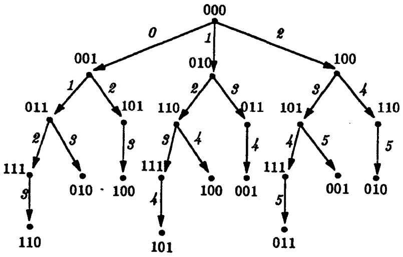


**Figure 3:** Routing in an MSBT graph with communication on one port at a time.

For communication restricted to one send *or* one receive operation per node, we can transform each previously defined cycle into two cycles. Notice that in the previous routing algorithm, all the communication links are used in only one direction during the first $n$ routing steps, and in the last routing step. Hence the MSBT graph allows $M$ elements to be broadcast in $2\lceil \frac{M}{B} \rceil + \log N - 1$ routing steps under the constraint of at most one receive or one send operation during each step. This is a strict lower bound for $\frac{M}{B} > 1$[8].

With a maximum packet size of $B$ and one receive operation concurrent with a send operation the communication complexity for MSBT-based broadcasting is $T = (\lceil \frac{M}{B} \rceil + \log N)(\tau + Bt_c)$, which is minimized for $B_{opt} = \sqrt{\frac{M\tau}{t_c \log N}}$. $T_{min} = (\sqrt{Mt_c} + \sqrt{\tau \log N})^2$. Restricting the communication to one send *or* one receive operation at a time, $T = (2\lceil \frac{M}{B} \rceil + \log N - 1)(\tau + Bt_c)$ and $T_{min} = (\sqrt{2Mt_c} + \sqrt{\tau(\log N - 1)})^2$.

For the case in which communication on all ports can take place concurrently the communication complexity is $T = (\lceil \frac{M}{B \log N} \rceil + \log N)(\tau + Bt_c)$. With optimal packet size $B_{opt} = \frac{1}{\log N}\sqrt{\frac{M\tau}{t_c}}$, $T_{min} = (\sqrt{\frac{Mt_c}{\log N}} + \sqrt{\tau \log N})^2$.

### 3.4. Comparison and Conclusion

In the following, we also compare the MSBT algorithm with the one based on a Two-rooted Complete Binary Tree (TCBT) [2] or a Hamiltonian Path (HP) as a broadcasting tree respectively. Table 1 shows the propagation delay of various algorithms. Interestingly, broadcasting through a Hamiltonian Path on a hypercube may be faster than broadcasting based on the SBT or even the TCBT, depending on the values of $M$, $t_c$, $\tau$ and $N$. With communication on all the ports concurrently, the MSBT-based algorithm can send out $\log N$ distinct packets every cycle while the SBT- and TCBT-based algorithms can only send out one distinct packet every cycle. Table 2 compares the number of cycles per distinct packet for various algorithms. Some variations exist, such as using two Hamiltonian paths with opposite directions sending distinct data, or using one Hamiltonian path such that the source node is at the center of the path. However, these variations only affect (either increase or decrease) delays, and the number of cycles per packet, by at most a factor of two.

The complexity estimates are summarized in Table 3. A potential for concurrent communication on all ports reduces the number of sequential start-ups and the bandwidth requirement by a factor of approximately $\log N$ for an arbitrary packet size in both SBT- and MSBT-based broadcasting. TCBT-based broadcasting does not fully utilize the bandwidth of a cube. The reduction in communication complexity for concur-

| Algorithm | 1 $s$ or $r$ | 1 $s$ and $r$ | all ports |
|---|---|---|---|
| HP | $N-1$ | $N-1$ | $N-1$ |
| SBT | $\log N$ | $\log N$ | $\log N$ |
| TCBT | $2\log N - 2$ | $2\log N - 2$ | $\log N$ |
| MSBT | $3\log N - 1$ | $2\log N$ | $\log N + 1$ |

**Table 1:** Propagation delays.

| Algorithm | 1 $s$ or $r$ | 1 $s$ and $r$ | all ports |
|---|---|---|---|
| HP | 2 | 1 | 1 |
| SBT | $\log N$ | $\log N$ | 1 |
| TCBT | 3 | 2 | 1 |
| MSBT | 2 | 1 | $1/\log N$ |

**Table 2:** Number of cycles per distinct packet.

| Algorithm | T | $B_{opt}$ | $T_{min}$ |
|---|---|---|---|
| HP, 1 $s$ or $r$ | $(2\lceil \frac{M}{B} \rceil + N - 3)(\tau + Bt_C)$ | $\sqrt{\frac{2M\tau}{(N-3)t_c}}$ | $(\sqrt{2Mt_c} + \sqrt{(N-3)\tau})^2$ |
| HP, 1 $s$ & $r$ | $(\lceil \frac{M}{B} \rceil + N - 3)(\tau + Bt_C)$ | $\sqrt{\frac{M\tau}{(N-3)t_c}}$ | $(\sqrt{Mt_c} + \sqrt{(N-3)\tau})^2$ |
| SBT, 1 port | $\lceil \frac{M}{B} \rceil \log N(\tau + Bt_c)$ | $M$ | $\log N(Mt_c + \tau)$ |
| SBT, $\log N$ ports | $(\lceil \frac{M}{B} \rceil + \log N - 1)(\tau + Bt_c)$ | $\sqrt{\frac{M\tau}{(\log N-1)t_c}}$ | $(\sqrt{Mt_c} + \sqrt{\tau(\log N - 1)})^2$ |
| TCBT, 1 $s$ or $r$ | $(3\lceil \frac{M}{B} \rceil + 2\log N - 5)(\tau + Bt_c)$ | $\sqrt{\frac{3M\tau}{(2\log N-5)t_c}}$ | $(\sqrt{3Mt_c} + \sqrt{\tau(2\log N - 5)})^2$ |
| TCBT, 1 $s$ & $r$ | $2(\lceil \frac{M}{B} \rceil + \log N - 2)(\tau + Bt_c)$ | $\sqrt{\frac{M\tau}{(\log N-2)t_c}}$ | $2(\sqrt{Mt_c} + \sqrt{\tau(\log N - 2)})^2$ |
| TCBT, $\log N$ ports | $(\lceil \frac{M}{B} \rceil + \log N - 1)(\tau + Bt_c)$ | $\sqrt{\frac{M\tau}{t_c(\log N-1)}}$ | $(\sqrt{Mt_c} + \sqrt{\tau(\log N - 1)})^2$ |
| MSBT, 1 $s$ or $r$ | $(2\lceil \frac{M}{B} \rceil + \log N - 1)(\tau + Bt_c)$ | $\sqrt{\frac{2M\tau}{t_c(\log N-1)}}$ | $(\sqrt{2Mt_c} + \sqrt{\tau(\log N - 1)})^2$ |
| MSBT, 1 $s$ & $r$ | $(\lceil \frac{M}{B} \rceil + \log N)(\tau + Bt_c)$ | $\sqrt{\frac{M\tau}{t_c \log N}}$ | $(\sqrt{Mt_c} + \sqrt{\tau \log N})^2$ |
| MSBT, $\log N$ ports | $(\lceil \frac{M}{B\log N} \rceil + \log N)(\tau + Bt_c)$ | $\frac{1}{\log N}\sqrt{\frac{M\tau}{t_c}}$ | $(\sqrt{\frac{Mt_c}{\log N}} + \sqrt{\tau \log N})^2$ |

**Table 3:** Communication Complexity based on various graphs.

| Communication Assumption | Algorithm | one packet | $\frac{M}{B} \gg \log N$ | $B = B_{opt}$, $\tau \log N \gg Mt_c$ | $B = B_{opt}$, $\tau \log N \ll Mt_c$ |
|---|---|---|---|---|---|
| 1 *send* or *recv* | SBT/MSBT | $\frac{\log N}{\log N+1}$ | $\frac{1}{2}\log N$ | 1 | $\frac{1}{2}\log N$ |
| 1 *send* or *recv* | TCBT/MSBT | $\frac{2\log N-2}{\log N+1}$ | 1.5 | 2 | 1.5 |
| 1 *send* and *recv* | SBT/MSBT | $\frac{\log N}{\log N+1}$ | $\log N$ | 1 | $\log N$ |
| 1 *send* and *recv* | TCBT/MSBT | $\frac{2\log N-2}{\log N+1}$ | 2 | 2 | 2 |
| $\log N$ ports | SBT,TCBT/MSBT | $\frac{\log N}{\log N+1}$ | $\log N$ | 1 | $\log N$ |

**Table 4:** Communication complexity compared to the MSBT routing.

rent communication on all ports is a factor of 2 or 3. Optimizing the packet size for each situation brings the number of start-ups to $O(\log N)$, irrespective of whether communication on one port or $\log N$ ports at a time is possible.

The MSBT-based broadcasting always offers a reduction in the bandwidth requirement for individual communication links by a factor of approximately $\log N$ over SBT-based broadcasting. With communication on all ports concurrently, the MSBT-based broadcasting has a communication complexity that is lower than that of TCBT-based broadcasting by a factor of $\log N$. Even with communication only on one port at a time, MSBT-based broadcasting still is faster than TCBT-based broadcasting by a factor of 1.5 or 2. The communication complexities of broadcasting based on the SBT and the TCBT are compared with that based on the MSBT in Table 4.[5]

## 4. Personalized Communication

In personalized communication no replication of information takes place during distribution, nor is there any reduction during the reverse operation. In broadcasting, the bandwidth requirement grows with the distance $i$ from the source node precisely as the number of nodes grow. In personalized communication the bandwidth requirement instead decreases in proportion to the number of nodes less than or equal to distance $i$ from the source. The root is the "bottleneck" in personalized communication. In this section we define a pruning strategy for the MSBT graph that generates a *balanced spanning tree* (BST) of height $\log N$. If concurrent communication on all $\log N$ ports (of the root) is possible, then a lower bound for the transmission time is $\frac{N}{\log N}Mt_c$, and a lower bound for the number of start-ups is $\log N$. The BST makes possible personalized communication in a time corresponding to this lower bound.

### 4.1. A Balanced Spanning Tree

In the SBT topology a node $i$ belongs to the $j^{th}$ subtree iff $a_j = 1, a_k = 0, k < j$. In the MSBT graph a node is an internal node of the $j^{th}$ ERSBT if $a_j = 1$. Bit $j$ can be considered as a *base* for $j^{th}$ subtree. For the BST we define the base as follows:

Let $J_i = \{j_1, j_2, \ldots, j_m\}$, where $j_1 < j_2 < \ldots j_m$, $R^u(i) = R^v(i)$, $u, v \in J_i$, and $R^u(i) < R^l(i)$, $u \in J_i$, $l \notin J_i$. $|J_i| = n/P_i$ where $P_i$ is the period of $i$. Then $base(i) = j_1$ and node $i$ is assigned to subtree $j_1$, i.e., the value of the base equals the minimum number of right rotations such that the rotated number has a minimum value among all the rotated values.[6] For example, $base((011010)) = 3$ and $base((110110)) = 1$. The period of (011010) is 6 and the period of (110110) is 3. For ease of notation we omit the subscript on $j$ in the following. For the definition of the *parent* and *children* functions we first find the position $k$ of the first bit cyclically to the right of bit $j$ that is equal to 1, i.e., $a_k = 1$, and $a_m = 0, \forall m \in \mathcal{M}_{MSBT}(i,j)$, ($k = j$, if every bit but $j$ is 0). For $i = 0$, $k = -1$. Then

$$parent_{BST}(i,0) = \begin{cases} \phi, & \text{if } i = 0; \\ (a_{n-1}a_{n-2}...\overline{a}_k...a_0), & \text{otherwise.} \end{cases}$$

$$children_{BST}(i,0) = \begin{cases} \{(a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, \forall m \in \{0,1,\ldots,n-1\}, & \text{if } i = 0; \\ \{q_m = (a_{n-1}a_{n-2}...\overline{a}_m...a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i,j) \text{ and } base(q_m) = base(i), & \text{if } i \neq 0. \end{cases}$$

[5] Notice that the entry for the last column and the last row in the table is based on the assumption that $B = B_{opt}, \tau \log^2 N \ll Mt_c$.

[6] The notion of *base* is similar to the idea of distinguished node used in [15] in that *base* = 0 distinguishes a node from a generator set (necklace).

The $parent_{BST}$ function preserves the base, since for any node $i$ with base $j$, $a_k$ is the highest order bit of $R^j(i)$. Complementing this bit cannot change the base. It is also readily seen that the $parent_{BST}$ and $children_{BST}$ functions are consistent.

Figure 4 shows the spanning tree generated by the algorithm above for the root located at node 0 in a 5-cube.

For an arbitrary source node $s$ we translate the BST rooted at node 0 to node $s$ by performing for each node the bit-wise exclusive-or function of its address and the address of the source node. The base of a node is determined from $c = i \oplus s$, and the children and parent functions are readily modified.

Let $J_{i,s} = \{j_1, j_2, \ldots, j_m\}$, where $j_1 < j_2 < \ldots j_m$, $R^u(c) = R^v(c)$, $u, v \in J_{i,s}$, and $R^u(c) < R^l(c)$, $u \in J_{i,s}$, $l \notin J_{i,s}$. Then $base(c) = j_1$. Then $k$ is defined by $c_k = 1$ and $c_m = 0, \forall m \in \mathcal{M}_{MSBT}(i \oplus s, j)$ with $k = -1$ if $c = 0$.

$$parent_{BST}(i,s) =$$
$$\begin{cases} \phi, & \text{if } c = 0; \\ (a_{n-1}a_{n-2}\ldots\overline{a}_k\ldots a_0), & \text{otherwise.} \end{cases}$$

$$children_{BST}(i,s) =$$
$$\begin{cases} \{(a_{n-1}a_{n-2}\ldots\overline{a}_m\ldots a_0)\}, \forall m \in \{0,1,\ldots,n-1\}, & \text{if } c = 0; \\ \{q_m = (a_{n-1}a_{n-2}\ldots\overline{a}_m\ldots a_0)\}, & \\ \quad \forall m \in \mathcal{M}_{MSBT}(i \oplus s, j) & \\ \quad \text{and } base(q_m \oplus s) = base(i \oplus s), & \text{if } c \neq 0. \end{cases}$$

**Lemma 4.1.** *The number of nodes in a subtree is of order* $O(\frac{N}{\log N})$.

*Proof.* With $A$ cyclic nodes there are at least $(N - A)/\log N$ nodes in a subtree. Denoting the number of generator sets for cyclic nodes by $B$ it follows that the maximum number of nodes in a subtree is $(N-A)/\log N + B - 1$. To derive bounds on $A$ we use the complex plane diagram used by Hoey and Leiserson [9] in studying the shuffle-exchange network. Leighton[15] shows that $B = O(\sqrt{N})$.

Full necklaces, i.e., non-cyclic nodes, are mapped to circles. Degenerate necklaces, i.e., cyclic nodes, are mapped to the origin. In the context of the shuffle-exchange graph each node that is mapped to the origin of the complex plane is adjacent (via an exchange edge) to a node at position (1,0) or (−1,0). Hence, for every full necklace of $\log N$ nodes there are at most 2 cyclic nodes. Node 0 is adjacent to a node of a full necklace, and so is node $N - 1$ (for $\log N > 2$). It follows that an upper bound on $A$ is $2\frac{N-\log N}{2+\log N}$ and the number of nodes in a subtree is at least $\frac{N+2}{2+\log N}$. The relative difference in the number of nodes in the maximum and minimum subtrees approaches 0 for $N \rightarrow \infty$. ∎

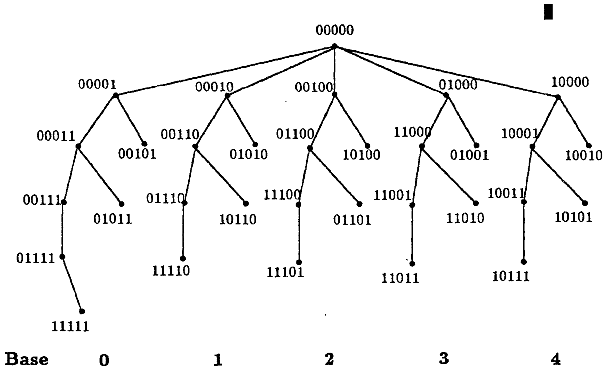


**Figure 4:** A balanced spanning tree in a 5-cube.

| $n$ | BST(max) | $(N-1)/\log N$ | ratio |
|---|---|---|---|
| 2 | 2 | 1.50 | 1.33 |
| 3 | 3 | 2.33 | 1.29 |
| 4 | 5 | 3.75 | 1.33 |
| 5 | 7 | 6.20 | 1.13 |
| 6 | 13 | 10.50 | 1.24 |
| 7 | 19 | 18.14 | 1.05 |
| 8 | 35 | 31.88 | 1.10 |
| 9 | 59 | 56.78 | 1.04 |
| 10 | 107 | 102.30 | 1.05 |
| 11 | 187 | 186.09 | 1.00 |
| 12 | 351 | 341.25 | 1.03 |
| 13 | 631 | 630.08 | 1.00 |
| 14 | 1181 | 1170.21 | 1.01 |
| 15 | 2191 | 2184.47 | 1.00 |
| 16 | 4115 | 4095.94 | 1.00 |
| 17 | 7711 | 7710.06 | 1.00 |
| 18 | 14601 | 14563.50 | 1.00 |
| 19 | 27595 | 27594.05 | 1.00 |
| 20 | 52487 | 52428.75 | 1.00 |

**Table 5:** A Comparison of maximum subtree sizes of the Balanced Spanning Tree and values of $(N-1)/\log N$.

Table 5 gives the sizes of the maximum subtrees generated according to the definition of the BST for up to 20-dimensional cubes. The relative difference approaches 0 rapidly. The last column contains the ratio of BST(max) to $(N-1)/\log N$.

Some properties of the BST are listed below. For detailed proofs see [8].

1. The height of one subtree is $\log N$, and the height of all other subtrees is $\log N - 1$.
2. The maximum fanout of any node at level $i$ in a BST is $\lceil\frac{\log N - i}{2}\rceil$ for $1 \leq i \leq logN$.
3. Let $\phi(i,j)$ be the number of nodes at distance $j$ from node $i$ in the subtree rooted at node $i$. Then, $\phi(i,j) \geq \phi(k,j)$ where node $k$ is a child of node $i$.[7]
4. Excluding node $i \oplus s = (11\ldots1)$, all the subtrees of the BST are isomorphic if $\log N$ is a prime number.
5. Subtrees $P$ to $\log N - 1$ contain no cyclic nodes with period $P$.
6. Any cyclic node is a leaf node of the BST.

## 4.2. The Complexity of Personalized Communication Based on the SBT and BST Topologies

### 4.2.1. Spanning Binomial Trees

For SBT-based distribution restricted to communication on one port at a time, the communication complexity for a maximum packet size of $B$ is $T \simeq (N-1)Mt_c + \tau(NM/B + \log\lceil\frac{B}{M}\rceil - 1)$, $M \leq B \leq NM/2$, which is minimized for $B = NM/2$ yielding $T = (N-1)Mt_c + \tau \log N$. For $B \leq M$, $T \simeq (NM/B - 1)(Bt_c + \tau)$. There exist several algorithms of this complexity. One such algorithm sends the cumulative data for the largest subtree to the root of this subtree first. Both nodes then recursively execute the same algorithm in their own $(n-1)$-subcube.

**Lemma 4.2.** *In distributing data from the root in a level-by-level order starting from level* $\log N$, *the time to complete the*

[7] This property is required in deriving the communication complexity for the BST routing.

*distribution is determined by the root, which terminates the distribution in a time proportional to the lower bound for a sufficiently large packet size.*

*Proof.* With potential concurrent communication on $\log N$ ports, the root can send data for level $\log N - i$ during step $i$, $0 \leq i \leq \log N - 1$, assuming a sufficiently large packet size. The amount of data sent from the root during step $i$ to the largest subtree is $\binom{\log N-1}{i}M$. The amount of data sent during the same step from the node at level $j$ to its largest subtree, which is the largest subtree at that level, is $\binom{\log N-1-j}{i-j}M$, where $1 \leq j \leq \log N - 1$. Since $\binom{\log N-1}{i} \geq \binom{\log N-1-j}{i-j}$ for all $i, j \geq 1$, and any other subtree rooted at level $j$ is isomorphic to a subgraph of the highest subtree rooted at level $j$, the transfer time is determined by the transfer time of the root. For a sufficiently large packet size, i.e., $B \geq \binom{\log N-1}{\frac{\log N-1}{2}}M \approx \frac{NM}{\sqrt{2\pi(\log N-1)}}$, $T_{min} = N/2Mt_c + \log N\tau$.

∎

With a potential for concurrent communication on $\log N$ ports, a reduction in the transfer time by a factor of 2 is possible compared to communication on one port at a time. We conclude that for SBT-based algorithms the packet size is of greater importance than concurrent communication on all ports.

### 4.2.2. Balanced Spanning Trees

With BST-based personalized communication restricted to one port at a time the root can send data to the subtrees cyclically. With a maximum packet size $B > M$, data for several nodes can be merged into one packet. The receiving node has sufficient time to retransmit pieces on all its ports, should that be required, since a new packet only arrives every $\log N$ cycles. The root requires a time of $T \simeq \frac{(N-1)M}{B\log N}(\tau + Bt_c)\log N$, which, if the data to the most remote nodes is transmitted first, is also the time to completion. For $B = M$, $T = (N-1)(\tau + Mt_c)$, i.e., the same as in the SBT algorithm. For $B \geq \frac{N}{\log N}M$ the root of the BST need only perform one communication per subtree, and it completes the communication in a time of $T = \tau\log N + (N-1)Mt_c$. But, unlike the SBT algorithm, the communication is not terminated when the root is done. The message to the last visited subtree needs to traverse $\log N - 2$ communication links. The bandwidth requirement of each subtree can be shown to be $\approx \frac{2N\log\log N}{\log N}$ (see [8] for detailed proof). An upper bound on the time for personalized communication based on the BST with unbounded packet size is $T = (2\log N - 2)\tau + N(1 + \frac{2\log\log N}{\log N})Mt_c$. The number of start-ups is almost twice that of the SBT-based personalized communication, and the total transfer time is higher by a lower order term. The time for personalized communication based on the BST is minimized for $B \geq \frac{N}{\log N}M$.

With a potential for communication on $\log N$ ports at a time, the time for personalized communication based on the BST topology is $T \simeq \frac{(N-1)M}{B\log N}(\tau + Bt_c)$ for $B \leq M$ and $T \simeq \sum_{i=1}^{\log N}(\lceil\binom{\log N}{i}\frac{M}{B_i\log N}\rceil + B_it_c)$, $B_i = min(B, \lceil\binom{\log N}{i}\frac{M}{B_i\log N}\rceil)$ for $B > M$. This complexity estimate is valid if data for nodes at distance $i$ are sent during step $\log N - i$, $1 \leq i \leq \log N$. If $B = M$ then $T \simeq \frac{N-1}{\log N}(\tau + Mt_c)$. The communication time is minimized for $B \geq \frac{N}{\log^{3/2} N}M$ by using a *level-by-level* algorithm as in lemma 4.2. By property **p3** of the BST, it follows that the amount of data sent from the root to any subtree during step $i$ is no less than the amount of data sent from any node during the same step. Hence, $T_{min} = \tau\log N + \frac{N-1}{\log N}Mt_c$, the minimum possible.

### 4.3. Comparison and Conclusion

With communication on one port at a time, if the fixed maximum packet size $B \leq M$ holds, then the complexity of SBT- and BST-based personalized communication is the same. For $B > M$, the SBT-based routing algorithm yields a lower complexity than the BST-based routing. For a sufficiently large maximum packet size the SBT-based algorithm has $\log N$ start-ups compared to $2\log N - 2$ start-ups for the BST-based algorithm. The transmission times are comparable, though the transmission time for the BST routing is higher. Note that, as in broadcasting, the minimum number of start-ups can be accomplished for sufficiently large maximum packet size.

With concurrent communication on $\log N$ ports the number of start-ups and the transmission time of BST-based routing is lower than that for the SBT by a factor of $\frac{1}{2}\log N$ for a maximum packet size $B \leq M$. With a sufficiently large packet size all routings yield a minimum of $\log N$ start-ups, but the BST routing has a total transmission time that is lower than that of SBT by a factor of $\frac{1}{2}\log N$. Moreover, it is achieved at a maximum packet size of $\frac{N}{\log^{3/2} N}M$, compared to a maximum packet size of $\frac{N}{\sqrt{2\pi(\log N-1)}}M$ for the SBT routing. We conclude that if communication can be performed on all $\log N$ ports concurrently, the communication complexity of the BST routing may be lower by a factor of $\frac{1}{2}\log N$ compared to the SBT routing. Table 6 lists the communication complexity for optimal packet size. The timing for TCBT routing is also included for easy comparison. See [8] for detailed analysis.

| Algorithm | $T_{min}$ |
|---|---|
| SBT, 1 port | $(N-1)Mt_c + \log N\tau$ |
| SBT, $\log N$ ports | $N/2Mt_c + \log N\tau$ |
| TCBT, 1 port | $\leq (2N - 2\log N - 1)Mt_c + (2\log N - 2)\tau$ |
| TCBT, $logN$ port | $(\frac{3}{4}N - 1)Mt_c + \log N\tau$ |
| BST, 1 port | $\leq N(1 + \frac{2\log\log N}{\log N})Mt_c + (2\log N - 2)\tau$ |
| BST, $\log N$ ports | $\simeq (N-1)/\log NMt_c + \log N\tau$ |

**Table 6:** Communication complexity of personalized communication.

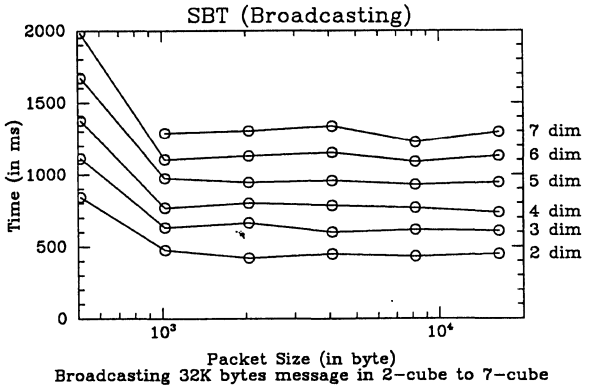
## 5. Experimental Results

### 5.1. Single Source Broadcasting Based on the SBT and MSBT

Figure 5 shows the measured time to completion of a broadcasting operation based on the SBT topology for cubes of various dimensions and a number of different external packet sizes. As expected, the communication time increases almost linearly for external packet sizes below $1k$ bytes. Figure 6 shows the measured time of SBT- and MSBT-based broadcasting for a message of $60k$ bytes with each packet being $1k$ bytes and for cube dimensions ranging from 2 to 6. Figure 7 shows the speed-up of broadcasting based on the MSBT topology over the SBT topology. The measured speed-up is approximately $\log N$, as predicted.

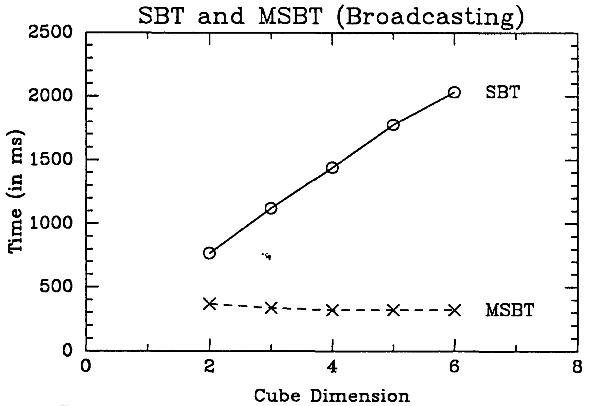
### 5.2. Single Source Personalized Communication

In implementing the SBT routing on the Intel *iPSC*, the root processes the data in descending order starting with the relative address $N - 1$. This order implies that data is transmitted over ports in an order corresponding to the transition sequence in a binary-reflected Gray code[17]. Hence, port 0 is



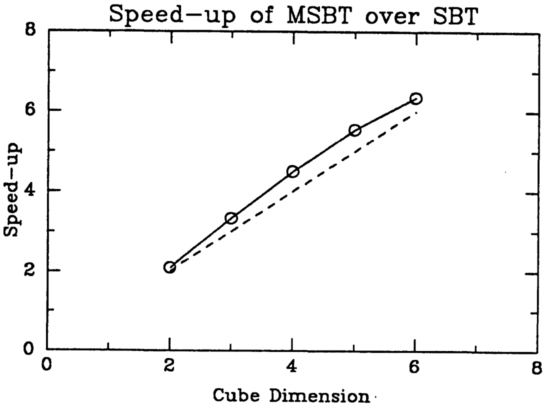
Broadcasting 32K bytes message in 2–cube to 7–cube

**Figure 5:** Broadcasting using SBT.



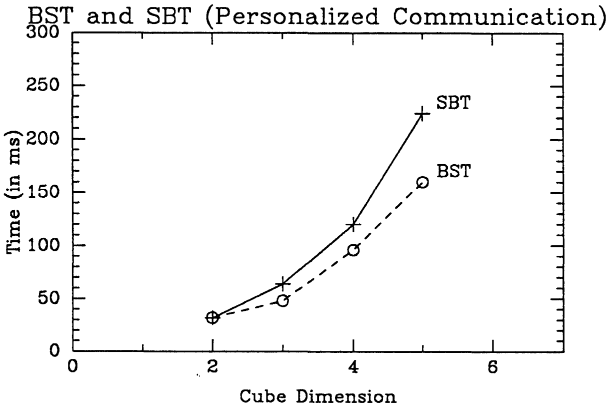
Broadcasting 60K bytes message in 2–cube to 6–cube

**Figure 6:** Broadcasting using SBT and MSBT.



broadcasting 60K bytes message in 2–cube to 6–cube
packet size = 1 K, solid line = measured speed–up
dashed line means "speed–up = logN"

**Figure 7:** Speed up of MSBT vs. SBT.



message size = 1 K bytes per node

**Figure 8:** Personalized communication using BST and SBT.

used every other cycle, port 1 every fourth cycle, etc. Internal nodes retransmit a message on a port chosen from among the ones that correspond to the leading zeroes in its relative address. The choice is made according to a binary-reflected Gray code on the leading zeroes.

For the implementation of the BST-based algorithm the routing order needs to be determined for each subtree of the root. Excluding cyclic nodes, the subtrees are isomorphic. The root only needs to keep one table of length $\approx \frac{N}{\log N}$ with each entry of size $\log N$ bits. The order of the entries corresponds to the transmission order for each port. The table entry points to the messages transmitted over port 0. The pointers for the other ports are simply obtained by (right) cyclic shifts of the table entries. The cyclic nodes can be handled by finding the period $P$ for each cyclic table entry, and not transmitting the message corresponding to this table entry for ports with index $j \geq P$.

For each subtree a depth-first or a reversed breadth-first order are viable transmission orders. With reversed breadth-first order we mean a breadth-first traversal of the subtree starting from the last level ($\log N - 1$ or $\log N - 2$ depending on subtree). The source node determines the order, and internal nodes can either route according to the destination address if it is included, or by the use of tables. If tables are used, then in the case of depth-first communication order it suffices for each internal node to keep a count for each port. Since the number of ports used in each subtree is at most $\frac{\log N}{2}$, and the number of nodes in the entire subtree is approximately $\frac{N}{\log N}$, a bound on the table size in each node is $\log^2 N$ bits. A breadth-first communication order can be implemented by internal nodes keeping a table of how many nodes there are at a given level in each of its subtrees. The table has at most $\log^2 N$ entries. An upper bound for the number of nodes in a subtree at any level is $\frac{N}{\log^{3/2} N}$, and the total table size in a node is approximately $\log^3 N$ bits. Hence, without a more sophisticated encoding the depth-first communication order is more effective with respect to table space. The measurements presented in Figure 8 are based on an implementation using a depth first order.

With communication on one port at a time the expected time for personalized communication based on the SBT topology or the BST topology is the same. The observed advantage of the BST- over the SBT-based communication is due to the fact that the BST can take better advantage of the overlap between communication on different ports. In the SBT case, the node with relative address (00...01) is not yet finished retransmitting the last packet received when a new packet arrives. In the BST a subtree receives a packet once every $\log N$ cycles, and full advantage of the 20% overlap in communications actions is taken.

## 6. Conclusion

We have shown that the Boolean $n$-cube topology allows for the embedding of $n$ edge-disjoint binomial trees, and we presented routing algorithms for broadcasting that have a complexity equal to the lower bound both for communication restricted to one port at a time and for concurrent communication on all $n$ ports of a node. We have also defined a balanced spanning tree for personalized communication. Each subtree of the balanced spanning tree we have defined has approximately $\frac{N}{\log N}$ nodes. For communication on one port at a time, personalized communication based on the balanced spanning tree has the same complexity as personalized communication based on a binomial tree for certain maximum packet sizes, and has at most a factor of 2 higher complexity in other cases. With concurrent communication on all ports, the routing based on the balanced spanning tree is superior by a factor of $\frac{1}{2} \log N$ for a variety of combinations of maximum packet sizes, start-up times, transfer rates, and data sizes.

Experimental results on the Intel *iPSC/d7* confirm the results of the analysis.

## 7. Acknowledgement

Many thanks go to Andrea Pappas and Chris Hatchell for their assistance with the manuscript. The generous support of the Office of Naval Research under contract N00014-84-K-0043 is gratefully acknowledged.

# References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*, pages 164–165. Addison-Wesley, 1983.

[2] S. Bhatt and I. Ipsen. *How to Embed Trees in Hypercubes.* Technical Report YALEU/CSD/RR-443, Yale University, Dept. of Computer Science, December 1985.

[3] S.R. Deshpande and R.M. Jenevein. *Scaleability of a Binary Tree on a Hypercube.* Technical Report TR-86-01, University of Texas at Austin, January 1986.

[4] M.J. Fischer. *Efficiency of Equivalence Algorithms*, pages 153–167. Plenum Press, 1972.

[5] G.C. Fox, S.W. Otto, and A.J.G. Hey. *Matrix Algorithms on a Hypercube I: Matrix Multiplication.* Technical Report Hm 206, Caltech, October 1985.

[6] D. Gannon and J. Van Rosendale. On the impact of communication complexity in the design of parallel numerical algorithms. *IEEE Trans. Computers*, C-33(12):1180–1194, December 1984.

[7] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[8] C.-T. Ho and S.L. Johnsson. *Tree Embeddings and Optimal Routing for Data Distribution in Hypercubes.* Technical Report YALEU/DCS/RR-475, Department of Computer Science, Yale University, 1986.

[9] D. Hoey and C.E. Leiserson. A layout for the shuffle-exchange network. In *Proc. 1980 International Conference on Parallel Processing*, IEEE Computer Society, August 1980.

[10] *iPSC System Overview.* Intel Corp., January 1986.

[11] S.L. Johnsson. *Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures.* Technical Report YALEU/CSD/RR-361, Dept. of Computer Science, Yale University, January 1985.

[12] S.L. Johnsson. *Odd-Even Cyclic Reduction on Ensemble Architectures and the Solution Tridiagonal Systems of Equations.* Technical Report YALE/CSD/RR-339, Department of Computer Science, Yale University, October 1984.

[13] S.L. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Stat. Comp.*, 1986. Also available as Report YALEU/CSD/RR-436, November 1985.

[14] P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, C-22(8):786–792, 1973.

[15] F.T. Leighton. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks.* MIT Press, 1983.

[16] O.A. McBryan and E.F. Van de Velde. Hypercube algorithms and implementations. In *2nd SIAM Conference on Parallel Computing*, November 1985.

[17] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice Hall, 1977.

[18] Y. Saad and M.H. Schultz. *Data Communication in Hypercubes.* Technical Report YALEU/DCS/RR-428, Department of Computer Science, Yale University, October 1985.

[19] Y. Saad and M.H. Schultz. *Topological Properties of Hypercubes.* Technical Report YALEU/DCS/RR-389, Department of Computer Science, Yale University, June 1985.

[20] C.L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[21] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 263–277, ACM, 1981.

[22] A.Y. Wu. Embedding of tree networks. *Journal of Parallel and Distributed Computing*, 2(3):238–249, 1985.

# The Architecture of a Homogeneous Vector Supercomputer

John L. Gustafson, Stuart Hawkinson, and Ken Scott

Floating Point Systems, Inc.
Beaverton, Oregon, 97005

## Abstract

A new homogeneous computer architecture developed by FPS combines two fundamental techniques for high-speed computing: parallelism based on the binary *n*-cube interconnect, and pipelined vector arithmetic. The design makes extensive use of VLSI technology, resulting in a processing node that can be economically replicated. Processor nodes incorporate high-speed communications and control, vector-oriented floating-point arithmetic, and a novel dual-ported memory design. Each node is implemented on a single circuit board and can perform 64-bit floating-point arithmetic at a peak speed of 16 MFLOPS. Eight nodes are grouped together with a system node and disk support to form modules. These modules, housed in cabinet-sized packages, are capable of 128 MFLOPS peak performance and make up the smallest homogeneous units of larger systems. The new FPS system achieves a careful balance between high-speed communication and floating-point computation. This paper describes the new architecture in detail and explores some of the issues in developing effective software.

## Introduction

The quest for increased computational power in scientific computing and the limits of physical electronic devices have lead to the exploration of new architectures as alternatives to traditional monolithic designs [9,2]. *Multiprocessor* designs hold the promise of tremendous performance increases, provided the interconnection network can support the parallelism inherent in the computation. *Vector pipelines* provide significant performance increments, exploiting finer grained parallelism. Further advantage is gained by using *parallel functional units* to overlap address calculations with memory references, floating-point adds, and floating-point multiplies [1].

Large scientific applications are sometimes easily partitioned among processors using a shared memory, yet most are just as amenable to distributed memory designs [3,4]. Shared memory systems are expensive when scaled to large dimensions because of the rapid growth of the interconnection network; the distance from memory to the processing elements also degrades performance by increasing latency [8]. Large system configurations are most readily realized with distributed memory based on a limited form of interconnection, such as the pyramid or binary *n*-cube [5]. Memory latency can be greatly reduced when each processor has its own high-speed store. Moreover, the cost of switching and the time to route messages is much smaller on such statically configured systems. With this view, much current computer architecture research has focused on the use of ensembles of identical processors in homogeneous configurations that employ message passing over limited forms of static interconnects [7,8].

Floating Point Systems (FPS) has developed a *homogeneous computer*, the FPS T Series, based on the *binary n-cube interconnection* scheme. The individual nodes are 64-bit floating-point computers that combine vector arithmetic, dual-port memory, and fast communications links between nodes. The peak performance of these nodes is 16 MFLOPS. The FPS T Series is built from modules containing eight of these nodes connected to each other and to a system support ring. These modules, with an aggregate performance of 128 MFLOPS, may be combined to form even larger systems that promise orders of magnitude increases in computing speed per dollar over today's supercomputers.

## Processor Node Architecture

An individual processor element is called a *node*. It contains a control processor, floating-point arithmetic, dual-port memory, and communication links to other nodes. The FPS T Series design provides all of these functions on a single printed-circuit board. Each of the major elements of the node has been implemented with advanced, cost-effective VLSI technology, in contrast with more traditional bit-slice designs.

### Control

The ability to interpret and execute programs resides in the central control unit. The T Series control unit is a 32-bit CMOS microprocessor with the following functional features:

- 7.5 MIPS instruction rate
- Byte addressability (4 GByte address space)
- 2048 bytes of on-chip RAM (single processor cycle)
- 3-cycle minimum access time for off-chip memory
- Four bidirectional serial communications links
- Stack-oriented instruction set with variable operand sizes
- Two-level process priority and interrupt services

The control processor executes system and user applications code and it also serves to arrange vector operands to be sent to the vector arithmetic hardware. The control processor can execute integer arithmetic and gather/scatter operations in parallel with the vector unit, and it provides inter-node communications via the serial links.

0190-3918/86/0000/0649 $01.00 © 1986 IEEE

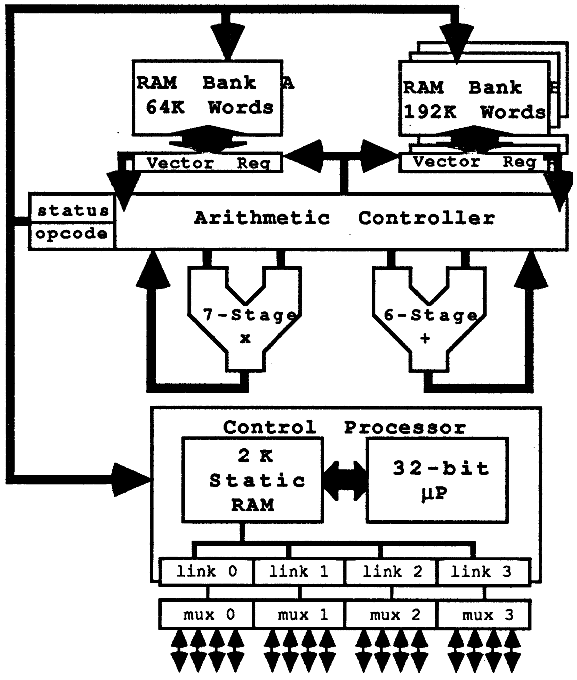


Figure 1. The FPS T Series processor.

All features of the microprocessor are directly accessed through a high-level language called *Occam*. Occam differs from languages like Pascal or C in that it directly provides for the execution of parallel, communicating processes, Channel commands can make direct data transfers between concurrent processes. A single process can be constructed from a collection by specifying *sequential*, *alternative*, or *parallel* execution of the constituent processes. This combination of program structure and integrated communication allows Occam to describe the control and data flow for virtually any scientific computing algorithm, and to control the high-level operation of the vector arithmetic unit (see below).

## Memory

An essential feature of a computer's architecture is its central memory, which supplies both instructions and operands to the processing units. The main memory of each FPS T Series node consists of 1 MByte of dual-ported dynamic RAM. The control processor and communications links read and write 32-bit words through a conventional random-access port, while the vector arithmetic unit makes use of a collection of *vector registers* closely coupled with main memory. A vector register can be loaded with an entire 1024-byte row of memory, in parallel (see Figure 1), in the same time that it would have taken to read or write a single 32-bit word. There is one parity bit for each byte in memory.

The control processor views the memory as a single bank of 256K words (32-bit). The vector arithmetic unit views memory as two banks of vectors, with 256 vectors in one bank and 768 vectors in the other, aligned on 1024-byte boundaries. Thus, for 32-bit operations, the vectors are 256 elements long, while for 64-bit operations, the vectors are 128 elements in length. The division of memory into two banks permits two inputs in parallel to the arithmetic unit on each cycle (125 ns). The output of the arithmetic unit shifts results into either or both banks. Hence, operations such as SAXPY, Vector Add, and Vector Multiply proceed at the full speed of the arithmetic components, without being limited by available memory bandwidth. This dual-bank memory organization allows the node to function without the need for auxiliary data registers or cache.

The control processor can access a 4-byte word in 400 ns. Its effective bandwidth is therefore

$$(4 \text{ bytes}) / (0.4\ \mu\text{s}) = 10 \text{ MB/s}$$

A primary use for the control processor is to gather operands into a contiguous vector, and scatter results back to random locations in memory. To move a 64-bit operand from one memory location to another requires two 32-bit reads and two 32-bit writes, which take a total of 1.6 µs. This is the gather-scatter time within a node. For 32-bit operands, it is 0.8 µs per element.

An entire row of data can be moved to or from a vector register in only 400 ns; this means that the effective bandwidth between memory and a vector register is

$$(1024 \text{ bytes}) / (0.4\ \mu\text{s}) = 2560 \text{ MB/s}$$

An application might make use of this extraordinary speed by moving data physically, rather than keeping linked lists of pointers to vectors, as for example, in pivoting rows of a matrix or sorting records.

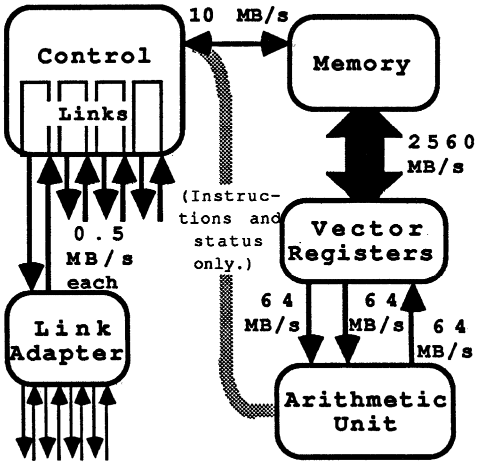


Figure 2. Processor bandwidths.

The vector registers each supply data to the arithmetic unit at a maximum rate of one 32-bit word every 62.5 ns, or one 64-bit word every 125 ns. The vector register bandwidth supports two vector inputs and one vector output every 125 ns in 64-bit mode. Thus, its bandwidth is

(3 words)x(8 bytes/word)/(0.125 μs) = 192 MB/s.

## Arithmetic

The ability to perform high-speed arithmetic is essential in scientific computing. The arithmetic hardware in the FPS T Series consists of a floating-point adder, floating-point multiplier, interconnection hardware, and some sequencing hardware. The adder and multiplier each can produce a 32- or 64-bit result every 125 ns, yielding peak performance of 16 MFLOPS per node. Floating-point operations are performed using the proposed IEEE floating-point standard format; however, gradual underflow is not supported. In 64-bit mode, the mantissa has approximately 15 decimal digits of precision and a dynamic range of roughly $10^{-308}$ to $10^{+308}$.

The arithmetic units operate in pipelined mode. The adder has a six-stage pipeline. It can perform floating-point addition and subtraction in 32- and 64-bit modes, comparisons and data conversions. The multiplier is five-stage in 32-bit mode and seven-stage in 64-bit mode. These pipeline lengths are appropriate for the vector access described above. Scalar operations can be efficiently performed by grouping like operations for level-order evaluation.

The arithmetic parts are supervised by a preprogrammed micro-sequencer, that implements a collection of vector arithmetic operations referred to as *vector forms*. The programmer only needs to describe the input and output vectors and the vector form desired. This frees the control processor for other tasks while vector operations are being executed. Scalars can be held in the input registers on each floating-point part, and outputs from the parts can be fed directly back as inputs to perform operations such as dot products and sums. This provides a wide range of useful vector forms without memory reference limitations. The complete arithmetic unit operates in parallel with the node control processor. The arithmetic unit only interrupts the controller when a vector operation has completed, or an error has occured.

## Communications

In a distributed computer system, communications channels are required for passing data between processors participating in a common computational process. The control processor of the FPS T Series contains drivers for four serial, bidirectional communications links each with a nomi- nal rate of 0.9375 Mbyte/s. Every 8-bit byte is sent with two synchronization bits and one stop bit, and requires two acknowledge bits from the receiver. This results in a maximum effective unidirectional bandwidth of over 0.5 MB/s per link. The total bandwidth of the four links is thus over 4 MB/s. With all links operating, the control processor performance is degraded only slightly. The links operate via DMA transfers with a startup time of about 5 μs.

Each link is multiplexed four ways to provide a total of 16 bidirectional sublinks per node. With software support, these sublinks divide the available bandwidth. Two sublinks are used for system communication, and two will often be utilized for mass storage and/or external I/O. This will typically leave 12 sublinks available for connection to other compute nodes.

A convenient way to interpret the relative bandwidths is with respect to the arithmetic processing time for 64-bit operations:

(ArithmeticTime):(GatherTime):(LinkTransferTime)
.125 μs          1.6 μs          16 μs

that are in the approximate ratios 1 : 13 : 130. Thus, a vector should enter into about 13 operations while gathering the next vector into an aligned, contiguous order. With this provision, the control processor can completely overlap the gather time with vector arithmetic, and the node can approach peak speed. Of course, if vectors are always aligned and elements contiguous, no such restriction applies. Similarly, roughly 100 operations should result from every 64-bit word that must be moved between nodes over a link.

## System Description

The T Series consists of a number of node processors connected as a binary *n*-cube. There are $2^n$ processors, with *n* connections per node. Numbering the processors from 0 to $2^{n-1}$, each processor is directly connected to all others whose numbers differ in only one binary digit. The binary *n*-cube can be mapped onto many important applications topologies, including meshes (up to dimension *n*), rings, cylinders, toroids, and even FFT butterfly connections of radix 2 [5,6]. Since the maximum number of connections between any two processors is *n*, long-range communication costs grow only as $O(\log_2 n)$.

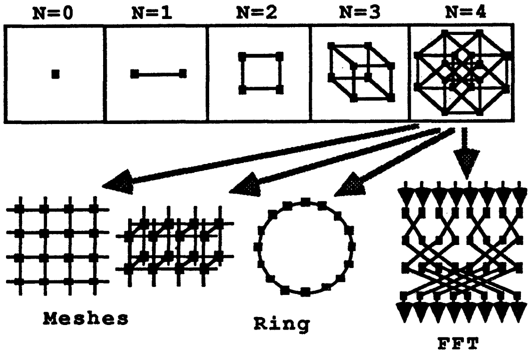


Figure 3. Binary *n*-Cube Mappings.

## Modules and System Ring

A processor node is constructed on a single etched circuit board. Eight nodes are combined with disk storage and a system board to form a *module*. Such a module has 128 MFLOPS peak floating-point performance, and 8 MB of user RAM. The local inter-node communications bandwidth is over 12 MB/s, while the system board can support 0.5 MB/s to an external connection.

The system board provides input/output and management functions. It is connected to the nodes by a thread of communications links that traverses the eight processor nodes. The system boards are directly connected by communications links to form a ***system ring*** that is independent of the binary *n*-cube network (connecting the processor nodes). The primary function of the system disk is to record memory "snapshots" which checkpoint computations for error recovery, and to backup snapshots from other modules. The user is able to specify the interval between snapshots. About 10 minutes provides a good compromise between time spent to record memory and interval between restart points. It takes about 15 seconds to take a snapshot, regardless of configuration.

The module requires three links for intramodule hypercube network communications, while the system board connections require two links from each processor node. This reduces remaining links to 11 for hypercube and external communications.

Two modules (16 nodes) form a *cabinet*, or 4-cube (a tesseract). A cabinet is modular and self-contained in a standard 19-inch rack-mounted assembly. With power supplies, system disks, and air cooling fans, it does not require any special "computer room" facilities. Larger systems are simply assembled from these units by interconnecting cables. Connections up to 40 feet can be made without special consideration.

## Larger Configurations

A four-cabinet (64-node) system has an aggregate peak speed of 1 GFLOP and total user memory of 64 MBytes. Eight system-disk units provide backup and restart capability. This configuration of the FPS T Series can be located in many laboratory and computing facilities. The air-cooled unit requires no special facilities beyond normal air conditioning, and the power requirements are supplied by typical 220 VAC services.

There are enough links per node to permit a 14-cube to be constructed as the largest T Series configuration. Using two links per node for external I/O and mass storage systems, a maximum-sized 12-cube consists of 4096 nodes arranged as 256 cabinets (4-cubes). Such a system has over 65 GFLOPS peak processing performance and 4 GBytes of primary RAM storage. Special facilities would be required to house the largest configurations.

Because the system is homogeneous, i.e., each module is identical and contains identical connections to other modules, programming is greatly simplified. This homogeneity also insures that the balance between computing speed, main storage, mass storage, and external I/O can be preserved as configurations become large. The specifications of any sized FPS T Series can be derived from the properties of the individual modules.

## Conclusion

The incorporation of high-speed vector processing into a homogeneous parallel architecture has resulted in a scientific computer with performance scalable over three orders of magnitude. The use of a dual-ported dynamic RAM achieves a new level of processor integration that eliminates the need for separate data registers or cache. Parallel floating-point adders and multipliers, accessed by standard vector operations, provide a close match to scientific computing algorithms.

With nodes organized into eight-processor modules, each with system disk and I/O services, the new architecture can be viewed as a truly homogeneous system. The FPS T Series, incorporating VLSI components to make the system both cost-effective and compact, provides a careful balance between processor speed, memory access, and interprocessor communications bandwidth.

## References

[1] Charlesworth, A. E., "An Approach to Scientific Array Processing: the Architectural Design of the AP-120B/FPS-164 Family." *IEEE Computer*, **14**, 9, (1981), 18-27.

[2] Charlesworth, A.E., and Gustafson, J.L., "Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer." *IEEE Computer*, **19**, 3 (1986).

[3] Fox, G.C. "Decomposition of Scientific Problems for Concurrent Processors." Technical Report CALT-68-986, Computer Science Dept., Caltech, 1983.

[4] Fox, G.C., and Otto, S.W., "Algorithms for Concurrent Processors." *Physics Today*, **37**, 5 (1984), 50-59.

[5] Pease, M.C., "The Indirect Binary *n*-Cube Microprocessor Array." *IEEE Transactions on Computers*, **C-26**, 5 (1977), 458-473.

[6] Saad, Y., and Schultz, M.H., "Topological Properties of Hypercubes." Technical Report YALEU/DCS/RR-389, Computer Science Dept., Yale Univ., 1985.

[7] Seitz, C.L., "Experiments with VLSI Ensemble Machines." *J.VLSI Comp. Systems*, **1**, (1984).

[8] Seitz, C.L., "The Cosmic Cube." *Comm. of the ACM*, **28**, 1 (1985), 22-33.

[9] Seitz, C.L., and Matisoo, J. "Engineering Limits on Computer Performance." *Physics Today*, **37**, 5 (1984), 38-45.

# Architecture of a Hypercube Supercomputer

John P. Hayes, Trevor N. Mudge, Quentin F. Stout

Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Dept.
University of Michigan
Ann Arbor, Michigan 48109

Stephen Colley, John Palmer

NCUBE
1815 N. W. 169th Place, Suite 2030
Beaverton, Oregon 97006

**Abstract:** The implementation of massively parallel computers based on hypercube architectures is discussed in this paper. It is argued that such machines offer an alternative to traditional supercomputers at far lower cost. The rationale for using hypercube machines for supercomputing applications is examined, including cost, node performance, communication speed, packaging, reliability, and programming requirements. These issues are illustrated by a recently introduced commercial hypercube supercomputer, the NCUBE/ten. The major design decisions underlying the NCUBE/ten's implementation technology, system architecture and operating system are described.

## 1. Introduction

A hypercube or (binary) $n$-cube computer is a multiprocessor characterized by the presence of $N = 2^n$ processors interconnected as an $n$-dimensional binary cube. Each processor $P_i$ forms a node (vertex) of the cube and is a self-contained computer with its own CPU and local main memory. $P_i$ has direct communication paths to $n$ other processors (its neighbors), which correspond to the edges of the cube that are connected directly to $P_i$. $2^n$ distinct $n$-bit binary addresses or labels may be assigned to the processors so that each processor's address differs from that of each of its $n$ neighbors in exactly one bit position. Figure 1 illustrates the hypercube topology for $n < 4$; note that a zero-dimensional hypercube is a conventional SISD computer. An $n$-dimensional hypercube $Q_n$ for $n \geq 2$ can be defined recursively in terms of the graph product operation $\times$ as follows [4], where $K_2 = Q_1$ is the complete 2-node graph:

$$Q_n = K_2 \times Q_{n-1}$$

As illustrated by Fig. 1, $Q_n$ is composed of two copies of $Q_{n-1}$. Every node $P_{0x}$ in one copy of $Q_{n-1}$ is the neighbor of a node $P_{1x}$ in the other copy.

For some time, it has been known that the hypercube structure has a number of features which make it a useful architecture for parallel computation. For example, meshes of all dimensions and trees can be embedded into a hypercube so that neighboring nodes are mapped to neighbors in the hypercube. The communication structures used in the Fast Fourier Transform and Bitonic Sort algorithm can similarly be embedded into the hypercube. Since a great many scientific applications use mesh, tree, FFT, or sorting interconnection structures, the hypercube is a good candidate for a general-purpose parallel architecture. Even for problems with less regular communication patterns, the fact that the hypercube has a maximal internode distance (the graph diameter) of $n = \log_2 N$ means any two nodes can communicate fairly rapidly. This diameter is larger than the unit diameter of a complete graph $K_N$, but is achieved with nodes having only degree or fanout of $\log_2 N$, as opposed to the $N - 1$ degree of nodes in $K_N$. Other standard architectures with small degree, such as meshes, trees, or bus systems, either have a large diameter ($N^{1/2}$ for a 2-dimensional mesh) or a resource which becomes a bottleneck in many applications because too much communication must pass through it (as occurs at the apex of a tree, or at a large shared bus). Thus, from general topological arguments it can be concluded that hypercube architectures offer a good balance between node connectivity, communication diameter, algorithm embeddability, and programming ease. This balance makes them suitable for an unusually wide class of computational problems.

Based on various considerations of the foregoing kind, proposals to build large hypercube computers have been made for more than twenty years. In 1962, Squire and Palais at the University of Michigan, motivated by the hypercube's rich interconnection geometry and programming ease, carried out a detailed paper design of a hypercube computer [13,14]. They estimated that a 4096-node (12-dimensional) version of their machine would require about 20 times as many components as the IBM Stretch, one of the largest and most complex contemporary computers. Around 1975 IMS Associates, an early manufacturer of personal computers, announced a 256-node commercial hypercube based on the Intel 8080 microprocessor, but its design details were not published and the machine was never produced. In 1977, Sullivan et al. presented a thorough analysis of hypercube architectures, and a proposal to build a large hypercube

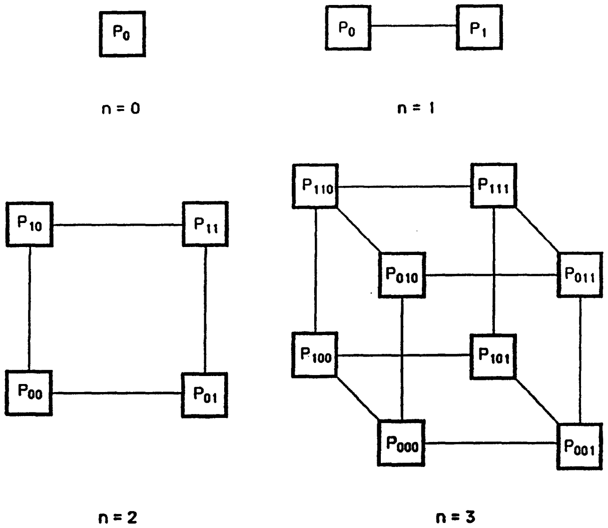


**Fig. 1.** $n$-dimensional hypercube for $n$=0,1,2,3.

0190-3918/86/0000/0653 $01.00 © 1986 IEEE

called CHOPP (Columbia Homogeneous Parallel Processor) containing up to a million processors [15,16]. In the same year, Pease published a study of the "indirect" binary *n*-cube architecture, in which a multistage interconnection network of the omega type is suggested for implementing the hypercube topology [8]. A number of other interesting architectures closely related to the hypercube have also been proposed, for example, the cube-connected-cycles structure [11].

It is clear that the early hypercube designs were impractical because of the the large number of components (logic and memory elements) they required using the then available circuit technologies. The situation began to change rapidly in the early 1980's as advances in VLSI technology allowed powerful 16/32-bit microprocessors to be implemented on a single IC chip, and RAM densities moved into the $10^5$–$10^6$ bits/chip range. A working hypercube computer was not demonstrated until the completion in 1983 of the first 64-node Cosmic Cube at Caltech [12]. For the hypercube node processor, it uses a single-board microcomputer containing the Intel 8086 16-bit microprocessor and the 8087 floating-point co-processor. Since then, Caltech researchers have built several similar hypercubes, and successfully applied them to numerous scientific applications, often obtaining impressive performance improvements over SISD machines of comparable cost [3].

Influenced primarily by the Caltech work, a number of commercial hypercubes have been developed since 1983. In July 1985, Intel delivered the first production hypercube, the 128-node iPSC (Intel Personal Supercomputer) which has a 16-bit 80286/287 CPU as its node processor. Assuming a peak performance of 0.07 MFLOPS per node, the 128-node iPSC has a potential throughput of about 8 MFLOPS, far below that of a traditional vector supercomputer such as the Cray-1 (160 MFLOPS). Other commercial hypercubes were also introduced in 1985 by Ametek Inc. and NCUBE Corp. The Ametek System/14 hypercube can have up to 256 nodes, which employ an 80286/287-based CPU similar to that of the iPSC, with the addition of an 80186 processor for communication management. The NCUBE/ten can accommodate up to 1024 nodes, each based on a VAX-like 32-bit custom processor with a peak performance of 0.5 MFLOPS. Thus a fully configured NCUBE system has a throughput potential of around 500 MFLOPS. This high performance level is supported by extremely fast communication rates (both input/output and node-to-node) making the NCUBE/ten a true supercomputer. NCUBE machines have been installed at several beta test sites, including the University of Michigan, since early 1985, and have been in general production since December 1985. Several other hypercube-style machines with supercomputing potential are presently under development, including the Caltech/JPL Mark III [9] and the Connection Machine [5]. Much faster successors to the current commercial hypercubes can also be expected to appear over the next few years. Because of the effort being devoted to the development of hardware and software for these machines, and their relatively low cost, hypercube supercomputers seem likely to provide an increasingly attractive alternative to conventional pipelined supercomputers for many applications.

This paper explores the architectural and technological issues influencing the design of supercomputing hypercubes, with the NCUBE/ten serving as an example. Particular attention is devoted to the influence of component packaging, reliability, communication speed, and the operating system environment on the system implementation. Section 2 discusses the general design requirements of hypercube supercomputers, while the specific design decisions made for the NCUBE/ten are covered in Sec. 3. Software issues are discussed in Sec. 4.

## 2. General Design Issues

Supercomputing performance requires extremely high integer and floating-point execution rates, as well as extremely high I/O throughput. Very large primary (RAM) and secondary (disk) memory spaces are also usually required. For the principal user base of scientific programmers, the programming environment needs to provide FORTRAN, and a powerful operating system such as UNIX. Low cost and high reliability imply minimizing the component count at all levels, particularly the numbers of chips and boards used. Some degree of fault tolerance is also very desirable. Since a very large amount of RAM storage is needed, memory fault detection and correction via an error-correcting code (ECC) is an important consideration, despite the fact that it increases the chip count. Reliability is increased, and operating cost decreased, by employing an air-cooled configuration suitable for a standard office environment. Based on an examination of various existing computer systems, it can be concluded that the air cooling limits the machine complexity to under 50,000 chips. Off-the-shelf parts decrease costs and usually increase reliability. If custom chips are needed, then computer manufacturers who rely on outside suppliers should use conservative design rules which will be accepted by multiple silicon foundries. In large-scale numerical calculations, the possibility of large error accumulation forces the individual calculations to be as accurate as possible. Numerical accuracy can be increased by adhering to the IEEE 754 floating-point standard and providing double-precision floating-point operations.

A key decision in the design of a parallel computer is the choice of the interconnection network to be used. Multistage interconnection networks have been advocated as simplifying the programming process by providing a global shared memory, but it did not seem possible to build a large multistage network using the technology available in 1983 without suffering significant delay in passing information through the network. Since this strongly affects performance, it was felt that to achieve supercomputer performance it would be necessary to use a direct connection network with local memory at every node. Many direct interconnection schemes have been analyzed and implemented but, as discussed in Sec. 1, the hypercube structure has a number of inherent advantages. The ease with which efficient application programs were developed for the hypercubes at Caltech has also shown the hypercube to be superior to alternative architectures such as meshes or trees. The neighbor-to-neighbor links of the hypercube provide almost the same communication capabilities as a complete graph, while using nodes with only a logarithmic degree. The achievable degree is constrained by a variety of packaging considerations, but with current technology it is possible to build hypercubes with thousands of nodes. In contrast, a complete graph connection of a few tens of nodes may not be possible.

There are additional features of the hypercube that are particularly useful in designing a supercomputer, but have not been previously exploited. For example, the hypercube is homogeneous in that all nodes look the same, hence it is natural to attach an I/O channel to each node. This provides the potential of extremely high system I/O rates. Also, since there are numerous ways to divide a hypercube into subcubes, it is easy to support multiprocessing where each user has a dedicated subcube. These subcubes can be allocated so that all processor-to-processor and I/O communications occur without using processors or communications lines in other subcubes. Further, by writing programs in which the size of the subcube is a user-defined parameter, it is possible to develop programs in small subcubes and then do production runs in larger subcubes. This partitionability also makes it easier to tolerate faults, since the operating system can allocate subcubes which avoid faulty processors or faulty communication lines.

As discussed in Sec. 1, technology developments have enabled a hypercube computer to be built reliably with a large number of processors. A fine-grained supercomputer architecture, i.e., one with a large number (over 1000) of very simple processors, has a high ratio of communication to computation. The Connection Machine is an example of a fine-grained hypercube-class computer, but its suitability for scientific computations is unclear. A very coarse-grained architecture with, say, 10 to 100 large and fast processors, requires that the nodes achieve extremely high performance. For example, to achieve $10^9$ instructions/sec with 10 processors requires processors capable of $10^8$ instructions/sec. The Caltech/JPL Mark III will be an example of a coarse-grained hypercube. It was felt by the designers of this machine that achieving $10^9$ instructions/sec is best done with 1000 processors running at $10^6$ instructions/sec.

Experience with the Caltech machines has demonstrated that a medium-grain MIMD hypercube architecture can attain high efficiency on a variety of scientific problems, with a tolerable amount of revision of serial code and algorithms [3]. This can be contrasted with the much greater amount of program and algorithm redesign required of users of fine-grained SIMD machines such as the MPP [10]. MIMD machines require each node processor to perform instruction fetching, decoding, and other functions that are not performed by SIMD nodes. Distributed-memory MIMD machines must supply a program to each node, so that MIMD machines may pay a large penalty in chip area and chip count. In general, for the same chip area and number of chips one can build more SIMD processors and have a greater potential system throughput; however, the gain in programming simplicity obtained by using an MIMD machine more than compensates for this, except for a narrow range of applications in which almost any penalty can be tolerated if it yields the required speed. Furthermore, MIMD machines can accommodate multiple independent users, while SIMD machines cannot.

Since there may be hundreds or thousands of nodes in a hypercube supercomputer, their chip count is the most significant component of the total system chip count. Using the densest possible memory chips available is the key factor in decreasing the number of chips. The NCUBE/ten, for example, uses 256K DRAM chips to implement the local memories of the hypercube nodes. The next most significant reduction in chip count can be achieved by putting all node functions onto a single chip. This implies that the processor chip must perform all communication, memory management, floating-point operations, and other data-processing functions. Unlike RAMs, there is not yet widespread market pressure to produce standard processor chips of this type, consequently, they are not available off the shelf. In 1983, when design of the NCUBE/ten started, the only way to achieve supercomputer performance with a one-chip node processor was to undertake the risky step of custom-designing such a complex chip. INMOS made a similar decision with the Transputer processor chip, with the important difference that the initial version of the Transputer does not provide floating-point operations, and has four rather than eleven I/O channels [6]. The performance and functionality demands on the NCUBE processor chip are quite severe, and numerous tradeoffs were needed to enable it to be built with current technology.

## 3. NCUBE/ten Architecture

The overall goal of the NCUBE designers was to use massive parallelism to build an inexpensive and reliable range of software-compatible machines achieving supercomputer performance at the high end. The largest model in the series, the NCUBE/ten, is a 10-dimensional hypercube containing 1024 powerful 32-bit processors of custom design, each with a 128-Kbyte local memory. Up to eight front-end host processors are used to manage I/O operations under control of a multiuser UNIX-based operating system. An unusually high level of system integration is employed that allows 64 processors with their memories and interconnections to be placed on a single printed-circuit board. A maximum-sized NCUBE/ten system is composed of 16 processor and 8 I/O boards (including host processors) and is housed in a small air-cooled enclosure.

The NCUBE node processor provides the functions of a 32-bit supermini-class CPU, including a full floating-point instruction set, and all the logic needed for memory management and interprocessor communication on a single VLSI chip; see Fig. 2. The design of the processor was started by NCUBE in 1983, constrained by the desire to use conservative design rules acceptable to several silicon foundries. The chip was designed using 2 $\mu$m (approx.) nMOS design rules, and contains about 160,000 transistors. It is housed in a pin-grid-array package with 68 pins. Combined with six 256K-bit DRAM chips (each of which is organized as 64K x 4 bits), an entire NCUBE/ten node requires only seven chips. Because of this, a 6-dimensional hypercube with 64 nodes and 8 Mbytes of memory can be packed into a single 16″ X 22″ board, a photograph of which appears in Fig. 3. The backplane connections are rather formidable since each node has off-board bidirectional channels to four more processors of the hypercube, plus one bidirectional channel to an I/O board, resulting in 640 backplane connections just for communication channels.

The instruction set of the NCUBE/ten is conventional and quite orthogonal, being similar to the VAX instruction set without the latter's 3-address addressing modes [7]. There are three main classes of information: addresses (unsigned integers), integers and floating-point numbers (reals). Addresses are 32 bits long, but the current node implementation only supports a 17-bit physical address space. Integers can be 8, 16 or 32 bits long. Floating-point numbers can contain either 32 or 64 bits, and conform to the IEEE 754 floating-point standard. There are 16 general-purpose registers of 32 bits each. A variety of addressing modes are available, including literal (immediate), register

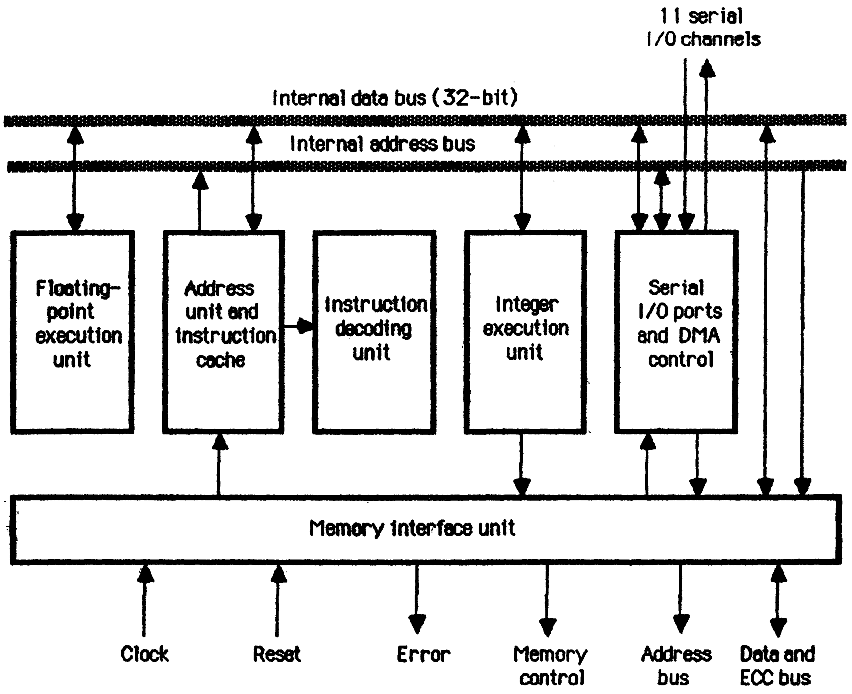

11 serial
I/O channels
Internal data bus (32-bit)
Internal address bus
Floating-point execution unit
Address unit and instruction cache
Instruction decoding unit
Integer execution unit
Serial I/O ports and DMA control
Memory interface unit
Clock
Reset
Error
Memory control
Address bus
Data and ECC bus

**Fig. 2.** Organization of the NCUBE processor chip.

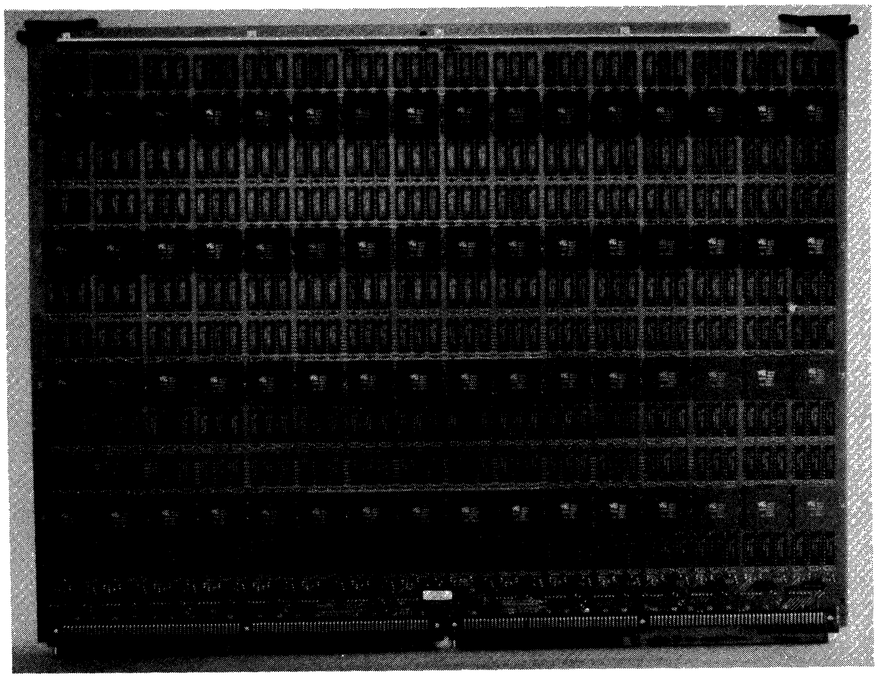


**Fig. 3.** The 64-node NCUBE processor board.

direct, autodecrement/increment, autostride, offset, direct, indirect, and push/pop. The instruction set contains a full complement of logical, shift, jump, and arithmetic operations (including square root). One instruction of particular use in hypercube routing is Find First One, which finds the bit position of the first 1 in a word, via a right-to-left scan. Using a 10-MHz clock, nonarithmetic instructions can be executed at about 2 MIPS, with single-precision floating-point operating at 0.5 MFLOPS, and double-precision at 0.3 MFLOPS. (These performance figures assume that register-to-register operations predominate.) A 32-byte instruction cache allows loops of up to 16 bytes to be executed directly from the cache. The node processor has a vectored interrupt facility, and generates different interrupts to indicate program exceptions such as numerical overflow or address faults, software debugging commands such as breakpoint and trace, I/O signals such as input ready, and hardware errors such as correctable or uncorrectable memory errors.

Pin and silicon space limitations forced a number of design compromises in the selection of the width of various system data paths. The node memory supplies data in 16-bit halfwords, plus an extra byte containing ECC check bits. The processor performs single-error correction and double-error detection (SECDED) on all memory words, generating an interrupt in case of an error. This is an example of a situation where the pin limitations affect performance, for it requires two memory fetches to obtain a full 32-bit word. It also increases the number of memory chips required, since the SECDED code used for 32-bit data could be supplied by five RAM chips organized as 32K × 8 bits, if such chips were available.

Communication with other nodes is performed via asynchronous DMA operations over 22 bit-serial I/O lines. The I/O lines are paired into 11 bidirectional channels, which permit formation of a 10-dimensional hypercube, and also allow one connection to an I/O board. Each node-to-node channel operates at 10 MHz with parity check, yielding a data transfer rate of about 1 Mbyte/sec per channel in each direction. A channel has two 32-bit write-only registers associated with it: an address register for the message buffer location in the node RAM, and a count register indicating the number of bytes left to send or receive. There is also a ready flag and an interrupt enable flag for each channel. Once a send or receive operation has been initiated by a processor checking its flags and setting the appropriate registers, the processor can continue with other operations while the DMA channel completes the internode communication operation. Interrupts can be used to signal when a channel is ready for a new operation. For general applications, this requires less processor overhead than would occur in a polling communication protocol. An interrupt is also generated if there is a channel overrun, which can occur only on an input operation if more than 9 channels are transmitting data into the node. To reduce DMA activity, a broadcasting feature is supported which transmits the same data word along an arbitrary set of output channels in a single DMA operation.

The NCUBE/ten's I/O boards provide the connections between the hypercube and the external world. Each system must have at least one host board, and may have as many as eight. The host board uses an Intel 80286 to run the operating system, and has 4 Mbytes of RAM used as a shared memory by the various processors on the host board. It has support for a variety of different peripherals, including eight ASCII-standard terminals, four SMD disks (which can currently be as large as 500 Mbytes), and three Intel iSBX connectors that can accept daughter boards for functions such as graphics control or networking. Miscellaneous functions found on the host board include a real-time clock, and temperature sensors for automatic shutdown on overheating. Besides the host board, other I/O boards currently available are a graphics board with a 2K × 1K × 8-bit frame buffer, an intersystem board to connect two NCUBE systems, and an open system board with about 75% of the board left for custom design.

A distinguishing feature of the I/O boards in the NCUBE/ten is the fact that each has 128 bidirectional channels directly connected to a subcube of the hypercube; see Fig. 4. This permits extremely high I/O data-transfer rates into the hypercube enabling, for example, a single I/O board to transfer 1024 × 1024 × 8-bit images at video rates (30 frames per second). To accomplish this, each I/O board contains 16 NCUBE processors chips, each of which serves as an I/O processor and is connected to eight nodes in the main hypercube. Like the hypercube node processors, an I/O processor has a 128-Kbyte RAM which occupies a fixed slot in the 80286 host's 4-Mbyte memory space. An input operation from the outside world, e.g., a disk read, is performed by first transferring the input data to the host's 4-Mbyte memory. Then the data is transferred through the DMA channels of the I/O processors directly to the target hypercube nodes. Output operations are handled in a similar fashion. In addition to sharing access to the host's memory, the 16 I/O processors on each I/O board are interconnected as two disjoint 3-dimensional cubes, (this disjointness occurs because each node has only 11 bidirectional channels.) Note that in a maximum-configuration NCUBE/ten system, the hypercube nodes do not have to redistribute I/O data to other nodes. This is not the case in smaller NCUBE systems where fewer channels are available for external I/O operations.

An NCUBE system has from one to eight I/O boards (at least one of which must be a host board), from one to 16 processor boards, and attached peripheral devices. All the I/O and processor boards of a fully configured system, along with their fans and power supplies, fit into a single enclosure that is less than 3′ on each side. A full-sized system dissipates about 8 kW, and can be housed in a standard air-conditioned environment. A peripheral enclosure is about 3′ × 2′ × 3′ and contains a 65-Mbyte cartridge tape drive and up to four disk drives. A minimal standalone NCUBE system consists of one host board and one processor board containing a 6-dimensional hypercube, and can handle up to 8 user terminals. By adding a second processor board, one obtains a 7-dimensional hypercube. Since the operating system can allocate subcubes of arbitrary size, it is possible to have a number of processor boards which do not form a complete hypercube. For example, three boards provide a 7-dimensional and a 6-dimensional hypercube, which could also be allocated as three 6-dimensional hypercubes or as numerous smaller hypercubes. A full-sized system (Fig. 4) contains a 10-dimensional hypercube (which explains the "ten" in NCUBE/ten). The 1024 processors of such a system together have a potential instruction execution rate of about 2 billion instructions/second, or about 500 MFLOPS. The total amount of memory in the nodes is 128 Mbytes. If

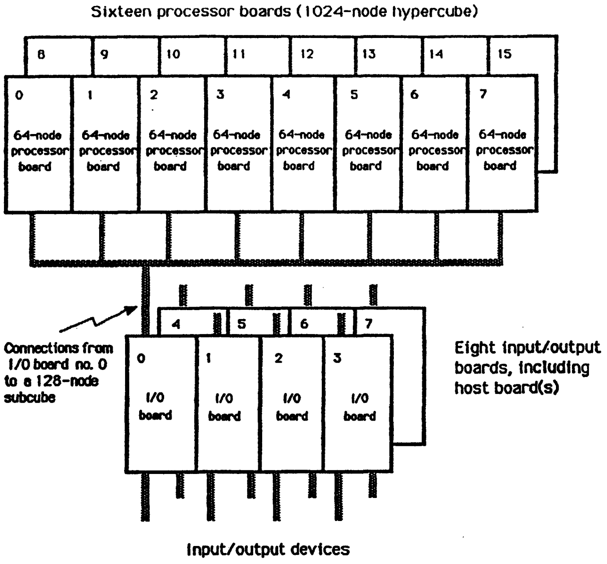

Sixteen processor boards (1024-node hypercube)
8
9
10
11
12
13
14
15
0
1
2
3
4
5
6
7
64-node processor board
Connections from I/O board no. 0 to a 128-node subcube
4
5
6
7
0
1
2
3
I/O board
Eight input/output boards, including host board(s)
Input/output devices

**Fig. 4.** Maximum-configuration NCUBE/ten system.

all of the I/O boards are host boards, it is possible to support 64 terminals, and provide as many as 16 billion bytes of storage. A host board can provide input or output at up to 90 Mbytes/sec, giving a system input or output rate of about 720 Mbytes/sec.

Figure 5 summarizes the results of some performance experiments designed by D. Winsor at the University of Michigan, which compare the NCUBE node processor to two other representative CPU's with floating-point hardware: the Intel 80286 (the NCUBE host processor served for this) and Digital Equipment Corp.'s VAX-11/780. The measurements were made with the NCUBE node and host processors running under 8 MHz clocks. Extrapolated figures for the 10 MHz version of the NCUBE node processor now nearing production are also given, assuming no wait states. Two widely used synthetic benchmark programs were employed in this study, the Dhrystone and the Whetstone codes [2,18]. The Dhrystone benchmark is intended to represent typical system programming applications and contains no floating-point or vectorizable code. The original Dhrystone Ada code [18] was translated into a FORTRAN 77 version with 32-bit integer arithmetic that attempts to preserve as much of the original program structure as possible. This entailed changes such as replacing Ada records by FORTRAN arrays which produce a substantial performance degradation compared to Dhrystone benchmarks in Ada, Pascal or C. However, any such degradation here appears to apply uniformly to all processors considered, since all were given the same FORTRAN source code and used very similar FORTRAN compilers. The Whetstone benchmark, which aims to represent scientific programs with many floating-point operations, was used in a single-precision FORTRAN 77 version that closely resembles the original ALGOL code [2]. The Dhrystone results in Fig. 5 are reported in "Dhrystones per second," each of which corresponds roughly to one hundred FORTRAN statements executed per second. The Whetstone figures represent the number of hypothetical Whetstone instructions executed per second. It can be concluded from the data of Fig. 5 that the NCUBE node processor is quite fast and fully meets its performance targets cited above.

### 4. System Software

The emergence of several commercial hypercube computer has demonstrated the feasibility of constructing low-cost massively parallel machines. The focus of research can now be expected to shift to the issue of how these machines can be programmed effectively. Indeed, the recent report on the Supercomputing Research Center concludes that the absence of appropriate parallel programming languages and software tools is the single biggest impediment to the successful use of parallel machines [1]. The operating system is also a major design issue, since memory management and interprocessor communication are critical to the functioning of the programming languages. Three software issues need to be considered. The first is the operating system that is used for developing application programs for the hypercube. The second is the operating system that provides run-time support for application programs running on the hypercube nodes. The third is the set of application languages to be used.

| Processor | FORTRAN Dhrystones/sec | FORTRAN Whetstones/sec |
|---|---|---|
| NCUBE node processor at 8 MHz | 999 | 381,000 |
| NCUBE node processor at 10 MHz (est.) | 1,249 | 476,000 |
| Intel 80286 (NCUBE host) at 8 MHz with 80287 floating-point co-processor | 510 | 101,000 |
| DEC VAX-11/780 with floating-point accelerator | 741 | 426,000 |

**Fig. 5.** Summary of processor benchmark results.

An attractive choice for a development operating system that provides the kind of environment associated with a "programmer's workbench" is UNIX. Unfortunately, there are two different versions of UNIX (System V and bsd 4.2), and a large number of lesser-known variants. This leaves the system designer with something of a dilemma: on the one hand UNIX offers a proven development environment that is widely known; on the other hand a UNIX standard has yet to emerge. The solution chosen by NCUBE was to develop a UNIX-like operating system called AXIS [7] that embodies the features common to the major UNIX dialects. Subsequent change or additions can be readily made to AXIS when a true UNIX standard is agreed upon. There are two features of AXIS that we shall elaborate on here because they are pertinent to the management of a very large hypercube. The first is the ability to share files, and the second is the way in which the main cube array is managed.

AXIS runs on the 80286 host processor that acts as the CPU for each I/O board. (Recall that up to eight I/O subsystems can be accommodated in a 1024-processor NCUBE/ten). It provides the large number of utilities for editing, debugging and file management that one has come to expect in a UNIX-like operating system. Consistent with the UNIX philosophy, the file system is the most prominent feature of AXIS, and almost all of the system resources are treated as files. Massively parallel systems require high I/O bandwidth if they are to be useful for applications that are not simply computation-intensive. This problem of I/O management was not foreseen in the earlier generation of massively parallel machines, and has proved to be a great limitation [10]. The ability to incorporate up to eight I/O subsystems in the NCUBE/ten is intended to avoid this problem. However, it introduces the potential for eight separate file systems. To avoid this, AXIS provides the capability to organize the eight file systems as one distributed file system; AXIS further allows complete systems to be networked through iSBX connections to provide a single multiuser file system. The principal mechanism for doing this is the device directory pointer (ddir). ddirs are items that can be placed in a file directory. Instead of representing a file name, they are pointers to disk drives. Each disk drive has a unique device identifier, which includes a *system number*, an I/O *board number*, and a *drive number*. Within a disk drive, the files are organized as a typical UNIX tree. ddirs can be placed in the root directory of a disk to point to the root of the other drives. This permits file sharing across all physically connected NCUBE systems. Figure 6 illustrates how the directory structure for two host boards, each having two physical disk drives, might be organized, if logical disk0 and disk4 are shared systemwide. Typically, the device directory of a physical disk0 (called "//") will contain the following:

1. The name of another directory which acts as the real root of the file structures on disk0 ("cb0" on host 0)
2. ddir's for the other drives connected to the same host
3. ddir's of disk directories on any other host board in the system
4. ddir's for disks on any other physically connected NCUBE system. Since *system number* is one of the components of a ddir, it can refer to disks on other NCUBE systems.

AXIS manages a hypercube of node processors as a device, which is simply one type of file. It can be opened, closed, written to, and read from as if it were a normal file. AXIS permits users to allocate subcubes that have the appropriate size for their application. Thus, one or two large problems, or several small problems may share the hypercube. This flexibility greatly increases the system efficiency, and gives a hypercube supercomputer a significant advantage over conventional supercomputers. Partitioning the main hypercube into subcubes is simplified by the fact that each subcube is protected from access by any other subcube.

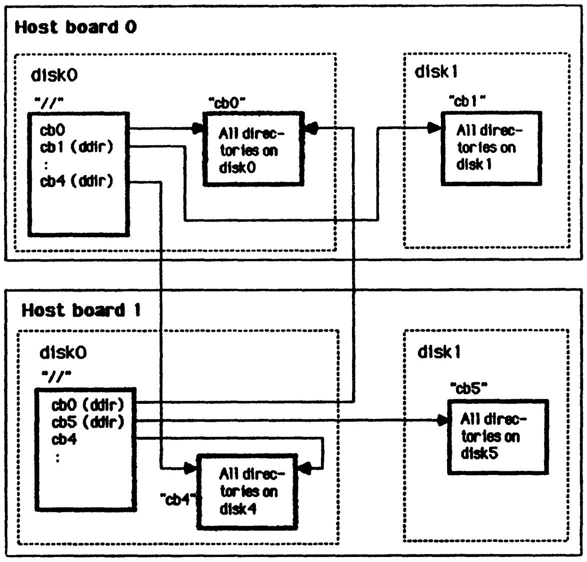


**Fig. 6.** Distributed files on the NCUBE/ten.

VERTEX, the operating system for the hypercube node processors, is a small nucleus (less than 4K bytes) that is resident in each of the NCUBE/ten nodes. Its primary function is to provide communication between the nodes, in the form of send and receive functions that transfer messages between any two nodes in the hypercube. The node processor has instructions that are used as primitives in the VERTEX communication calls, **nwrite** and **nread,** which implement the internode send and receive functions, respectively. The messages transferred by **nwrite** and **nread** are arrays of bytes having four associated attributes: source, destination, length and type. The first two attributes are numbers in the range 0 to 1023, and indicate the logical nodes being used for source and destination. The length attribute is the number of bytes in the message; messages as long as 64K bytes are supported. Finally, the type attribute can be used to distinguish messages, and so permit their selective reception at a destination node.

The subroutine **nwrite** may be represented as

**nwrite** *(length, messages, dest, type, status, error)*

where *length* is length of the outgoing message in bytes, *message* is the name of the buffer from which the message is to be taken, *dest* is the logical number of the node in hypercube that is to receive the message, *type* is the type number of the message, *status* indicates when the message has left the buffer, i.e. when the buffer is reusable, and *error* is an error code. Message transmission breaks the message into packets of 512 bytes (or some other user-defined size), and sends them to the destination node using the following routing algorithm. Assume that in an $n$-dimensional cube, the logical number of the source node is $s_n s_{n-1} \ldots s_2 s_1$ and that of the destination is $d_n d_{n-1} \ldots d_2 d_1$. The bit-wise exclusive-or $x_n x_{n-1} \ldots x_2 x_1$, of the two numbers is formed as follows: $x_i = s_i \oplus d_i$ for $i = 1, \ldots, n$. The values of the $x_i$'s are used to control the routing process. Those values of $i$ for which $x_i = 1$ indicate the dimensions that must be traversed to transfer a message from source to destination. The routing algorithm was chosen for its simplicity; however, as noted by Valiant [17], there is a potential for congestion in some situations. He defines an alternative routing algorithm that avoids congestion by routing each message to a randomly chosen node; from there the message is forwarded to its originally intended destination. The randomization assures that message congestion at nodes will be dispersed. Unfortunately, Valiant's router does not perform as well as the straightforward algorithm in many routine parallel processing tasks, and its more complex implementation requirements discouraged use of it in the initial NCUBE/ten design. Future insights into the behavior of parallel algorithms may change this, however.

In addition to determining the routing path, VERTEX must perform the store-and-forward function at each node along the path. At the destination node the message is placed in a queue that is allocated from a heap of 20K bytes. The receive function, which can be represented by

**nread** *(length, message, source, type, status, error)*

looks for the first message from *source* of type *type* in the input queue, and copies it to buffer *message*. Don't care conditions are indicated for *source* or *type* by setting these parameters to -1. This allows the next message from a particular source to be received regardless of type, it allows the next message of a particular type to be received from any source, and it allows the next message of any type from any source to be received. Messages with negative types other than -1 are system messages for VERTEX and are used for process control at a node, e.g., for node program debugging. In summary, the calls **nwrite** and **nread** provide a fast internode message communication mechanism. The main contributers to this speed are the machine instructions provided explicitly for internode communication, and the fact that messages enter nodes through DMA channels. Measurement of the internode communication performance of the NCUBE system is presently under way.

The current NCUBE/ten application languages, apart from the node and host assembly languages, are FORTRAN 77 and C. The choice of FORTRAN 77 and C was made because the computer is targeted for a user community interested primarily in scientific problems; this group has traditionally programmed in FORTRAN. Compilers for other languages, including Occam, are presently being developed. The programming model adopted for the initial set of languages (FORTRAN and C) is a simple extension of the conventional uniprocessor model. Each node is treated as a separate processor. No symbols are shared between nodes: the naming scope is contained within a node. Values of variables are shared by calls to the VERTEX subroutines **nwrite and nread.**

## 5. Conclusion

Hypercube architectures are well suited to implementing massively parallel supercomputers, given the constraints imposed by current technology. They offer an unusually good combination of high node connectivity, software flexibility, and system reliability. The NCUBE/ten is an example of a new generation of low-cost and compact hypercube machines with the capability of supercomputing performance. Unlike earlier machines, it exploits the inherent homogeneity of the hypercube to provide a multiuser UNIX-like programming environment, along with support for extremely high I/O data-transmission rates.

## Acknowledgement

The University of Michigan portion of the work reported here was supported in part by the Office of Naval Research under Contract No. N00014 85 K 0531.

## References

[1] Anon: Report of the Summer Workshop on Parallel Algorithms and Architectures for the Supercomputing Research Center, Aug. 1985.

[2] H.J. Curnow and B.A. Weichman: "A synthetic benchmark," *Computer Jour.* vol. 19, pp. 43-49, Feb. 1976.

[3] G. Fox: "The performance of the Caltech hypercube in scientific calculations," Caltech Report CALT-68-1298, April 1985.

[4] F. Harary: *Graph Theory,* Addison Wesley, Reading, Mass., 1969.

[5] W.D. Hillis: *The Connection Machine,* Cambridge, Mass., MIT Press, 1985.

[6] INMOS Corp: *Transputer Reference Manual,* Colorado Springs, 1985.

[7] NCUBE Corp.: *NCUBE Handbook,* version 1.0, Beaverton, Ore., Apr. 1986.

[8] M.C. Pease: "The indirect binary $n$-cube microprocessor array," *IEEE Trans. on Computers,* vol. C-26, pp. 458-473, May 1977.

[9] J.C. Peterson et al.: "The Mark III hypercube-ensemble concurrent processor," *Proc. Int'l Conf. on Parallel Processing,* pp. 71-73, Aug. 1985.

[10] J.P. Potter (ed): *The Massively Parallel Processor,* Cambridge, Mass., MIT Press, 1985.

[11] F.P. Preparata and J. Vuillemin: "The cube-connected cycles: a versatile network for parallel computation," *Communic. of the ACM,* vol. 24, pp. 300-309, May 1981.

[12] C.L. Seitz: "The Cosmic Cube," *Communic. of the ACM,* vol. 28, pp.22-33, Jan. 1985.

[13] J.S. Squire and S.M. Palais: "Physical and logical design of a highly parallel computer," Tech. Note, Dept. of Elec. Engin., Univ. of Michigan, Oct. 1962.

[14] J.S. Squire and S.M. Palais: "Programming and design considerations for a highly parallel computer," *Proc. Spring Joint Computer Conf.,* pp.395-400, 1963.

[15] H. Sullivan and T R. Bashkow: "A large scale, homogeneous, fully distributed parallel machine, I," *Proc. Computer Architecture Symp.,* pp. 105-117, 1977.

[16] H. Sullivan, T.R. Bashkow and D. Klappholz: "A large scale, homogeneous, fully distributed parallel machine, II," *Proc. Computer Architecture Symp.,* pp. 118-124, 1977.

[17] L. G. Valiant: "A scheme for parallel communication," *SIAM Jour. of Computing,* vol. 11, pp.350-361, May 1982.

[18] R.P. Weicker: "Dhrystone: a synthetic systems programming benchmark," *Communic. ACM,* vol. 27, pp. 1013-1030, Oct. 1984.

# Scalability of a Binary Tree on a Hypercube

Sanjay R. Deshpande and Roy M. Jenevein

Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712.

## Abstract

The concept of scalability is important for the next generation of super-multiprocessors. Two aspects of scalability of an architecture are *resource scalability* and *application scalability*. Both types of scalabilities are qualitative measures of goodness of an inductive architecture [4]. The application scalability measures the utilization of resources and the efficiency of execution of application. The resource scalability is a measure of growth rate of architectural properties.

In this paper, the resource scalability characteristics of the hypercube architecture are assessed and the concept of application scalability is applied to it in the context of a binary tree structured application graph. The criterion used is the utilization of processing nodes. Results are also derived for a modification of the application graph. A logN[1] time complexity distributed algorithm that can be used to set up the modified structure is outlined. This algorithm is shown to be useful for handling a single node fault in the architecture as well.

## 1 Introduction

An important consideration in designing an architecture for a super-multiprocessor system is its scalabilities [4,5]. We recognize two types of scalabilities: resource scalability and application scalability.

By resource scalability we imply the asymptotic growth rates of architectural properties and their associated costs. The smaller the cost, the better scalable is the architecture. For example, the multistage interconnection networks such as the omega, the banyan and the baseline are preferred for multiprocessor systems over the crossbar for the reason that these networks scale better than the crossbar. The hardware cost increases as NlogN for these networks compared to $N^2$ for the crossbar, for N interconnected resources.

Application scalability is a measure of utilization of architectural resources and efficiency of execution for a particular application under asymptotic growth in their size. In other words, it is a measure of optimality of mapping between the application and the architecture for all sizes. This concept is of relevance for inductive architectures [4], which are regular in structure and have the number of processing resources as a parameter. Some examples of inductive architectures are TRAC [13], PASM [14], Hypercube [3], CHiP [15] etc.. For these, the mappings can be evaluated as a function of size. By evaluating their application scalability, we can predict the asymptotic execution behavior of an application.

Consider an application mapped on a given architecture. The mapping results in certain utilization of resources and execution efficiency. If, when the architecture and the application are equally increased in size, the same utilization and execution efficiency is achieved, the architecture is considered to be well scalable with respect to that application. Of course, in some cases, a drop in execution efficiency can be allowed if it is the result of increased hardware delays.

It is not sufficient for a super-multiprocessor architecture to have good resource scalability; it must also have good application scalability for a number of applications. For instance, a bus oriented architecture has excellent resource scalability, but its bandwidth is exceeded for even modest communication requirements per processor, as the number of processors is increased beyond a certain value. The evaluation process should therefore account for architectural constructs embedded in software as well as hardware. These include, but are not limited to, software communication switches, polling routines, priority encoders, data translators, etc.. If these constructs are not accounted for, they may result in bottlenecks and degraded performance as the application grows in size.

The evaluation of application scalability of an architecture involves mapping of an application graph on the architectural topology. Conventionally, the mapping implies embedding of an application graph within the resource graph of the supporting architecture. The mapping problem arises because the two graphs are topologically nonisomorphic [6]. The solution results in assignment of one or more computation nodes of the application graph to a processing node in the system such that the processing and storage requirements are satisfied.

[1] Throughout this paper, unless otherwise specified, logN is used as short for $\log_2 N$.

0190-3918/86/0000/0661 $01.00 © 1986 IEEE

We have chosen an application graph which has the structure of a balanced binary tree. Binary tree graphs are fundamental structures and appear in several computation graphs. They naturally result from *divide and conquer* methodology of formulating parallel algorithms for sorting, merging and min-max problems. A binary tree based algorithm is used for the recursive doubling method of computing a global histogram [7]. The Dictionary Machine of Atallah and Kosaraju uses a binary tree structured machine architecture [8]. Binary tree structures result from search trees in inference systems. Horowitz and Zorat suggest application of binary tree shaped interconnection networks for multiprocessor systems [9]. It would therefore be interesting to see how such an important computational topology is supported on a popular architecture.

Recently, substantial attention has been given to the hypercube architecture for multiprocessors [1, 2]. Many applications have been successfully implemented on it and impressive speed-ups have been obtained [3]. Most previous architectures have failed to furnish respectable speed-ups for more than just a few applications. On the other hand, the hypercube architecture seems to have lived up to its promise of being a good supporting architecture for a wide variety of application areas. This effectiveness of the hypercube architecture is seen to derive from the rich internode connection topology afforded by the underlying graph. But, before hypercube topology can be considered as a defining topology for a future generation super-multiprocessor, its resource scalability should be evaluated. Also of consequence to its applicability for a dedicated high performance computing engine is its application scalability for the targeted use. Binary tree structured algorithms being so common in the area of parallel processing, we focus here on their scalability on the hypercube.

## 2 Resource Scalability of Hypercube Architectures

A boolean or binary hypercube graph of order n (alternately called n-cube) has $2^n$ nodes connected with edges as defined in Appendix I. Figure 1 shows an example of a hypercube of order 3. In a hypercube based multiprocessor, nodes of the graph are occupied by independent processing elements. The edges between the nodes represent the point-to-point communication links between the processors. Each link or edge is dedicated to the corresponding node-pair. In some architectures, like the Intel's iPSC System, there is a separate processor called the Cube Manager to coordinate the parallel execution of a job [10]. The Cube Manager is connected to the processors in the cube by a broadcast bus for global communication, I/O and control. However, the concepts of Cube Manager and the global bus are not inherent to the hypercube architecture, and we will therefore ignore them. We will use the well-known rectangular, multistage, single-sided interconnection networks (MSINs) such as banyan, omega and baseline as a reference when comparing properties. These networks have been shown to be topologically isomorphic in [18]. We assume in this paper that the MSINs have logN stages of N/2 switches each, and that switches allow "turn-around". Thus a message going from one processor to another travels a certain number of levels into the network and then turns around and travels to the destination in the reverse direction.

### 2.1 Communication Interface Costs

Each processing element in an n-cube has communication links to n other processors. If we focus on the number of processors, $N = 2^n$, in the cube, then there are logN communication interfaces on each processor. Thus the total communication interface cost of the network is NlogN. This is the same as the cost of the MSINs, although there is one minor difference: when the size of the hypercube is doubled, each of the original nodes has to be modified to allow one more communication link. The nodes of the original cube could be provided with ports for extra links, and when the cube is doubled, one of these could be utilized. However, it is important to remember that modification is necessary throughout the original system in the case of hypercube; whereas in the multistage network architectures, the modifications are localized to the boundary of the original system.

### 2.2 Communication Reliability

The communication reliability of the hypercube ($n \geq 3$) is better than that of the MSINs. For them, unless extra stages are specifically provided for [16, 17], there are a constant number of disjoint paths (= out-degree of a node) between two processing nodes. On the other hand, in the hypercube architecture, there are (logN) disjoint paths available for any given pair of nodes [11].

Note that, in the hypercube architecture, the number of contingent paths and the reliability of interprocessor communication increases with logN, while those for the MSINs remain constant. Thus for the hypercube architecture, the communication reliability also scales better than that for the multistage networks.

### 2.3 Interprocessor Distance

The diameter of a network can be defined as that of the underlying graph structure in which the processing resources and switches are represented as nodes and the communication paths are represented as edges. For a graph, the diameter is the maximum distance between any pair of nodes. For the MSINs, the diameter is equal to twice the number of stages in the network; that is 2logN. For a hypercube network, the message passes at most logN links before reaching the destination [12], since for any given node, there is one node which is at distance logN from it. So for the hypercube, the diameter scales as logN.

It can be shown that the average distance from a given node, of all nodes including itself, is 2(logN-1) + 2/N, for MSINs. The same is logN/2 for a hypercube. Thus, under asymptotic growth, although both scale as logN, the hypercube has a better scaling coefficient[2].

## 3 Application Scalability for a Binary Tree

For the sake of this analysis, we will assume that the binary tree application is the sole task being executed in the system. Under this condition, we would like to execute the largest possible size of the application. Ideally, we would like to utilize the hypercube architecture fully, so that when the architecture and the application are scaled, no wastage results. If that is not possible, a constant resource overhead is allowable since the percentage of overhead would diminish with equal increase in the application size and architecture. The overhead that grows linearly with the number of nodes in the hypercube is unacceptable.

Let us consider mapping a binary tree on an order n hypercube. A balanced binary tree with n levels of nodes (called in this paper an n level tree), has a total of N-1 nodes of which N/2 are at the leaf level. It should therefore be possible to map an n level binary tree on an n-cube. This would result in all but one of the nodes being utilized and would represent constant under-utilization. If such mapping is possible for all n, the binary tree application could be considered well-scalable on a hypercube architecture. We ask precisely this question: Is such mapping possible?

It is possible to map a two-level binary tree on a 2-cube that has a square topology (See Figure 2). For all n $\geq$ 3, it is impossible to map an n level binary tree on a hypercube of order n. Appendix I contains the proof for this result. For n $\geq$ 3, the largest sized binary tree that can be mapped on an n-cube is of n-1 levels. Thus the tree occupies only about half ($2^{n-1}$-1) nodes out of the $2^n$ available. Resultant under-utilization of computing resources is $2^{n-1}$+1 nodes, which is slightly more than 50%. In terms of the total computing resource available in the system, the overhead is linear and therefore unacceptable.

Proof that an n-1 level binary tree can be mapped on an n-cube forms a part of Appendix II. In fact, two n-1 level binary trees can be simultaneously mapped on an n-cube.

The result obtained in Appendix I tends to question the suitability of hypercube topology for super-multiprocessor architecture on the grounds that it cannot support the scaling of a fundamental application. A question we may ask is: Is it possible to modify the application graph such that the new structure scales well on the hypercube architecture without introducing overheads that grow inordinately?

Appendix II shows that it is indeed possible to slightly modify the original tree graph and make it well-scalable on the cube. That is, a modified n level tree is optimally mappable on an n-cube. By the introduction of a single two-degree node as a son of its root, and thereby *stretching* (or equivalently double-rooting) it, the tree can be made to utilize the cube completely (see Figure 3b). The extra node so introduced is used only for communication between the root and one of its sons. It is interesting to note that only one such node is required for any n. This node therefore represents a constant overhead for all n $\geq$ 3 which diminishes in percentage as n becomes larger.

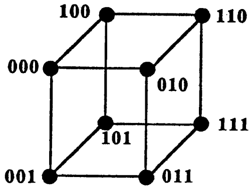


Figure 1.

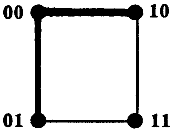


Figure 2.

Since each processing node in the cube accommodates a single computation node of the tree, equal increase in sizes of both application and architecture does not change the computational load on an individual processor. Upon scaling, the links between the processors continue to correspond to edges between neighboring nodes in the tree on a one-to-one basis and handle the same level of traffic as before. This ensures that the computational efficiency of the application too does not drop significantly, although a constant time is added because of the communication through the spacer node.

## 4 A Distributed Tree Set-Up Algorithm

In the previous section we saw that an n level stretched binary tree can be mapped onto a hypercube of order n. In order to prove that result, we have introduced transformations of the node labels. It is possible to use these transformations, namely FT3 and BT3, to obtain a distributed algorithm to set up the referred mapping. This algorithm has practical importance for executing a binary tree shaped application on a hypercube. It is used first to set up the communication topology among the processors of the cube, and then the application is executed. The algorithm does not require that the entire system be configured as a tree. It works as well within any subcube of the overall system. The algorithm is explained below.

The algorithm is executed by each processor in the hypercube. It is divided into two phases; namely compute and distribute. During the first phase, the nodes of the cube compute the loci of the mapped trees as they are transformed and merged to form larger trees. In the second phase, the port configurations of the nodes forming the final tree are sent to the appropriate nodes of the cube.

The algorithm starts by setting up initial $2^{n-3}$ three level trees having a predetermined configuration. That is,

[2] Certain other multi-tree networks, like the KYKLOS [19], are known to offer average interprocessor distances which are better than that for the MSINs.

every node in the cube is a member of a three-level tree; its position in the tree is determined by the least significant three bits of its ID. Each three-level tree is defined over a set of eight nodes which have identical n-3 most significant bits. The configuration of these trees are shown in Figure 5. The labels of the nodes of these mapped trees are henceforth referred to as *virtual addresses*. At each successive iteration of the algorithm, trees are merged to form larger trees until eventually the final tree is attained. These intermediate trees are not realized by activating links between appropriate nodes in the cube. However, the cumulative information available in the cube is sufficient to make this possible.

At the outset, the virtual address of a tree node corresponds to the label of the physical node in the cube to which it is mapped. After this, the physical nodes compute the loci in the n-cube of these initial tree nodes as the mappings are transformed and merged to form larger trees. At the end of the compute phase of the algorithm, each physical node holds the node label to which the virtual address would eventually be mapped, and also the port assignments required at this final node to configure the tree properly. These port assignments are relayed to the final node during the distribution phase of the algorithm.

For an n level tree in an n-cube, O(logN) iterations are required, and if we assume no collisions during the distribution phase, it too is O(logN) long, since each node in the cube emits one message. It therefore implies that the algorithm is of O(logN) time complexity. The algorithm is given in Appendix III.

## 5 Tree Mapping in Presence of a Node Fault

We consider here the mapping of a stretched binary tree topology in presence of single node faults. We make an assumption while formulating the strategies: the knowledge of this failure is global. That is, all processing nodes know before hand the label or Id of the failed node. This prior knowledge is not essential, and an algorithm to distribute the labels of the failed nodes can be formulated if necessary.

We will first consider a multiple step approach where each step represents one mapping of a portion of the computation graph and its subsequent execution. Clearly, an n-1 level sub-tree with a spacer node occupies $2^{n-1}$ nodes and thus is mappable into half of a hypercube of order n. By utilizing this partitioning of the cube, the corresponding computational graph must be segmented into three components: the root, the left sub-tree and the right sub-tree. In this way, each sub-tree segment of the graph can be mapped onto the valid half partition of the cube in which no nodes are faulty, while the faulty node is isolated to the other half.

Because the proof in Appendix II works inductively for successively higher order cubes, the reverse fragmentation to lower cubes follows directly, provided the cube partition is done on a dimensional boundary. Remembering that each successively higher order cube adds one bit to the node addresses, a mapping partition can consist of all nodes whose high order bit is the complement of that of the faulty node. In this way the problem is reduced to that of mapping a sub-tree to a subcube. Figure 6 shows the valid half of the cube as those nodes with the high order bit in their labels to be 1 since it is assumed that the faulty node has a 0 as the high order bit of its label.

The disadvantage with this approach, however, is that it may lead to a three-step process of map-and-execute (left tree, right tree and then root), and additionally, because of a single faulty node, $2^{n-1}$-1 nodes are left unutilized. This seems a high price to pay to avoid a fault. We now address the question of whether one can map-and-execute in a single step.

A three-level tree with the spacer node is shown in Figure 5. In this figure, we can see that the root of the tree occupies physical node ID 000 and the spacer node ID is 100. After mapping, only one node within the cube has no corresponding computational requirement in the stretched tree. We observe that the spacer node (S100 in Figure 4a) has as its only responsibility, the communication of results from the right sub-tree to the root. Certainly, this communication is an essential part of the computation graph. But, S100 is not the only node linking S000 and S110 within the hypercube, and because the node S010 likewise links them, it could be used instead.

Let us assume for the moment that we have a means to "force" the faulty node to be S100. In this way, all computation nodes are mapped to working processors and S010 doubles as both a computation node and an alternate link from the right sub-tree to the root (see Figure 9). In all of the tree algorithms discussed, at the time when this communication is needed all members of the right sub-tree have completed their computation and gone idle. This implies that no timing conflict arises as a result of this dual role for S010.

The remaining open question concerns the ability to "force" the faulty node to the spacer node. We answer this question via an example of a 3-cube. The method can be extended easily to an n-cube.

We will assume a single fault at node 001, which according to the mapping in Figure 7, is a computational node. We assume that the faulty node address (001) is available to all nodes in the cube. We now provide a means to remap the initial three level tree such that the fault will map to a logical node 100 and all other nodes will be transformed through three dimensional space conserving their adjacency relationships.

Figure 8 shows that an XOR bit mask controls the reflection or non-reflection of each bit in the node label. If the bit in the mask is 0, no reflection occurs; but if it is a 1, then a pair of node labels are swapped in that dimension.

Thus, a remapping of the labels occurs, but adjacency is preserved. This mask can be generated by taking the exclusive OR(XOR) between the fault node id and the spacer id (100). The XOR of this mask and each node id create virtual addresses which force the fault to be mapped to the spacer node as shown in Figure 9.

The above method can be extended to an n-cube if we can designate a particular label as that of the spacer node. The tree set-up algorithm of Appendix III always produces a spacer node at S100, where S is a string of n-3 zeros. Thus, in the case of an n-cube, all nodes use their own physical labels as virtual addresses to start with. Subsequent to the tree set-up algorithm, the resultant virtual addresses are XORed with the mask to obtain an appropriate tree with a spacer node mapped onto the faulty physical node.

## 6 Summary and Conclusions

In this paper the concept of application scalability was introduced and applied to an example of a hypercube architecture and a binary tree application. For inductive architectures like the hypercube, application scalability is an indicator of how well a given application will be supported with increasing sizes. The parameters of interest, while evaluating application scalability, are utilization of hardware resources and efficiency of execution. The first of the parameters is evaluated by mapping a maximum sized application graph onto the architecture graph and performing a calculation of the utilization as a function of size. The second is analyzed by measuring the growth of computation and communication delays as a function of size.

It was concluded that a balanced binary tree is not well scalable on a hypercube. It was also proved that a stretched binary tree results in a scalable mapping giving near 100% utilization of processing resource in the architecture. In addition, an O(logN) distributed algorithm was derived to generate a stretched binary tree in the hypercube. The tree construction algorithm was also shown to be adaptable to generate a mapping in presence of a single node fault by augmenting it with a subsequent fault-avoidance step.

Of the two approaches evaluated for executing the tree application in the presence of a single node fault, one provides complete utilization of fault-free nodes for an n level tree in an order n hypercube. Thus, even with single node fault, the binary tree application can be mapped and run with virtually no degradation in performance.

## Acknowledgements

This research was supported by AFOSRF49620-84-C-0020, DCR-8116099, DE-AS05-81ER-10987 and Space and Naval Warfare Systems Command Contract N00039-86-C-0167. The authors would like to thank Prof. J. C. Browne, Matt Sejnowski and Ashok Adiga for their insightful comments.

## Appendix I

Definition: A hypercube of order n (an n-cube) is an undirected graph with $2^n$ vertices labelled 0 through $2^n$-1. There is an edge between a given pair of vertices if, and only if, the binary representation of their labels differ by one and only one bit. Hypercube of order 3 is shown in Figure 1.

Definition: Parity of a node is odd or even and is determined by the number of 1's in the label of the node.

Lemma 1: If nodes a and b are connected by an edge, a and b have opposite parities.

If we try to map a binary tree on a hypercube, we can observe the following lemma:

Lemma 2: The node parities alternate over the levels of a mapped binary tree.

We now count the number of odd and even nodes necessary to map a binary tree, and the number of odd and even nodes available in a hypercube:

By symmetry, the number of odd and even nodes available in the hypercube is the same and is half the total number of nodes, that is $\frac{1}{2}(2^n) = 2^{n-1}$. For a binary tree, we get the following values:

**Case 1:** n is even.

Number of even nodes $= 2^0+2^2+....+2^{n-2} = \frac{1}{3}(2^n-1)$

Number of odd nodes $= 2^1+2^3+....+2^{n-1} = \frac{2}{3}(2^n-1)$

**Case 2:** n is odd.

Number of even nodes $= 2^0+2^2+....+2^{n-1} = \frac{1}{3}(2^{n+1}-1)$

Number of odd nodes $= 2^1+2^3+....+2^{n-2} = \frac{2}{3}(2^{n-1}-1)$

In both cases the number of even and odd nodes required by the tree do not match those of the cube. This completes the proof.

## Appendix II

Figure 3a shows a binary tree of n=3. The figure also shows the even and odd levels of the tree, assuming level 0 of the tree is at an even level. Figure 3b shows the construction of a 4-level tree out of two 3-level trees using an extra node at level 1 of the right hand subtree. The resultant tree has equal number of odd and even nodes which can be matched by those in a 4-cube. The following is a constructive proof that such a mapping is possible for n $\geq$ 3.

Definition: An n-cube is said to be k-dimensionally transformed when k bits in the label of each node of the cube are transformed according to a transformation. (Note that the bits undergoing transformation need not be contiguous.)

We define two 3-dimensional transformations, FT3 and BT3, as shown in Table 1. The $x_i$, $x_j$ and $x_k$ are the original values of the bits, and $y_i$, $y_j$ and $y_k$ are the corresponding resultant values.[3]

Definition: Distance between a pair of nodes in a graph is the minimum number of edges that have to be traversed when going from one node to the other. For an n-cube, this is also equal to the number of bits by which the binary representations of labels of the two nodes differ [11]. The *adjacency* in an n-cube is the distance relationship of a node with the rest of the nodes in the cube.

We state the following two theorems proved in [12].

Theorem 1: The FT3 preserves cube adjacency.

Theorem 2: The BT3 preserves cube adjacency.

We can now state the following corollaries:

Corollary 1: Nodes with distinct labels map to distinct resultant labels.

Corollary 2: If two nodes are neighbors and thus have labels differing in one bit position, their new labels after the transformations also differ in one bit position. Thus, neighborhood between the two nodes is preserved.

Theorem 3: A graph G mapped on an n-cube remains unchanged in structure after the transformations.

Proof: The proof follows as a direct consequence of corollaries 1 and 2, and is given in [12].

We can therefore state the following corollary:

Corollary 3: The transformations FT3 and BT3 preserve a tree structure mapped on an n-cube.

Theorem 4: A stretched binary tree (of the form shown in Figure 3b) of n levels can be mapped on an n-cube, for $n \geq 2$.

Proof: The case for n=2 is trivial. The case of n=3 is shown in Figure 5. The construction of a four level tree from a pair of 3-cubes can be found in [12]. We will now describe the inductive step. We assume that an n level stretched binary tree, as shown in Figure 4a, is mapped on an n-cube. To obtain an n+1 level tree in an n+1-cube:

- Duplicate the cube and the mapping. See Figure 4b.
- Apply FT3 to the duplicate mapping using bit-2 as $x_i$, bit-1 as $x_j$ and bit-0 as $x_k$ to obtain a new mapping as in Figure 4c.
- Form an n+1-cube by connecting the nodes with like labels in the two n order subcubes. Append a 0 to the left of labels in the original subcube and a 1 to the left of those in the duplicate subcube (see Figure 4d).
- Deallocate links: 0S100-0S110 and 1S100-1S000, and allocate links: 0S000-1S000, 0S110-1S110 and 0S100-1S100. See Figure 4e.
- Apply the BT3 to the n+1 cube to obtain an n+1 level stretched binary tree rooted at 0S000. While applying the BT3, use the most significant bit as $x_i$, the third least significant bit (bit-2) as $x_j$, and the least significant bit (bit-0) as $x_k$. Replace the string 0S by S' to obtain a structure similar to the base structure we started with. See Figures 4f and 4g.

## Appendix III

The tree set-up algorithm uses the following variables:

*current-port*: Every node in the cube has logN ports, each corresponding to a bit position. This variable keeps a running pointer to the current bit position being considered.

*physical-id*: Original label of a node.

*current-id*: A node label during current iteration.

*port-relation*(1..logN): An array of values specifying the current active connections of a node. All are initialized to "null" (inactive).

Following values are assigned to *port-relation(i)* variable:

| | |
|---|---|
| "null": | No active connection. |
| "f": | Connection to "father" node. |
| "s": | Connection to "son" node. |

The following is the compute phase of the tree set-up algorithm. This is followed by a distribution phase during which the connectivity information is sent to the appropriate physical node specified by the *current-id* variable.

```
for-all nodes do
  begin
  current-id = physical-id;
  {Set up $2^{n-3}$, 3 level trees.}
  case current-id( bits: 2..0) of
     0:  port-relation(0) = port-relation(2) = "s";
     1:  port-relation(0) = "f";
         port-relation(1) = port-relation(2) = "s";
     2:  port-relation(2) = "f";
     3:  port-relation(1) = "f";
     4:  port-relation(1) = "s";
         port-relation(2) = "f";
     5:  port-relation(2) = "f";
     6:  port-relation(0) = port-relation(2) = "s";
         port-relation(1) = "f";
     7:  port-relation(0) = "f";
```

[3] The two transformations define two of the many *rigid* transformations of a 3-cube. The rigid transformations include rotations and reflections of the original cube structure which preserve the adjacency between its nodes.

```
end-case
for current-port = 3 to logN-1 do
   begin
   {Form larger trees iteratively.}
   if (current-id(current-port) = 1) then
   begin
   Apply FT3 to current-id's bits 2, 1 and 0;
   end
   if (Bits 3 through current-port-1 are 0) then
   begin
      case current-id( bits: current-port, 2, 1, 0 ) of
      0:  port-relation(2) = "f";
          port-relation(current-port) = "s";
      4:  port-relation(2) = "s";
          port-relation(current-port) = "s";
          port-relation(1) = "null";
      6:  port-relation(1) = "null";
          port-relation(current-port) = "f";
      8:  port-relation(2) = "null";
          port-relation(current-port) = "f";
      12: port-relation(2) = "null";
          port-relation(current-port) = "f";
      14: port-relation(current-port) = "s";
      end-case
   end
   Apply BT3 to bits current-port, 2 and 0 of current-id;
   end
end
```

## References

[1] "The Mark III Hypercube-Ensemble Concurrent Computer"; J. C. Peterson et al., *ICPP* 1985.

[2] "The Cosmic Cube"; C. Seitz, *CACM*, January 1985.

[3] "CALTECH/JPL Mark II Hypercube Concurrent Processor"; J. Tuazon et al., *ICPP* 1985.

[4] "Inductive Computer Architectures: A Class of Supercomputer Architectures"; G. J. Lipovski, M. Malek and J. C. Browne, MCC Report, Parallel Processing Group, Microelectronic and Computer Technology Corporation, Austin, Texas.

[5] "Microprocessors: Architecture and Applications"; P. C. Patton, *IEEE Computer*, June 1985.

[6] "On the Mapping Problem"; S. Bokhari, *IEEE Transactions on Computers*, March 1981.

[7] Introduction to Computer Architecture; Editor: H. Stone, Science Research Associates Inc., 1975.

[8] "A Generalized Dictionary Machine for VLSI"; M. J. Atallah and S. R. Kosaraju, *IEEE Transaction on Computers*, February 1985.

[9] "The Binary Tree As An Interconnection Network: Applications to Multiprocessor Systems and VLSI"; E. Horowitz and A. Zorat, *IEEE Transactions on Computers*, April 1981.

[10] iPSC System Overview; Intel Corporation, October 1985.

[11] "Topological Properties of Hypercubes"; Y. Saad and M. H. Schultz, Research Report YALEU/DCS/RR-389, June 1985.

[12] "Scalability of a Binary Tree on a Hypercube"; S. R. Deshpande, Department of Computer Sciences Technical Report TR-86-01, University of Texas at Austin.

[13] "An Overview of the Texas Reconfigurable Array Computer"; M. C. Sejnowski et al, *Proc. of AFIPS NCC Conference*, 1980.

[14] "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition"; H. J. Siegel, *IEEE Transactions on Computers*, December 1981.

[15] "Introduction to the Configurable, Highly Parallel Computer"; *IEEE Computer*, January 1982.

[16] "Reliability and Fault Diagnosis Analysis of Fault-Tolerant Multistage Interconnection Networks"; V. Cherkassky et al, *FTCS*, 1984.

[17] "The extra stage cube: a fault-tolerant interconnection network for supersystems"; G. B. Adams III and H. J. Siegel, *IEEE Transactions on Computers*, May, 1982.

[18] "On a Class of Multistage Interconnection Networks"; C. L. Wu and T. Y. Feng, *IEEE Transactions on Computers*, August, 1980.

[19] "KYKLOS: A Linear Growth Fault-Tolerant Interconnection Network"; B. L. Menezes and R. M. Jenevein, *ICPP*, 1985.

| FT3 | | | | | | BT3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_i$ | $x_j$ | $x_k$ | $y_i$ | $y_j$ | $y_k$ | $x_i$ | $x_j$ | $x_k$ | $y_i$ | $y_j$ | $y_k$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Table 1.

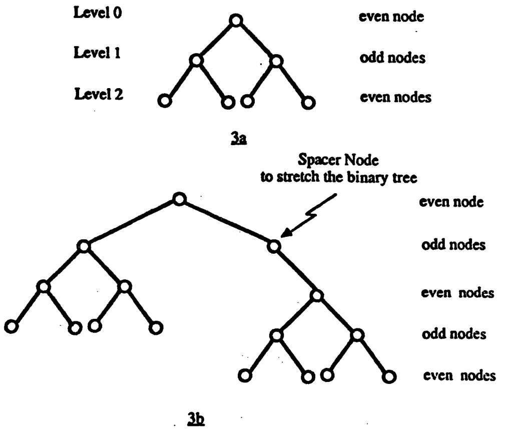


Figure 3.

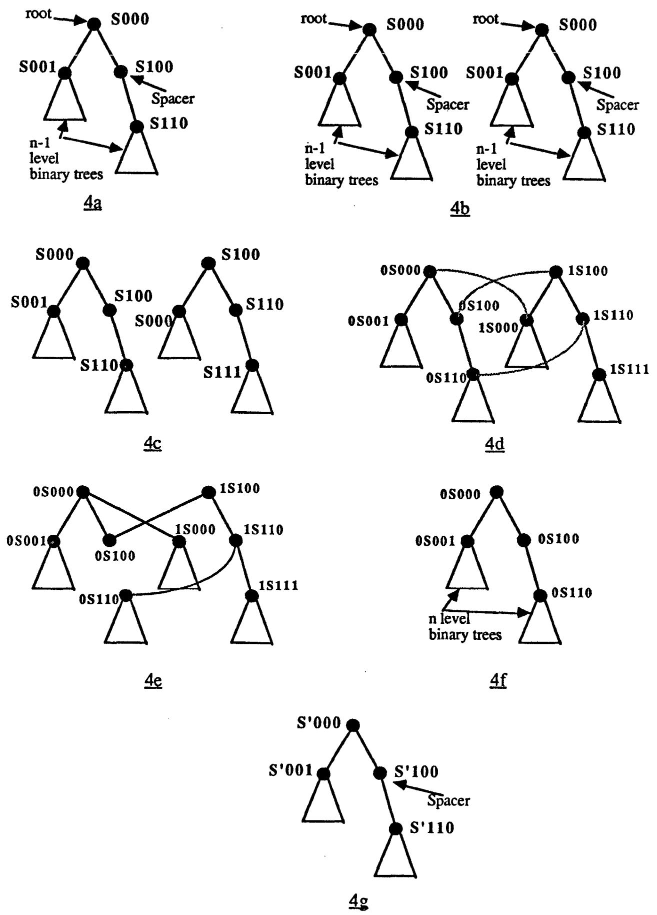


Figure 4.

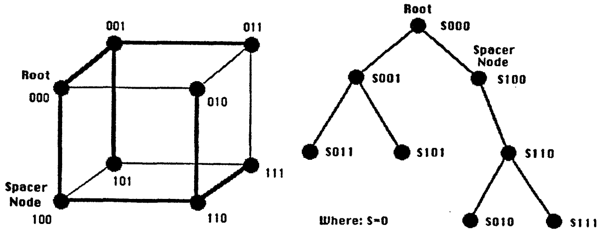


Figure 5.

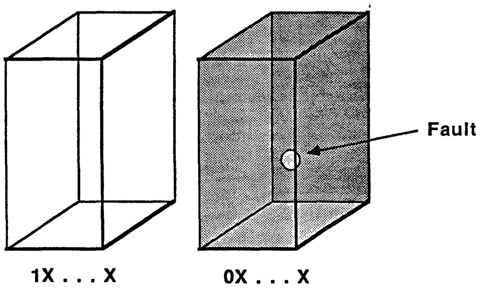


Figure 6.

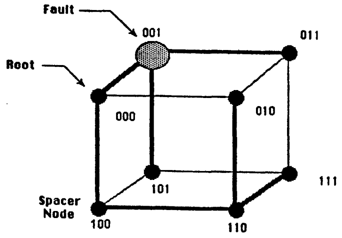


Figure 7.

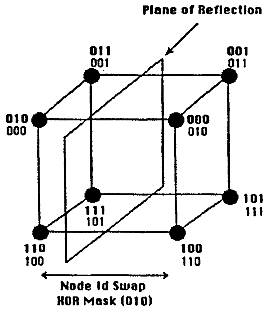


Figure 8.

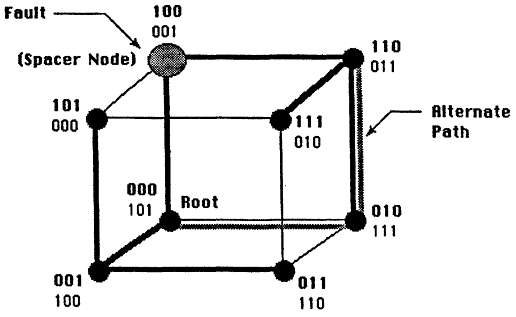


Figure 9.

# SYSTOLIC TREE IMPLEMENTATION OF DATA STRUCTURES

Jik H. Chang[1]
Oscar H. Ibarra[1]
Department of Computer Science,
University of Minnesotta,
Minneapolis, MN 55455.

Moon Jung Chung[2]
Kotesh K. Rao[2]
Department of Computer Science,
and Center for Integrated Electronics,
Rensselaer Polytechnic Institute,
Troy, NY, 12181.

## ABSTRACT

We present systolic tree architectures for data structures such as stacks, queues, priority queues, deques and dictionary machines. Except for the dictionary machine, all data structures have a unit response time. In each node of the tree the mechanism for controlling the transmission and distribution of data is a finite state machine. The depth of a tree with n data elements is O(log n). For stacks, queues, and deques, no compress signals are required.

## 1. Introduction

Several researchers have explored systolic machines for implementing data structures. Leiserson [LEI79], and Guibas and Liang [GUI82] have given linear systolic array implementations of priority queues, stacks and queues. Ottman et. al. [OTT82], Atallah and Kosaraju [ATA85], and Somani and Agarwal [SOM85] have shown that dictionary operations can be implemented on a tree. These schemes require additional clock cycles[LEI79][GUI82], or extra connections[OTT82], or storage of several addresses [ATA85].

This paper presents a relatively uniform approach to handling several types of data structures. All the data structures use finite state control, and with exception of the dictionary machine, have a unit response time. The response time of the dictionary machine is O(logn). The input and output in these machines is done through the root of a tree. Throughout this paper, n is the number of data elements in the tree. We will first present a tree-based implementation of stacks, queues and deques. Using a different approach, a priority queue and a dictionary machine are developed with a wide repertoire of operations. Compress cycles are necessary in the second approach to allow successive deletions.

## 2. Stack, Queue, Deque

An output restricted deque(Knuth[KNU68]), shown in Figure 1, is a data structure which permits insertions to both the front and the rear. Deletions, however, are only permitted from the front. The output-restricted deque is a generalization of both the stack and the queue since restricting insertions from the rear makes this a stack while restricting deletions from the front makes this a queue.

Consider a binary tree each of whose nodes has a data register for storing data, a buffer for pipelining instructions to its children, and two flags. The root of the tree corresponds to the front of the output restricted deque. An instruction entering a node is executed, and the instruction to be pipelined is stored in the buffer of the node. In the next cycle, the buffer contents is sent to either the left child or the right child.

There are two insert instructions- a Stack Insert (SI) and a Queue Insert(QI). A Stack Insert into a node causes the inserted value to be stored in the node, while the previous contents are directed to one of the children. A Queue Insert causes the inserted data to be pipelined to one of the children while retaining the contents of the data register. Each node has two binary flags, a queue flag(QF) and a stack flag(SF). When an empty node receives an insert instruction, its flags are assigned 0. In every non-empty node a stack insertion is directed towards the left*(right)* subtree if the SF is 1*(0)*. A queue insert is directed left*(right)* if the Queue Flag(QF) is 0*(1)*. A deletion is from the left*(right)* subtree if the SF is 0*(1)*. The SF is complemented if either a Stack Insert or a Delete enters a node, while the QF is complemented if there is a Queue Insert. The behaviour of the output restricted deque is given in Table 1. Figure 2 shows the evolution of the output restricted deque under a sequence of instructions. A deque can be implemented by combining two output restricted deques [CHA85]. The complete implementation of the stack and queue is also described in [CHA85].

## 3. Priority Queue

In our tree implementation of a priority queue and a dictionary machine, each node stores upto three data elements. Data is 'snaked' as in [ATA85]. Balancing the tree is achieved by a flag at each node. All operations can be pipelined.

First let us consider the priority queue. Our implementation supports Deletemax and Deletemed in addition to Insert and Deletemin. Deletemin, Deletemax and Deletemed applied to a node x remove the smallest, largest and the $\lceil n/2 \rceil$th smallest (i.e. median) elements from the list of data elements contained in the subtree generated by x.

Every node x has three storage registers (L(eft), M(iddle) and R(ight)), the contents of which are denoted by x.L, x.M and x.R. Each non-leaf node x has x.L, x.M and x.R as the smallest, median and largest data elements respectively in the subtree generated by x. A non-leaf node is called "balanced" if the number of data elements of its left subtree is equal to that of its right subtree. A node is called "right biased" if its right subtree has one more data element than its left subtree. Every node is either "balanced" or "right biased" which is denoted by a flag. If a node x does not contain three data (i.e. it is a leaf node), insert and delete operations at that node are obvious. For example, if x has two data elements, x.L and x.M, and if I[Y] is to be executed, we only need to adjust x.L, x.M, and x.R, accordingly. No instruction will be pipelined further.

Consider operations on a node x which has three data elements. First consider I[Y] applied at a node x with the left child y and the right child z. Temporarily, we ignore the balancing.

(case 1) If Y< x.L, x sends I[x.L] to its left child y and sets x.L to Y.
(case 2) If x.L<Y< x.M, x sends I[Y] to its left child y.
(case 3) If x.M<Y< x.R, x sends I[Y] to its right child z.
(case 4) If Y>x.R, x sends I[x.R] to its right child z and sets x.R to Y.

[1] Research supported in part by NSF Grant MCS 83-04756.
[2] Research supported in part by GE and DEC.

0190-3918/86/0000/0669 $01.00 © 1986 IEEE

Now we consider the balancing. If a node is balanced and sends an insert instruction to its right child, or if it is right biased and sends an insert instruction to its left child the node simply changes its flag. Otherwise, the tree needs to be balanced. To simplify the description, we assume that if a leaf node w has only one element. then $w.L=w.M$. Also, if a leaf node has two data elements, they are stored in the registers L and R, accordingly.
(i) If node x is balanced before the insertion I[Y] is applied and $Y<x.M$, an insert instruction must be sent to its left child (as in case 1 and 2) above. The following modification is therefore required.

1. x sends I[x.M] to its right child z and sets x.M to max(Y,y.R).

2. If $Y<x.L$, x sends an instruction IR[x.L] to the left child y and sets x.L to Y. (The instruction IR[b] when applied to a node y will delete y.R and insert b at y).

   If $x.L<Y<x.M$ and $Y>y.R$, then no instruction is sent to y (note that x.M is set to Y).

   If $x.L<Y<x.M$ and $Y<y.R$, then x sends IR[Y] to its left child y (note that x.M is set to y.R)

3. x changes its flag to 1.

(ii) If x is right biased before I[Y] is applied and $Y>x.M$, then the following modification is necessary because an insert instruction must be sent to its right child.

1. x sends I[x.M] to its left child y and sets x.M to min(Y,z.L).

2. If $Y>x.R$, x sends an instruction IL[x.R] to its right child z and sets x.R to Y. where an instruction IL[W] applied at node z will delete z.L and insert W at z.

   If $x.M<Y<x.R$ and $Y<z.L$. then no instruction is sent to z (note that x.M is set to Y).

   If $x.M<Y<x.R$ and $Y>z.L$. then x sends IL[Y] to its right child z.

3. x changes its flag to 0.

As shown above, some insert instructions may generate instructions IL and IR in the next stage. Let us consider the process of IL[Y] at node x. IR[Y] can be executed similarly. If node x is a leaf node, IL[Y] simply deletes x.L and inserts Y into node x. Consider an IL[Y] to a nonleaf node x with left child y and right child z.
If $Y<y.L$, x.L is set to be Y and no further processing is necessary. Otherwise, x sets x.L to y.L and does the following process:
(a) If $y.L<Y<x.M$, x sends IL[Y] to y.
(b) If $x.M<Y<z.L$, x sends IL[x.M] to y, sets x.M to Y.
(c) If $z.L<Y<x.R$, x sends IL[x.M] to y, sets x.M to z.L and sends IL[Y] to z.
(d) If $Y>x.R$, x sends IL[x.M] and IL[x.R] to y and z respectively, and sets x.M to z.L and x.R to Y.

Note that neither IL nor IR instruction changes the flag of the node. The case when node x has only one child can be handled in a similar way. Figure 3 shows the data distributions after a sequence of insert operations are done.

A Deletemin instruction at node x with left child y and right child z causes the following actions. First, x sets x.L to y.L. If x is balanced (before Deletemin is applied), x sends Deletemin to its left child y. If x is right biased, x sends IL[x.M] to y, sets x.M to z.L and sends Deletemin to z. Then, x changes its flag.

Similar operations can be derived for Deletemax and Deletemed. [CHA85] Note that each delete operation must be followed by a compress cycle so that when a delete operation is applied at a non-leaf node x, x must have three data elements after the operation.

## 4. Dictionary Machine

The dictionary machine, in addition to the above priority queue operations, permits Delete[Y], which removes the key Y from the dictionary and is denoted by D[Y]. First, we discuss the action taken at each node as a result of each operation under the assumption that no redundant operation is allowed. Later the issue of redundancy is considered. D[Y] applied at a leaf node can be handled in an obvious way. Suppose that D[Y] is applied at node x with left child y and right child z.

(i) If $x.L=Y$, x sets x.L to y.L and sends down Deletemin to its left child y. If node x is balanced (before D[Y] is applied), no further processing is necessary for balancing. If node x is right biased, then x sends I[x.M] together with Deletemin to its left child y. sets x.M to z.L and sends Deletemin to its right child z.

(ii) If $x.L<Y<x.M$. D[Y] must be sent to the left child y. If x is balanced, no further process at node x is necessary for balancing. If x is right biased, x sends I[x.M] together with D[Y] to its left child y, sets x.M to z.L and sends down Deletemin to its right child z.

(iii) If $x.M=Y$ and x is balanced, x sets x.M to y.R and sends Deletemax to its left child y. If $x.M=Y$ and x is right biased, x sets x.M to z.L and sends Deletemin to the right child z.

(iv) If $x.M<Y<x.R$, D[Y] must be sent to the right child z. If x is right biased, no further process is needed at node x for balancing. If x is balanced, x sends I[x.M] together with D[Y] to its right child z, sets x.M to y.R and sends Deletemax to its left child y.

(v) If $x.R=Y$, x sets x.R to z.R and sends Deletemax to its right child z. If x is right biased, no further process is needed at node x for balancing. If x is balanced, x sends I[x.M](together with the Deletemax) to its right child z, sets x.M to y.R and sends Deletemax to its left child y.
In each of the above cases, node x changes its flag.

As described above, a delete instruction at a node may require the node to send a pair of instructions to its child in the next stage. The execution of an instruction pair is very similar to the execution of the instructions IL and IR generated in a priority queue. Also, the processes that take place for an instruction pair arriving at a leaf node or a node having only one child can be described in an obvious way. Figure 4 shows the configurations of a dictionary machine before and after a deletion.

When redundant insert and delete are allowed, we need extra O(log n) time cycles to check the redundancy. Each instruction has three phases: broadcasting phase, verifying phase and execution phase. During the broadcasting phase, the given instruction will be broadcasted down the tree. The verifying phase starts when the instruction arrives at the leaf nodes. During the verifying phase, the instruction arriving at a leaf node moves up to the root by checking data stored in nodes and the instructions which are in the execution phase. Thus, at the root node, the redundancy of the instruction is finally verified. For example, the instruction I[X] which follows I[X]. and D[X] which follows D[X] should be flagged as redundant. Special care must be taken to consider the two cases.
(i) I[X] is in the execution phase and two D[X] instructions are in the verifying phase. The first D[X] should be non-redundant.
(ii) D[X] is in the execution phase and two I[X] are in the verifying phase. The first I[X] is non-redundant.

## REFERENCES

[ATA85] M.J. Atallah, and S. Rao Kosaraju, "A generalised dictionary for VLSI", IEEE Trans. Comp., vol. C-34, pp. 142-151,Feb. 1985.

[CHA85] J.H. Chang, M.J. Chung. O. H. Ibarra, and K. K. Rao, "Systolic Tree Implementations of Data Structures", Technical Report, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1985.

[GUI82] L.J. Guibas, and F.M. Liang, "Systolic Stacks, Queues, and Counters", in Proc. Conf. on Adv. Res. on VLSI, MIT, Cambridge, Jan. 1982.

[KNU68] D.E. Knuth, The Art of Computer Programming, Vol 1, Addison Wesley, 1968.

[LEI79] C.E. Leiserson, "Systolic priority queues", Report CMU-CS-79-115, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1979.

[OTT82] T. Ottman, A.L. Rosenberg, and L.J. Stockmeyer, "Dictionary machine for VLSI," IEEE Trans. Comp., vol. C-31, pp. 892-897, Sept. 1982.

[SOM85] A.K. Somani, and V.K.Agarwal, "An efficient Unsorted VLSI Dictionary machine," IEEE Trans. Comp., Vol. C-34, pp.841-852, Sept. 1985.

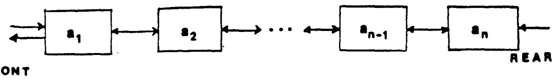


**Figure 1. Output Restricted Deque**

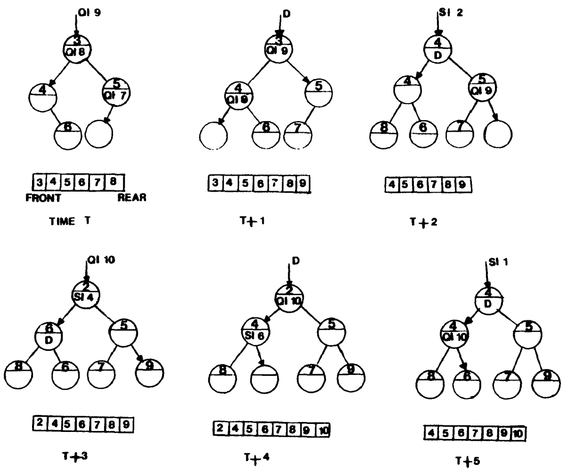
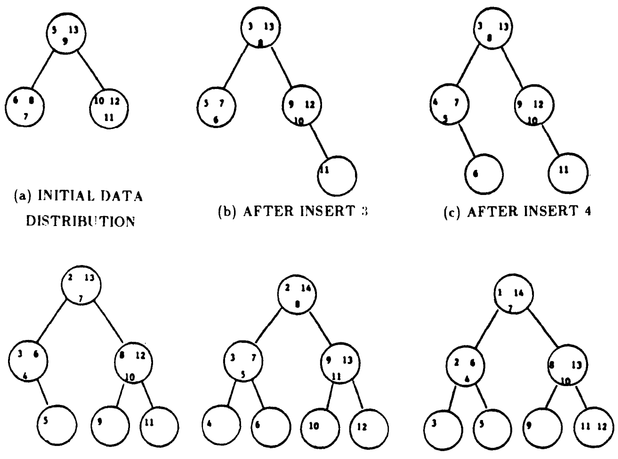


**Figure 2. Evolution of the Output Restricted Deque after Successive instructions.**

| Time T | | | | | | Time T+1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| Instr. | Flags QS | Data | Buffer | Left Child | Right Child | Flags | Data | Buffer | Direction |
| SI[Y] | ab | $\phi$ | ? | ? | ? | 00 | Y | $\phi$ | ? |
| QI[Y] | ab | $\phi$ | ? | ? | ? | 00 | Y | $\phi$ | ? |
| SI[Y] | a0 | X | ? | ? | ? | a1 | Y | SI[X] | Right |
| SI[Y] | a1 | X | ? | ? | ? | a0 | Y | SI[X] | Left |
| QI[Y] | 0b | X | ? | ? | ? | 1b | X | QI[Y] | Left |
| QI[Y] | 1b | X | ? | ? | ? | 0b | X | QI[Y] | Right |
| D | ab | $\phi$ | ? | ? | ? | a$\bar{b}$ | $\phi$ | $\phi$ | ? |
| D | ab | X | $\phi$ | $\phi$ | $\phi$ | a$\bar{b}$ | $\phi$ | $\phi$ | ? |
| D | a0 | X | QI[Y] | $\phi$ | $\phi$ | a1 | Y | D | Left |
| D | a1 | X | QI[Y] | $\phi$ | $\phi$ | a0 | Y | D | Right |
| D | a0 | X | SI[Y] | ? | ? | a1 | Y | D | Left |
| D | a1 | X | SI[Y] | ? | ? | a0 | Y | D | Right |
| D | a0 | X | Not SI | Z | ? | a1 | Z | D | Left |
| D | a1 | X | Not SI | ? | Z | a0 | Z | D | Right |

**TABLE 1. Output Restricted Deque Behavioral Description**

*: $\phi$ implies empty
"Not SI" implies either a QI, D or $\phi$



(a) INITIAL DATA DISTRIBUTION

(b) AFTER INSERT 3

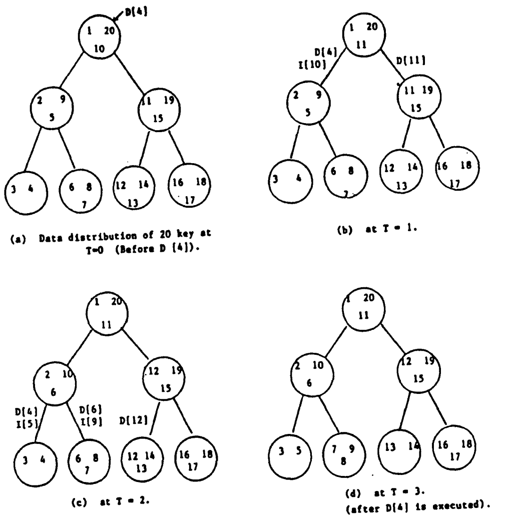
(c) AFTER INSERT 4

(d) AFTER INSERT 2

(e) AFTER INSERT 14

(f) AFTER INSERT 1

**Figure 3. Data distribution of the priority queue after successive inserts.**



(a) Data distribution of 20 key at T=0 (Before D [4]).

(b) at T = 1.

(c) at T = 2.

(d) at T = 3. (after D[4] is executed).

**Figure 4. Configuration of the dictionary machine before and after a deletion.**

# A COMPARATIVE STUDY OF TWO SYSTEMATIC DESIGN METHODOLOGIES FOR SYSTOLIC ARRAYS

*M. T. O'Keefe and J. A. B. Fortes* *
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

## Abstract

This paper identifies equivalences between two systematic methodologies for the design of systolic arrays and illustrates the benefits of understanding these relationships. The methods are the parameter method of Li and Wah and the dependency method of Moldovan and Fortes. After a review of the core ideas, models and parameters of each method, mathematical relations between them are derived. The usefulness of these relationships is illustrated by showing how (1) optimization procedures for the parameter method suggest similar procedures for the dependency method, and (2) new systolic equations for the parameter method result from the knowledge of equivalent equations in the dependency method. Also, systolic designs for convolution, obtained through different methods, can be mathematically proven to be identical [5].

## I - Introduction

In this paper, two proposed methodologies for the systematic design of systolic arrays are comparatively studied. They are the data dependency method of Moldovan and Fortes, [1]-[3], and the parameter method of Li and Wah [4]. We find and expose the recondite relationships and equivalences between the two methodologies and use this information to improve them and verify similar designs.

Section II provides a short description of both methodologies. Section III establishes equivalences between the mathematical expressions used to systematically design systolic arrays in the two methods. The equivalences of section III are used in section IV to propose optimization procedures and improvements for both methodologies.

## II - Introduction to the Parameter and Data Dependency Methods

### 2.1 Parameter Method [4]

This methodology considers the design of optimal pure planar systolic arrays for a class of linear recurrences which take the general form

$$z_{i,j}^{k} = f\left[z_{i,j}^{k+\delta}, x_{\hat{x}(i,k)}, y_{\hat{y}(k,j)}\right], \delta = \pm 1 \qquad (2.1.1)$$

where $f$ is the function to be executed by each cell of the array and $\hat{x}(i,k)$, $\hat{y}(k,j)$ are linear indexing functions for the two-dimensional input variables **X** and **Y**. In the following presentation, the coefficients of i,j, k are either 1 or -1. One-dimensional recurrences have the general form

$$z_{i}^{k} = f\left[z_{i}^{k+\delta}, x_{\hat{x}(i,k)}, a_{\hat{a}(k)}\right], \qquad \delta = \pm 1 \qquad (2.1.2)$$

Three sets of parameters are used to characterize a systolic array: velocities of data flow, data distributions, and periods of computation. The velocity of a datum $x$ is the directional distance passed by that datum in one clock cycle and is denoted by $\vec{x}_d$. The distance between two PEs is defined to be one. Thus, $\vec{x}_d$ must be less than or equal to one because broadcasting is not allowed in pure systolic arrays.

Data distributions are defined using row and column displacements. For two-dimensional input and output matrices, the elements along a row or column are arranged in a straight line and the distance between adjacent elements in a row or column remains constant as the data flows through the array. To define the row displacement of array **X**, suppose that the row and column indices of **X** are $i$ and $j$, respectively. The row displacement of **X** is the directional distance between $x_{\hat{x}(i,j)}$ and $x_{\hat{x}(i+1,j)}$ and is written as $\vec{x}_{is}$. Similarly, the column displacement is the distance between $x_{\hat{x}(i,j)}$ and $x_{\hat{x}(i,j+1)}$ and is written as $\vec{x}_{js}$.

Periods of computation are described using two functions, $\tau_c$ and $\tau_a$. $\tau_c$ is defined as the time at which a computation is performed, whereas $\tau_a$ defines the time at which a variable is accessed. The periods of $i$ and $j$ for two-dimensional outputs are defined as

$$t_i = \tau_c(z_{i+1,j}^{k}) - \tau_c(z_{i,j}^{k}) \qquad (2.1.3)$$

$$t_j = \tau_c(z_{i,j+1}^{k}) - \tau_c(z_{i,j}^{k}) \qquad (2.1.4)$$

$$t_k = \tau_c(z_{i,j}^{k+1}) - \tau_c(z_{i,j}^{k}) \qquad (2.1.5)$$

It will be assumed that $t_k$ is positive. If this is not true for a given recurrence, the recurrence can be rewritten to satisfy this condition. In computing $z_{i,j}^{k}$, $x_{\hat{x}(i,k)}$ and $y_{\hat{y}(k,j)}$ are accessed and two additional periods can be included to describe this interaction. They are

$$t_{kx} = \tau_a(x_{\hat{x}(i,k+1)}) - \tau_a(x_{\hat{x}(i,k)}) \qquad (2.1.6)$$

$$t_{ky} = \tau_a(y_{\hat{y}(k+1,j)}) - \tau_a(y_{\hat{y}(k,j)}) \qquad (2.1.7)$$

Depending on the order of access, $t_{kx}$ and $t_{ky}$ may be negative. Since operands to be used in a computation must arrive at a PE simultaneously, the magnitude of the periods must equal $t_k$, i.e. ,it must be true that

$$t_k = |t_{kx}| = |t_{ky}| \qquad (2.1.8)$$

The periods are independent of the indices $i$, $j$, and $k$, and they must be greater than or equal to one to prevent broadcasting.

These parameters (velocity, data distribution, and periods) can be combined into a set of equations which describe the operations of a systolic array. These equations, for the two-dimensional case, are

$$t_{kx}\vec{x}_d + \vec{x}_{ks} = t_{kx}\vec{z}_d \qquad (2.1.9)$$

$$t_{ky}\vec{y}_d + \vec{y}_{ks} = t_{ky}\vec{z}_d \qquad (2.1.10)$$

$$t_i\vec{x}_d + \vec{x}_{is} = t_i\vec{y}_d \qquad (2.1.11)$$

$$t_i\vec{z}_d + \vec{z}_{is} = t_i\vec{y}_d \qquad (2.1.12)$$

$$t_j\vec{y}_d + \vec{y}_{js} = t_j\vec{x}_d \qquad (2.1.13)$$

$$t_j\vec{z}_d + \vec{z}_{js} = t_j\vec{x}_d \qquad (2.1.14)$$

For the one-dimensional case, the last two equations are removed from the set of systolic equations given above.

* This work was supported in part by the National Science Foundation under Grant DMC-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and was administered through the Office of Naval Research under contract no. 00014-85-k-0588.

0190-3918/86/0000/0672 $01.00 © 1986 IEEE

The optimal systolic array for a given recurrence can be found by systematically enumerating the possible solutions using a search order that guarantees that the first feasible solution found is, in fact, the optimal one. Consider optimizing **T**, the total time needed to complete the computation. First set the magnitudes of the directional distance traversed by the variables equal to 1, i.e., $k_1 = k_2 = k_3 = 1$ [4]. Then, set the magnitudes of the periods $t_i$, $t_j$, and $t_k$ equal to one and determine if a feasible solution exists. If a feasible solution is found it is the optimal solution for **T** because **T** is a linear function of the periods that increases monotonically with increases in the magnitude of these periods. If no feasible solution is found with $t_i = t_j = t_k = 1$, one of the periods is increased by one and the search for a feasible solution repeats. If no feasible solution can be found with $k_1 = k_2 = k_3 = 1$, one of the $k_{i}$, $1 \leq i \leq 3$, is increased by one and the search begins again. A flowchart describing this optimality procedure is shown in [5]. A similar procedure is used to optimize the $\mathbf{AT}^2$ measure [4],[5].

## 2.2 Data Dependency Method [1]-[3]

The essence of the data dependency method is the representation of the dependency structure of an algorithm in concise, matricial form. That is, the dependencies between computations are condensed into an matrix that can be easily understood and manipulated.

Let $Z^n$ denote the nth cartesian power of Z, the set of nonnegative integers. To describe an algorithm **A**, a five tuple $\mathbf{A} = (\mathbf{J}^n, \mathbf{C}, \mathbf{D}, \mathbf{X}, \mathbf{Y})$ is used where $\mathbf{J}^n \subset \mathbf{Z}^n$ is the index set, **C** is the set of computations, **D** is the set of dependence vectors, **X** is the set of input variables, and **Y** is the set of output variables. The data dependencies describe the structure of the algorithm and are given as a set of triples $(\bar{d}, v, \bar{j})$ such that the computation indexed by $\bar{j}$ requires the variable v, generated at index $\bar{j} - \bar{d}$, as an operand.

Linear indexing functions [3] describe how variables are referenced. A linear indexing function $\bar{\mathbf{F}}: \mathbf{J}^n \rightarrow \mathbf{Z}^m$ is defined by an equation of the form $\bar{\mathbf{F}}(\bar{j}) = \bar{\mathbf{C}}_0 + \mathbf{C}\bar{j}$ where $\bar{\mathbf{C}}_0 \in \mathbf{Z}^{(m \times 1)}$ is called the index displacement and $\mathbf{C} \in \mathbf{Z}^{(m \times n)}$ is called the indexing matrix. For example, the variable $a(j_1 - j_2, j_2 + j_3 - 1, j_3 - j_1)$ has a linear indexing function for which

$$\mathbf{C} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \bar{\mathbf{C}}_0 = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} .$$

A transformation matrix **T** can be used to describe a linear bijection which transforms the dependency matrix and index set of an algorithm so that it can be executed in a VLSI array. **T** can be partitioned into two matrices, $\pi$ and S:

$$\mathbf{T} = \begin{bmatrix} \pi \\ \mathrm{S} \end{bmatrix}$$

The $\pi$ matrix defines the time transformation whereas S defines the space transformation to be applied to the dependence matrix and index set of an algorithm. The time at which a computation indexed by $\bar{j}$ is executed is determined by $\pi\bar{j}$, while $\mathrm{S}\bar{j}$ specifies which processor is to execute this computation. Of course, $\pi$ and S must satisfy certain conditions if they are to be considered valid transformations. Let *m* be the number of columns in the dependency matrix. Time transformations must satisfy $\pi\bar{d}_i > 0$, $i=1,2,\ldots,m$, where $\bar{d}_i$ is a column vector in the dependence matrix. This constraint results from the requirement that a variable must be generated before it is used in a computation.

The space transformation $\underline{\mathbf{S}}$ maps the computation indexed by $\bar{j}$ into processor $\mathrm{S}\bar{j}$. This assumes a processor array model consisting of a grid which has the dimensionality of the array Each point of the grid corresponds to a processor and the coordinates of the point are the index of the processor. Certain restrictions must be placed on possible solutions for **S** due to the limited interconnections available in VLSI arrays. These restrictions can be embodied in the **P** and **K** matrices. The **P** matrix describes the interconnection primitives available within an array, i.e., the vector differences between indices of connected processors. The utilization matrix **K** describes the interconnections used by the transformed algorithm during execution. The relationship between **K**, **P**, **S**, and **D** is

$$\mathbf{SD} = \mathbf{PK} \tag{2.2.1}$$

where the entries of **K** must satisfy the following constraint

$$\sum_{j=1}^{r} k_{ij} \leq \pi\bar{d}_i \ , \ i=1,\ldots,m \quad . \tag{2.2.2}$$

This last constraint requires that the time between the generation and use of a variable must be greater than or equal to the number of interconnection primitives needed by the datum to travel from the PE in which it was generated to the PE in which it will be used. Optimization procedures for the dependency method are given in [2],[3].

## III - Equivalences between the Parameter and Data Dependency Methods

*Lemmas* 1–3 provide equivalences between the different parameters of the two methods, while *Lemma* 4 describes the form of the dependency matrices for algorithms considered in the data dependency method. These lemmas are then applied in *Theorem* 1 to show that the space equations and systolic processing equations are equivalent. The proofs are omitted here but can be found in [5].

The first lemma gives expressions for the data distribution and velocity vectors of the parameter method in terms of the transformations and indexing matrices of the data dependency method.

*Lemma 1*

Let S, $\pi$ be as defined previously in section 2.2, and let *v* be any of the variables *x*, *y*, *z* as given for the parameter method. Also, let $\mathrm{C}^v$ represent the indexing matrix for variable *v*. Then the following relationships hold for the two-dimensional case:

$$\mathrm{S}\begin{bmatrix} \mathrm{C}^v \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} \pm1 \\ 0 \\ 0 \end{bmatrix} = \vec{v}_{is} \ , \ \mathrm{S}\begin{bmatrix} \mathrm{C}^v \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ \pm1 \\ 0 \end{bmatrix} = \vec{v}_{js} \ ,$$

$$\mathrm{S}\begin{bmatrix} \mathrm{C}^v \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ \pm1 \end{bmatrix} = \vec{v}_d$$

For the one-dimensional case, the following relationships apply:

$$\mathrm{S}\begin{bmatrix} \mathrm{C}^v \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} \pm1 \\ 0 \end{bmatrix} = \vec{v}_s \ , \ \mathrm{S}\begin{bmatrix} \mathrm{C}^v \\ \pi \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ \pm1 \end{bmatrix} = \vec{v}_d$$

The following lemma describes the relationship between the $\pi$ vector of the data dependency method and the periods $t_i$, $t_j$, $t_k$ of the parameter method. The relationship is remarkably simple.

*Lemma 2*

$$\pi = \begin{bmatrix} t_i & t_j & t_k \end{bmatrix}$$

Thus, the periods of the parameter method are the elements of the $\pi$ matrix. The next lemma relates the elements of the data dependency method's **K** matrix and the constants, $k_i$ $(1 \leq i \leq 3)$, as defined in [4]; i.e., $|t_k| \, |\vec{z}_d| = k_1$ , $|t_i| \, |\vec{y}_d| = k_2$ , $|t_j| \, |\vec{x}_d| = k_3$ . Let $\hat{k}_i$ be the single nonzero entry of the i'th column of **K**.

*Lemma 3*

$$\hat{k}_i = k_i \qquad 1 \leq i \leq 3$$

The next lemma describes the form of the dependency matrices for the class of recurrences considered in the parameter method.

*Lemma 4*

The dependency matrices for the class of recurrences considered in the parameter method have the following structure:

**Two-dimensional Recurrence:**

$$\mathbf{D} = \left[\begin{array}{ccc|c} \pm 1 & 0 & 0 & \\ 0 & \pm 1 & 0 & \bar{d}_1 \ldots \bar{d}_r \\ 0 & 0 & \pm 1 & \end{array}\right]$$

**One-dimensional Recurrence:**

$$\mathbf{D} = \left[\begin{array}{cc|c|c} \pm 1 & 0 & c_1 & \\ 0 & \pm 1 & c_2 & \bar{d}_1 \ldots \bar{d}_r \end{array}\right] , \; | c_1 | = | c_2 | = 1$$

where $\bar{\mathbf{d}}_1, \ldots, \bar{\mathbf{d}}_r$ are dependency vectory which are a function of the recurrence, as is r, the total number of these additional dependencies.

The following theorem states that the equations used in both methods to describe the operation of a systolic array are equivalent.

*Theorem 1*

The constraint equations (2.1.9 - 2.1.14) of the parameter method are equivalent to the space equations, **SD = PK**, of the data dependency method.

## IV - Optimization Procedures and Examples

### 4.1 Optimization Procedures

Optimization procedures for the parameter method were discussed previously in section II. By directly translating the parameters and constraints of this method into the corresponding elements of the dependency method, we can devise a similar procedure which is applicable to the recurrences considered in [4]. However, by using a slightly different approach, it is possible to propose a related optimization procedure applicable to all cases for which disp$\pi$=1 in the dependency method. Note that disp$\pi$ refers to the number of parallel wavefronts in the array. It differs from that proposed for the parameter method in that it checks all possible values of K before considering longer execution times (i.e., different $\pi$'s). The flowchart of Figure 1 describes the new optimization procedure. In words, it starts by finding all transformations $\pi$ which minimize execution time. This is relatively easy, since only the case disp$\pi$=1 is considered and execution time is therefore a monotonic function of the entries of $\pi$. Hence, one can start with all entries of $\pi$ being zero and progressively increase their absolute values considering all possible combinations of signs and magnitudes (while, of course, checking for the validity of each $\pi$). Possible $\pi$'s, which might result from further increases in the absolute value of the entries of a particular $\pi$ for which execution time is larger than the known minimum, need not be considered due to monotonicity property mentioned above. Thus, the search space is finite, and, in fact, rather small for most cases. Once the set of $\pi$'s is known, it is necessary to check if there exists a solution to the equation SD=PK for at least one of the possible values of K. If a solution is found, then the corresponding $\pi$ (as well as the design determined by $\pi$ and S) is optimal with respect to execution time. Otherwise, a new set of $\pi$'s must be found which increase execution time by the least amount and the process is repeated again. The procedure always terminates, since, in the worst case, serial execution is reached as a feasible solution.

A similar reasoning can be used to optimize measures combining area and execution time, e.g., AT or $AT^2$. Figure 2 illustrates such a procedure. It differs from that of figure 2 in that the search space is reduced to the set of $\pi$'s which result in execution time bounded above by a constant factor [4]. In this finite space, all valid values of $\pi$ and S are considered and those which optimize the combined measure of area and time determine the optimal solution. This is exactly the same approach used in the parameter method. The key idea consists of limiting the search space by choosing bounds for $\pi$ and, thus, for the execution time.

### 4.2 New systolic equations

Convolution can be expressed as the following recurrence equation

$$y_i^0 = 0 \quad 1 \leq i \leq n$$

$$y_i^k = y_i^{k-1} + a_k x_{i+k-1} \quad 1 \leq i \leq n,$$

$$1 \leq k \leq m, \; x_j = 0 \text{ for } j > n \tag{4.1}$$

Another possible description with the order of access of the input terms reversed is

$$y_i^k = y_i^{k-1} + a_{m-k+1} x_{m-k+i} \quad 1 \leq i \leq n,$$

$$1 \leq k \leq m, \; n > m \tag{4.2}$$

*Theorem 1* showed that the systolic equations of the parameter method are equivalent to the space equations, **SD = PK**, of the data dependency method. Systolic equations for the one-dimensional case are equivalent to $\mathbf{S\bar{d}_y = P\bar{k}_y}$ and $\mathbf{S\bar{d}_a = P\bar{k}_a}$, respectively. The subscripts on the vectors $\bar{\mathbf{k}}$ and $\bar{\mathbf{d}}$ indicate which variable is associated with a particular vector. Thus, the $\mathbf{S\bar{d}_x = P\bar{k}_x}$ space equation is not contained within the systolic equations of the parameter method for the one-dimensional case.

The systolic equations for the $x$ dependency will take the general form

$$f_a \vec{\mathbf{a}}_d + \vec{\mathbf{a}}_s = f_a \vec{\mathbf{x}}_d \tag{4.3}$$

$$f_y \vec{\mathbf{y}}_d + \vec{\mathbf{y}}_s = f_y \vec{\mathbf{x}}_d \tag{4.4}$$

where $f_a$ and $f_y$ are linear functions of $t_i$ and $t_k$. To determine the functions $f_a$ , $f_y$ , the equivalences defined in *Lemmas* 1 and 2 are applied to (4.3-4.4) resulting in $f_a = -(t_i + t_k)$ , $f_y = (t_i + t_k)$ and these equations become

$$-(t_i + t_k)\vec{\mathbf{a}}_d + \vec{\mathbf{a}}_s = -(t_i + t_k)\vec{\mathbf{x}}_d$$

$$(t_i + t_k)\vec{\mathbf{y}}_d + \vec{\mathbf{y}}_s = (t_k + t_i)\vec{\mathbf{x}}_d$$

The above analysis was applied to the recurrence of equation (4.2). A similar analysis of the recurrence expressed in equation (4.1) results in $f_x = (t_k - t_i)$ and $f_y = -(t_k - t_i)$ in equations (4.3) and (4.4). This new set of systolic equations, derived from the data dependency method through the equivalences described in Section III, can be added to the set of systolic equations. From this new set, only four equations are needed to provide equivalent solutions to those derived from the original four equations.

### References

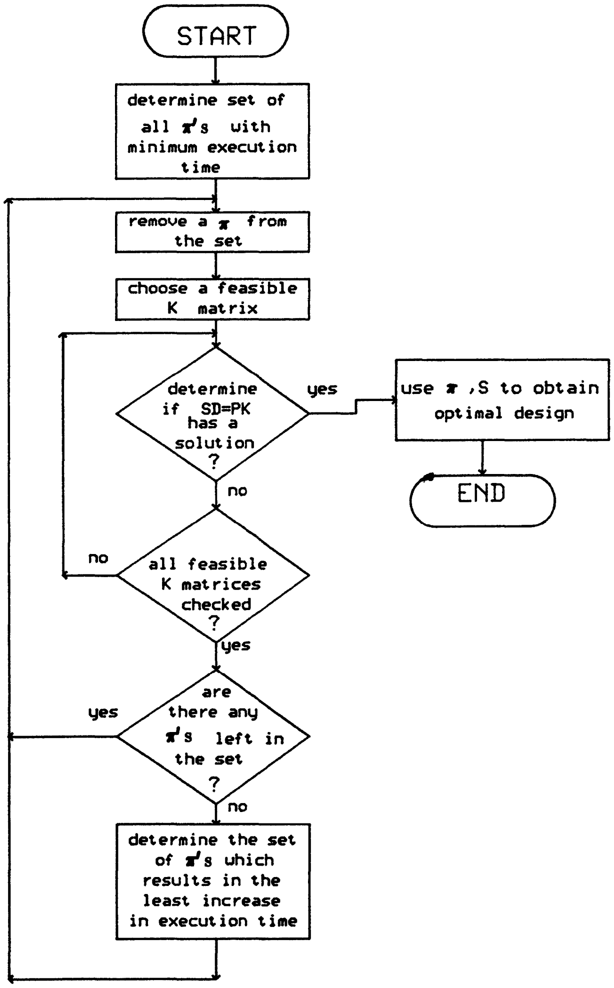
[1] D.I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," Proc. IEEE, Vol. 71, No. 1, Jan. 1983.

[2] J.A.B. Fortes, "Algorithm Transformation for Parallel Processing and VLSI Architecture Design," Ph.D. dissertation, University of Southern California, L.A., CA. Dec. 1983.

[3] J.A.B. Fortes and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," 1984 Int. Conf. Parallel Processing, 1984.

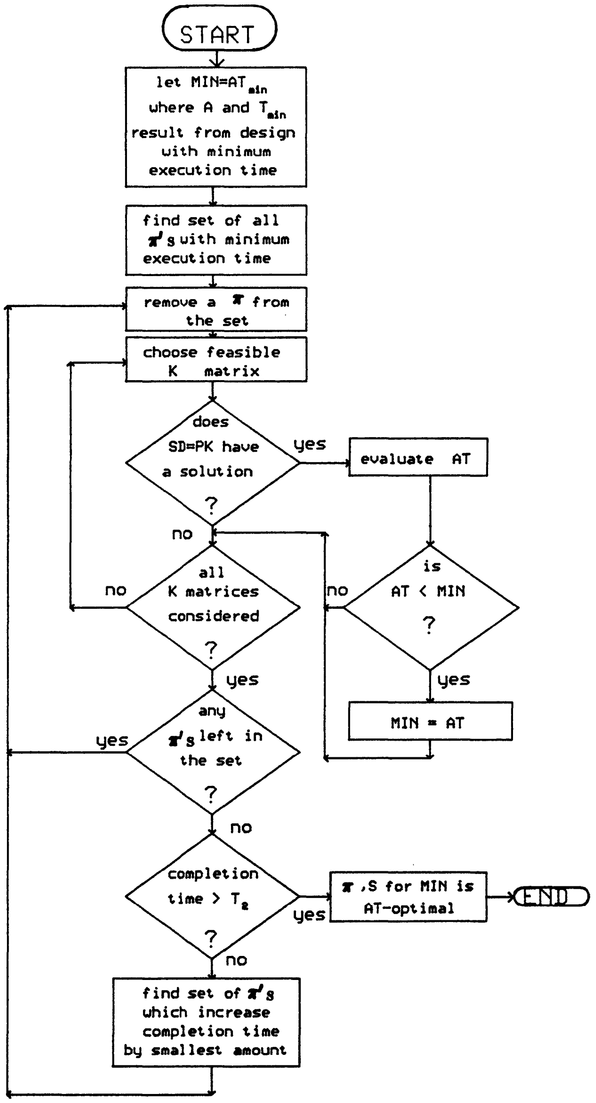
[4] G. J. Li and B. W. Wah, "The Design of Optimal Systolic Arrays," IEEE Transactions on Computers, Vol. C-34, pp.66-77, Jan. 1985.

[5] M. O'Keefe, "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays," MSEE Thesis, School of Electrical Engineering, Purdue University, 1986.



START
determine set of all π's with minimum execution time
remove a π from the set
choose a feasible K matrix
determine if SD=PK has a solution ?
yes
use π ,S to obtain optimal design
END
no
all feasible K matrices checked ?
no
yes
are there any π's left in the set ?
yes
no
determine the set of π's which results in the least increase in execution time

Figure 1 – Optimization procedure for obtaining a systolic array design with minimum execution time (T) using the dependency method.



START
let MIN=AT_min where A and T_min result from design with minimum execution time
find set of all π's with minimum execution time
remove a π from the set
choose feasible K matrix
does SD=PK have a solution ?
yes
evaluate AT
no
all K matrices considered ?
no
is AT < MIN ?
no
yes
MIN = AT
yes
any π's left in the set ?
yes
no
completion time > T_s ?
yes
π ,S for MIN is AT-optimal
END
no
find set of π's which increase completion time by smallest amount

Figure 2 – Optimization procedure for obtaining a systolic array design with minimum area.execution time (AT) using the dependency method.

# SOLVING MATRIX PROBLEMS WITH NO SIZE RESTRICTION ON A SYSTOLIC ARRAY PROCESSOR

Juan J. Navarro
José M. LLaberia
Mateo Valero

Departmento de Arquitectura de Computadores
Facultad de Informática (UPC) Pau Gargallo 5.
08028 BARCELONA (SPAIN)

## ABSTRACT

In this paper we propose several data structures partitioning and transformation schemes, in order to get an efficient execution of various matrix algorithms without any size restriction. The following matrix operations are considered:

- Matrix-Matrix multiplication
- Triangular matrix equations.
- L-U decomposition and
- Inverses of triangular and dense matrices.

All these algorithms are to be executed on a problem-size independent spiral systolic array processor. The array topology is fixed, and a simple feedback and control are needed. For all the algorithms that have been considered, the PE's utilization tends to the maximum possible value.

## 1. INTRODUCTION

With the advent of VLSI, cheap processors can be nowadays designed to solve at very high speed one subset of numerical or symbolic applications. An interesting class of systems are the systolic array processors (SAP) /1/,/2/.

In a strict sense, a SAP allows the hardware implementation of an algorithm which settles the functions to be performed in every PE, the number of PEs and the system topology. Many systolic algorithm implementations have been proposed: /2/, /3/. Nevertheless, a clear trend exists towards the realization of SAP connected to a host, in order to execute several algorithms.

Two important problems arise when this kind of processors is considered: the PEs operation and sometimes the interconnection topology must be programmable, and computations on different sized data structures must be executed in a fixed number of PEs. In order to get the maximum efficiency of the global system, the partitioning algorithms must impose the lower overhead possible with respect to the execution time spent by the complete problem, the extra control needed to modify the PEs functionality, their interconnections, and the access from the boundaries PE to the data structures. To evaluate the system efficiency, the utilization factor of the systolic array, or the number of time units (T) needed to execute the algorithm in the system can be used.

In this paper, some data structures partitioning and transformation schemes are proposed, in order to get an efficient execution of various matrix algorithms, with no size restriction, on a fixed dimension single array processor. All the algorithms are to be executed on a contraflow hexagonal SAP with w-by-w PEs (see fig. 1). The array topology includes connections to recirculate and feedbacking of PEs with partial results (SAP with this interconnection topology has been named spiral SAP, (SSAP) /4/). Some of the boundaring PEs must be able of executing several functions as those to be described in section 2.

The notation to be used in the rest of this paper is explained in the following lines:

- Capital letters A, B, C, D, X, Y,... denote matrices and capital letters N, M, P,... the dimension of those matrices.
- Without any loss of generality, the matrices dimensions are supposed to be multiples of w. When this assumption is not true, we extend the matrix with

This work was supported by Ministery of Education of Spain (CAICYT) under Grant Number 2906-83 C03-03 and by CTNE.

0190-3918/86/0000/0676 $01.00 © 1986 IEEE

a sufficient and minimum number of rows and/or columns formed with zero-valued elements. Accordingly, we define: $\bar{N} = N/w$, $\bar{M} = M/w$, $\bar{P} = P/w$.

- Regarding the partitioning, if A is an $\bar{N}$-by-$\bar{M}$ matrix, we shall consider it as a $\bar{N}$-by-$\bar{M}$ block matrix. And now $A_{i,j}$ is the (i,j) block with w-by-w elements.
- The following submatrices of A are defined:
  - $A^k_{i,*}=(A_{i,1},A_{i,2},\dots A_{i,k})$ for $1 \le k \le \bar{M}$ and $1 \le i \le \bar{N}$. $A^k_{i,*}$ is a block-row matrix with w-by-kw elements.
  - $A^k_{*,j}=(A_{1,j},A_{2,j},\dots A_{k,j})^T$ for $1 \le k \le \bar{N}$ and $1 \le j \le \bar{M}$. $A^k_{*,j}$ is a block-column matrix with kw-by-w elements.
  - $A^k_{*,*}=(A^k_{1,*},A^k_{2,*},\dots,A^k_{k,*})^T= =(A^k_{*,1},A^k_{*,2},\dots A^k_{*,k})$ for $1 \le k \le \min(\bar{N},\bar{M})$. $A^k_{*,*}$ is a k-by-k block matrix.

Every $A_{i,j}$ is, in turn, decomposed into the following matrices:

$$A_{i,j} = A_{Li,j} + A_{Di,j} + A_{Ui,j} = \\ = A_{LDi,j} + A_{Ui,j} = A_{Li,j} + A_{DUi,j}$$

where: $A_L$ ($A_U$) are strictly lower (upper) triangular matrices, $A_D$ is a diagonal matrix, and $A_{LD}$ ($A_{DU}$) are lower (upper) triangular matrices.

The outline of this paper is as follows: in section 2 the concepts of partitioning and transformation of a dense problem into an equivalent band problem is introduced; in section 3 the problem D=A.B+C is solved, and the solution presented there is reused in subsequent sections;insection 4 the methodology to solve triangular matrix equations is applied; in section 5 the LU decomposition without pivoting is solved; and in section 6 operation of inverting triangular matrices and $A^{-1}=U^{-1}.L^{-1}$, is presented. Finally, in section 7 the main conclusions of this work are summarized.

## 2. PARTITIONING AND DATA RESTRUCTURATION

Several partitioning algorithms of matrix problems have been recently proposed. These algorithms are intended to work on SIMD array processors, pipeline vector supercomputers /5/, Modular VLSI arithmetic systems /6/, or systolic array processors /7/, /8/.

It is imperative to attain a good matching between the resulting partitioned submatrices and the array processing system, because the maximum utilization of the SAP PEs is desired. The data restructuration serves to define the sequence to chain the execution of subproblems in a SAP. This function must ensure the correctness of the solution and must allow a simple control and a maximum PEs utilization. Overlapping of a subproblem loading with the unloading of the precedent, is an adequate feature to maximize the system utilization.

In our work a contraflow SAP is used. This SAP constitute a good solution to problems requiring the computation of recurrences (e.g. a linear triangular system solving, LU decomposition... /9/).

A triangular partitioning, and various types of data restructurations to transform dense to band matrices, are proposed in /10/. Those transformations yield to maximum efficiency of the PEs utilization. Such a methodology is applied to matrix-matrix and matrix-vector multiplications.

To achieve this goal, each N-by-M matrix A is partitioned into $\bar{N}$-by-$\bar{M}$ block matrices. Each w-by-w block matrix $A_{i,j}$ is split into two triangular matrices $A_{LDi,j}$ and $A_{Ui,j}$ (or $A_{Li,j}$ and $A_{UDi,j}$). The optimal restructuration is attained by juxtaposing these triangular submatrices, by means of different algorithms, in order to obtain a $\bar{N}\bar{M}$+1-by-$\bar{N}\bar{M}$ ($\bar{N}\bar{M}$-by-$\bar{N}\bar{M}$+1) block lower (or upper) band matrix, $\bar{A}$. Maximal efficiency is achieved because the $\bar{A}$ band contains only elements from the original matrix, A, with no empty positions.

When problems that include inner product operations are appointed, several considerations to obtain $\bar{A}$ must be taken into account.

a) For $1 \le k \le \bar{N}\bar{M}$, if $\bar{A}_{kk}=A_{LDi,j}$, then $\bar{A}_{k+1,k}$ must be equal to $A_{Up,j}$ for any p such that $1 \le p \le \bar{N}$.

b) For $2 \le k \le \bar{N}\bar{M}$, if $\bar{A}_{k,k-1}=A_{Ui,j}$ then $\bar{A}_{k,k}$ must be equal to $A_{LDi,p}$ for any p such that $1 \le p \le \bar{M}$.

c) Obviously, dependencies in the computational sequence needed by some algorithms must be respected.

We shall name this type of transformations as Dense to Band Matrix Transformation by Triangular block Partitioning, (DBT).

The PRT /11/ and PCT /12/ transformations, proposed by R.W. Priester et al., can be regarded as particular cases of the DBT. The transformation of a N-by-N dense matrix into a band matrix with bandwith w=N is proposed in the mentioned papers; they attain a 50% reduction of the number of PEs in the SAP, with no overhead in the algorithm time under certain conditions.

Our DBT method serves to transform all the algorithms mentioned above, and the obtained algorithms are to be executed on a single hexagonal SSAP with w-by-w PEs (fig. 1). The feedbacking needed to recirculate all partial results introduces a delay of w cycles, except in the case of the main diagonal which is 2w cycles. As the SSAP must execute various algorithms, some PEs must be able of executing several operations, according to the precise algorithm and the precise cycle. In figure 2, the elementary processor is shown, as well as the types of operations needed, named OP(A), OP(B)etc.. The PEs(i,j), $2\leq i,j\leq w$, perform only OP(A), which corresponds to the type of inner product step processor /19/. The PEs(1,j), with $2\leq j\leq w$, must perform OP(B) and OP(D). The PEs(i,1), with $2\leq i\leq w$, must perform OP(A) and OP(F). Finally, the PE(1,1) performs all types of operation.

## 3. MATRIX-MATRIX MULTIPLICATION

We consider now the operation:

$$D := A \cdot B + C \qquad (1)$$

to be performed on a w-by-w systolic array. When A,B and C are M-by-N, N-by-P and M-by-P matrices, the problem is partitioned into $\bar{M}\bar{P}$ disjoint subproblems, according to the algorithm:

```
Algorithm 1:
For m:= 1 to $\bar{M}$ do
    For p:= 1 to $\bar{P}$ do
        $D_{m,p} := A^{\bar{N}}_{m,*} B^{\bar{N}}_{*,p} + C_{m,p}$        (2)
```

Each one of the subproblems in (2) is solved sequentially by the SAP. Every subproblem is of the type:

$$D := A \cdot B + C \qquad (3)$$

where A,B and C are, respectively, w-by-N, N-by-w and w-by-w matrices. We focus our attention on the execution of one of these subproblems.

By means of DBT transformations on the dense matrices A and B, the problem (3) is mapped onto a banded problem (see fig. 3)

$$\bar{D} := \bar{A} \cdot \bar{B} + \bar{C} \qquad (4)$$

To define the transformed problem, partitioning of $A=(A_1 A_2 \ldots A_j \ldots A_{\bar{N}})$ and $B=(B_1 B_2 \ldots B_i \ldots B_{\bar{N}})^T$ into w-by-w triangular submatrices is required, as follows:

$A_j = A_{LDj} + A_{Uj} \quad 1\leq j\leq \bar{N}$

$B_i = B_{DUi} + B_{Li} \quad 1\leq i\leq \bar{N}$

Applying one DBT to A, $\bar{A}$ is obtained. $\bar{A}$ is a $(\bar{N}+1)$-by-$\bar{N}$ block lower two-diagonal matrix, with w-by-w blocks $\bar{A}_{i,j}$:

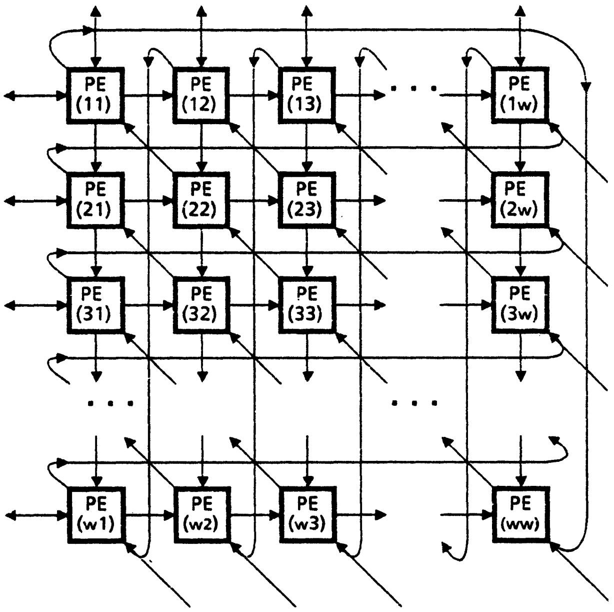
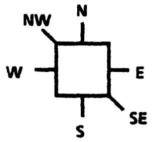


Fig. 1. Spiral Systolic Array Processor.



| OP (A) | OP (B) |
|---|---|
| NW←SE+N*W<br>S←N<br>E←W | N←SE*W<br>NW←0<br>S←-SE*W<br>E←W |

| OP (C) | OP (D) | OP (E) | OP (F) | OP (G) |
|---|---|---|---|---|
| N←SE/W<br>NW←0<br>S←-SE/W<br>E←1/W | NW←SE-N*W<br>S←-N<br>E←W | N←SE<br>W←1<br>S←1/SE<br>E←1 | NW←0<br>W←SE*N<br>S←N<br>E←SE*N | NW←0<br>W←SE/N<br>S←1/N<br>E←SE/N |

Fig. 2. PE's ports and operations.

for $1 \leq i \leq \bar{N}+1$ and $1 \leq j \leq \bar{N}$

$$\bar{A}_{i,j} = \begin{cases} A_{LDj} & \text{if } i=j \\ A_{Uj} & \text{if } i=j+1 \\ 0 & \text{if } i>j+1 \text{ or } i<j \end{cases}$$

In a similar way, we define another DBT on B, yielding $\bar{B}$ which is a $\bar{N}$-by-$(\bar{N}+1)$ block upper two-diagonal matrix with blocks $\bar{B}_{i,j}$.

for $1 \leq i \leq \bar{N}$ and $1 \leq j \leq \bar{N}+1$

$$\bar{B}_{i,j} = \begin{cases} B_{DUi} & \text{if } j=i \\ B_{Li} & \text{if } j=i+1 \\ 0 & \text{if } j>i+1 \text{ or } i<j \end{cases}$$

We define now $\bar{C}$ as a $(\bar{N}+1)$-by-$(\bar{N}+1)$ block tridiagonal matrix:

$$\bar{C}_{i,j} = \begin{cases} C \text{ if } i=j=1 \\ 0 \text{ otherwise} \end{cases} \quad \text{for } 1 \leq i,j \leq \bar{N}+1$$

Then $\bar{D}$ is a $(\bar{N}+1)$-by-$(\bar{N}+1)$ block tridiagonal matrix.

$\bar{D}_{1,1} := A_{LD1} \cdot B_{DU1} + C$

for $2 \leq k \leq \bar{N}$

$\bar{D}_{k,k} := A_{U(k-1)} \cdot B_{L(k-1)} + A_{LDk} \cdot B_{DUk}$

$\bar{D}_{\bar{N}+1,\bar{N}+1} := A_{U\bar{N}} \cdot B_{L\bar{N}}$

$\bar{D}_{Lk,k+1} := A_{LDk} \cdot B_{Lk}$ for $1 \leq k \leq \bar{N}$

$\bar{D}_{Uk+1,k} := A_{Uk} \cdot B_{DUk}$ for $1 \leq k \leq \bar{N}$

$\bar{D}_{i,j} := 0$ for $j<i-1$ or$j>i+1$

Figure 3.a shows the triangular blocks decomposition for problem (3). Figure 3.b shows the resulting band problem (4). Matrix $\bar{D}$ can be efficiently obtained on a hexagonal array with w-by-w PEs, similar to the one proposed by H.T. Kung in /19/. This can be accomplished in the w-by-w SSAP, by programming all the PEs with OP(A). The D matrix is the result of the original problem, and can be found from $\bar{D}$ by means of the following computation:

$$D = \sum_{k=1}^{\bar{N}} (\bar{D}_{k,k} + \bar{D}_{Lk,k+1} + \bar{D}_{Uk+1,k}) + \bar{D}_{\bar{N}+1,\bar{N}+1}$$

This computation may be performed inside the array by using the spiral systolic array feedbacks, causing no calculation time overhead.

To specify the required feedback, we define two matrices: I (input) and O (output), which are $(\bar{N}+1)$-by-$(\bar{N}+1)$ tridiagonal block matrices. I contains data and partial results which must come in through the SE ports of the South and East PEs. O contains the partial results coming out through the NW ports of the North and West PEs.

We have, consequently, the following inputs to the array:

$I_{1,1} := C$

for $2 \leq i \leq \bar{N}+1$

$$I_{Li,i} := O_{Li-1,i} = \sum_{k=1}^{i-1} \bar{D}_{Lk,k} + \sum_{k=1}^{i-1} \bar{D}_{Lk,k+1}$$

$$I_{Ui,i} := O_{Ui,i-1} = \sum_{k=1}^{i-1} \bar{D}_{Uk,k} + \sum_{k=1}^{i-1} \bar{D}_{Uk+1,k}$$

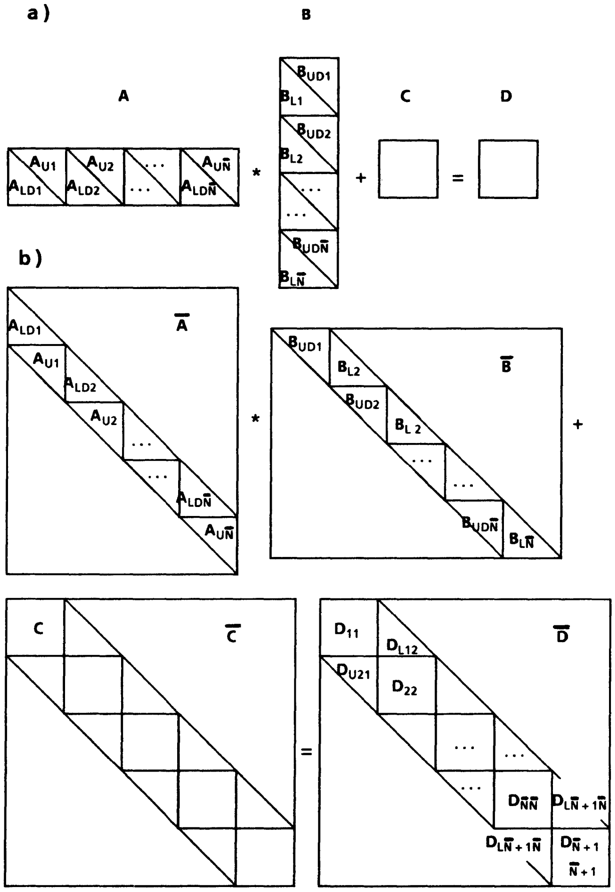


Fig.3. a) Particioning D = A*B + C in triangular submatrices
b) Band problem originated through a DBT transformation

$$I_{Di,i} := O_{Di-1,i-1} = \sum_{k=1}^{i-1} \bar{D}_{Dk,k}$$

for $1 \leq i \leq \bar{N}$

$$I_{Li,i+1} := O_{Li,i} = \sum_{k=1}^{i} \bar{D}_{Lk,k} + \sum_{k=1}^{i-1} \bar{D}_{Lk,k+1}$$

$$I_{Ui+1,i} := O_{Ui,i} = \sum_{k=1}^{i} \bar{D}_{Uk,k} + \sum_{k=1}^{i-1} \bar{D}_{Uk+1,k}$$

And the last output from the array is the desired result for (3):

$$O_{U\bar{N}+1,\bar{N}+1} := \sum_{k=1}^{\bar{N}+1} \bar{D}_{kk} + \sum_{k=1}^{\bar{N}} (\bar{D}_{Lk,k+1} + \bar{D}_{Uk+1,k}) = D$$

Remark that the elements of the output submatrices, $O_U$ or $O_L$, are required in the SE input ports w cycles later than their appearance at the output NW ports. The elements of $O_D$ must be delayed by 2w cycles, instead.

To compute D:=AB+C, when A and B are w-by-N and N-by-w matrices, in the SSAP, the elapsed time is T = 3N+4w. When A and B are M-by-N and N-by-P matrices, respectively, the resulting $\overline{MP}$ subproblems must be chained, originating a computation elapsed time: $T = 3MPN/w^2 +$ $+ 3MP/w+w$. The value of T can be diminished by $3(MP-w^2)/w$ time units if the DBT algorithm presented in /10/ is used, because a total chaining of the $\overline{MP}$ subproblems is attained.

## 4.TRIANGULAR MATRIX EQUATIONS

One of the basic problems in linear Algebra is the solving of a linear system of equations. A matrix equation

$$AX = B \qquad (5)$$

is a multiple set of linear systems, all working with the same matrix, A, and different column-vectors. If the main submatrices of A are all non singular, then there exist a unique lower-triangular matrix L with $l_{kk}=1$, and a unique upper-triangular matrix so that A=LU /24/. As a consequence, (5) is equivalent to LUX=B, yielding to a decomposition into two triangular matrix equations:

$$LY = B \qquad (6)$$
and
$$UX = Y \qquad (7)$$

In this section we consider the solution of the lower triangular matrix equation (6). The solving of equation (7) shall be briefly commented at the end of this section, as it resembles the solution of (6). Let us consider that L is a N-by-N lower triangular matrix, and Y, B are N-by-M matrices. The problem is first partitioned into $\bar{M}$ disjoint subproblems, according to the following algorithm:

**Algorithm 2:**

```
For m:=1 to $\bar{M}$ do
Compute $Y^{\bar{N}}_{*,m}$ from: $B^{\bar{N}}_{*,m} - L\,Y^{\bar{N}}_{*,m} = 0$   (8)
```

Each subproblem (8) is solved by the array in a sequently fashion. We focus our attention, now, in the resolution of one of these subproblems, e.g. the first one (m:=1)

$$B^{\bar{N}}_{*,1} - L\,Y^{\bar{N}}_{*,1} = 0 \qquad (9)$$

The direct solution of (9), by forward substitution, is again decomposed as follows:

**Algorithm 3:**

```
Compute $Y_{1,1}$ from: $B_{1,1} - L_{1,1}\,Y_{1,1} = 0$   (10)
for n:=2 to $\bar{N}$ do
$B_{n,1} := B_{n,1} - L^{n-1}_{n,*}\,Y^{n-1}_{*,1}$   (11)
Compute $Y_{n,1}$ from: $B_{n,1} - L_{n,n}\,Y_{n,1} = 0$   (12)
```

The submatrices, $Y_{n,1}$ with $1 \leq n \leq \bar{N}$, appearing in (10) and (12) must satisfy an equation as:

$$B - LY = 0 \qquad (13)$$

where L, Y and B are w-by-w matrices.

To find the Y matrix satisfying (13) using the SSAP, the problem (13) must be decomposed into subproblems with triangular matrices structures:

$$B_{DU} - (LY_{DU})_{DU} = 0 \qquad (14)$$

$$B_L - (LY_{DU})_L - LY_L = 0 \qquad (15)$$

If the boundary PEs are programmed in a way such that a) PE(1,1) performs OP(C) (see fig. 2), b) PEs (1,j) $2 \leq j \leq w$ perform OP(B) and c) the rest perform OP(A), the SSAP can solve equations of the type

$$\bar{B} - \bar{L}\,\bar{Y} = \bar{C} \qquad (16)$$

where $\bar{L}$ is a w-by-w lower triangular matrix, $\bar{Y}$ is a w-by-2w upper band matrix with bandwidth w, and $\bar{B}$ and $\bar{C}$ are w-by-2w band matrix with bandwidth 2w-1. In order to get the desired solution, $\bar{L}$

must be input through the $\bar{W}$, ports of the west boundary PEs, and $\bar{B}$ through the SE ports of the south and west PEs. The result $\bar{Y}$ is obtained at the ports of the north PEs, and the following equations are satisfied: $\bar{B}_{DU1,1} - (\bar{L}\,\bar{Y}_{1,1})_{DU} = 0$ and $\bar{B}_{L1,2} - \bar{L}\,\bar{Y}_{1,2} = 0$. Besides, matrix $\bar{C}$ is obtained at the NW ports of the North and West PEs, with: $\bar{C}_{DU1,1} = = \bar{C}_{L1,2} = 0$ and $\bar{C}_{L1,1} = \bar{B}_{L1,1} - (\bar{L}\bar{Y}_{1,1})_L$. By making $\bar{L} = L$ and $\bar{B}_{1,1} = \bar{B}$, $Y_{DU}$ can be obtained to satisfy (14), and $\bar{Y}_{1,1}$, $\bar{C}_{L1,1}$ are attained as: $\bar{Y}_{1,1} = Y_{DU}$ and $\bar{C}_{L1,1} = B_L - (L\,Y_{DU})_L$. If, in addition, $\bar{B}_{L1,2} = \bar{C}_{L1,1}$, the $\bar{Y}_{1,2}$ resulting submatrix is equal to the $Y_L$ submatrix that satisfies (15).

To complete the solution of problem (9) according to algorithm 3, the subproblems (11) must be solved. These subproblems consist in a block-row matrix $(L^{n-1}_{n,*})$ by block-column matrix $(-Y^{n-1}_{1,*})$ multiplication with accumulation of $B_{1,n}$. Each one of these subproblems can be solved in the array just like has been described in section 3, except that now the PEs(1,j) $1\leq j\leq w$ must perform OP(D) to change the sign of the previously calculated matrix, $Y^{n-1}_{1,*}$.

The chaining of the subproblems to be executed in the SSAP, according to algorithm 3, is shown in fig. 4. This figure shown the I/O block-data stream at the W, N, SE ports of the West, North, South and East PEs, respectively, for the case $\bar{N}$=3. The blocks labeled as f indicate that the corresponding data come from the feedback links.

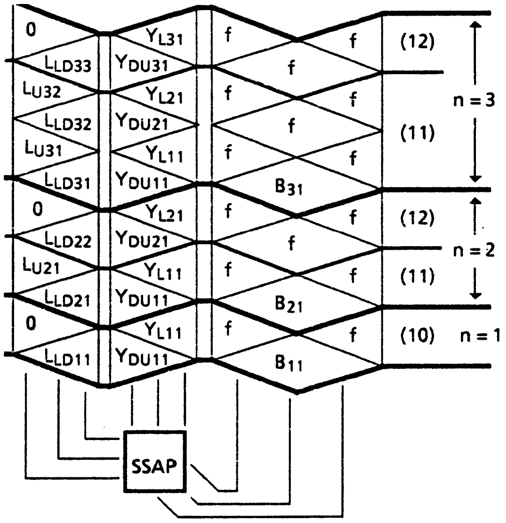


Fig.4. I/O block - data stream to solve matrix equations (with $\bar{N}=3$).

The computational time of one subproblem (8) is

$$T = 3N^2/2w + 3N/2 + w \tag{17}$$

The computational time of the $\bar{M}$ subproblems needed to solve the matrix equation LY=B where L is a N-by-N lower triangular matrix, and Y, B are N-by-M matrices is

$$T = 3MN^2/2w^2 + 3MN/2w + w \tag{18}$$

The problem (7) is solved by back substitution, that is through reordering of matrices and solution by forward substitution of the problem

$$X'\,U' = Y' \tag{19}$$

where X' and Y' are block-row matrices with $x'_i = x_{N-i+1}$ for $1\leq i\leq N$ and $y'_i = = y_{N-i+1}$ for $1\leq i\leq N$; and U' is upper triangular, with $u'_{i,j} = u_{N-j+1,N-i+1}$ for $1\leq i,j\leq N$.

If equation (19) is transposed, we arrive to the problem

$$U'^T\,X'^T = Y'^T \tag{20}$$

which is of type of problem (6), previously analyzed in this same section.

Normally, problem (7) is solved after performing a LU decomposition. In this case, as will be shown in the next section, the final results of matrix U are obtained at the North boundary PEs. Consequently, in the execution of (20) we shall consider that matrix $U'^T$ is available at the North boundary PEs, and $X'^T$ is obtained at the West boundary PEs. In order to get the solution, in those cycles where the PE(1,1) performed OP(C), the OP(G) must be performed now; and in the cycle where PEs (1,j) $2\leq j\leq w$ performed OP(B), now the PEs $(\bar{i},\bar{1})$ $2\leq i\leq w$ must perform the OP(F), instead. The value of T for problem (7) are the same as for (6).

## 5. LU DECOMPOSITION

In this section we consider the LU decomposition

$$A = L\,U \tag{21}$$

for a N-by-N matrix in the w-by-w SSAP. The partitioning method and the execution sequence of the resulting subproblems are specified in the following algorithm:

**Algorithm 4:**

Compute $L_{1,1}$ and $U_{1,1}$ from

$$A_{1,1} = L_{1,1} U_{1,1} \qquad (22)$$

For n:= 2 to N do

Compute $U^{n-1}_{*,n}$ from: $L^{n-1}_{*,*} U^{n-1}_{*,n} = A^{n-1}_{*,n}$ (23)

Compute $L^{n-1}_{n,*}$ from $L^{n-1}_{*,*} U^{n-1}_{*,*} = A^{n-1}_{n,*}$ (24)

$A_{n,n} := A_{n,n} - L^{n-1}_{n,*} U^{n-1}_{*,n}$ (25)

Compute $L_{n,n}$ and $U_{n,n}$ from $A_{n,n}$ =

= $L_{n,n} U_{n,n}$ (26)

The subproblem (22) and (26) involve a LU decomposition of a w-by-w matrix, $A_{n,n}$, $1 \leq n \leq N$. This can be directly accomplished in the w-by-w SSAP (without need of any transformation), by programming the boundary PEs in such a manner that PE(1,1) perform OP(E), The PEs (1,j) $2 \leq j \leq w$ perform OP(B), and the PEs (i,1) $2 \leq i \leq w$ perform OP(F).

Every subproblem (24) implies to find a block-column matrix, $U^{n-1}_{*,n}$, that satisfies the lower triangular matrix equation

$$L^{n-1}_{*,*} U^{n-1}_{*,n} = A^{n-1}_{*,n}$$

Every subproblem (28) implies to find a block-row matrix $L^{n-1}_{n,*}$, that satisfies the upper triangular matrix equation

$$L^{n-1}_{n,*} U^{n-1}_{*,*} = A^{n-1}_{n,*}$$

Both types of subproblems, and their solution in a w-by-w SSAP, have been presented in section 4 of this paper.

The subproblems (25) involve a block-row matrix $(L^{n-1}_{n,*})$ by a block-column matrix $(-U^{n-1}_{*,n})$ multiplication, with accumulation of $A_{n,n}$. This was considered in section 3.

Therefore, all the subproblems present in the partitioning algorithm 4 have been solved. The chained execution of these subproblems can be seen in fig. 5.

The computational time of the LU decomposition of a N-by-N matrix on the w-by-w SSAP is $T=N^3/w^2 + N^2/2w + N/6 + w$

## 6. MATRIX INVERSION

In most cases, the computation of inverse matrices can be avoided. Nevertheless, in some statistical and engineering applications, this calculation is mandatory.

Let us consider the case of obtaining the inverse $A^{-1}$, of a N-by-N matrix, A, which has previously suffered an LU decomposition. We have

$$A^{-1} = U^{-1} L^{-1} \qquad (27)$$

The matrix equation L X = I must be solved, to find $L^{-1}$, where I is the N-by-N identity matrix, and $L^{-1}$ is the unknown matrix, X.

This computation can be performed in a w-by-w SSAP, taking into account that $L^{-1}$ is also a lower triangular matrix; in this case

$$T = N^3/2w^2 + 3N^2/2w + N + w \quad (28)$$

In a similar way, by means of the solution of the matrix equation YU=I we

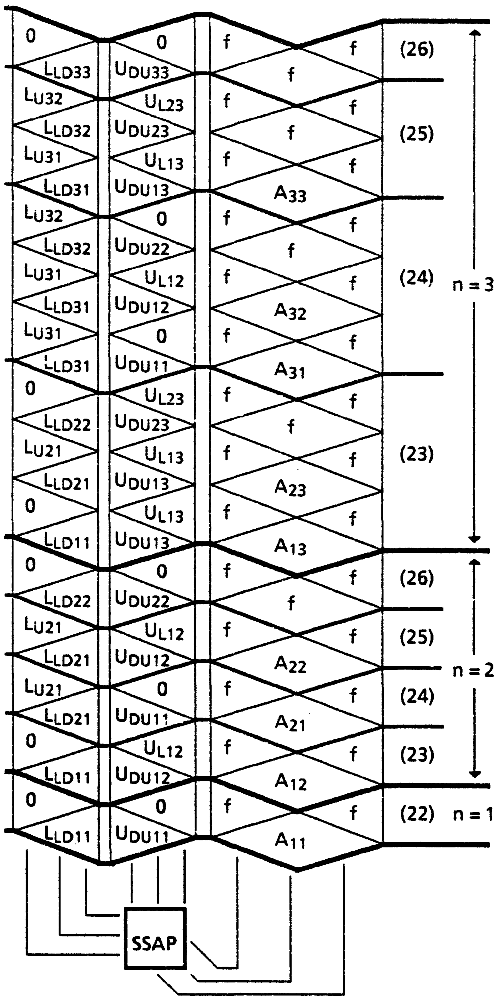


Fig.5. I/O block - data stream for a_LU decomposotion problem (with N = 3).

obtain $Y=U^{-1}$; taking into account that $U^{-1}$ is upper triangular, T is (28).

To find $A^{-1}$ from (27), the algorithm of section 3 may be used eliminating the multiplications by blocks 0. Then, we have $T = N^3/w^2 + 3N^2/2w + N/2 + w$. Therefore, the elapsed computational time to calculate the inverse of a N-by-N matrix, A, by using the LU decomposition, inverting the triangular matrices and multiplying, is:

$$T = 3N^3/w^2 + 5N^3/w + 8N/3 + 4w$$

## 7. CONCLUSIONS

In this paper, an efficient execution of several matrix algorithms with variable dimensions on a single systolic array of fixed size, has been presented. The processor is a SSAP with w-by-w PEs. Each PE is of the hexagonal type, and some of them must be able to perform several different operations. The array topology is fixed, and simple feedback and control are needed.

A partitioning method to divide the problem into subproblems matching the array size and characteristics, has been developed. Further conditions have been defined for the transformation algorithm that maps the original problem onto a different problem, which has been solved. The algorithms that have been described in this paper are: matrix multiplications, matrix equations, LU decomposition and calculation of the inverse matrices.

The array control is very simple and regular, indeed. Every partial result obtained in one boundary PE, that must be reused, is input through that same PE. The ordering of production and reuse of these data, follows a FIFO policy by blocks. No storing of intermediate results outside the systolic array is needed. The feedback networks utilizes storage elements to transmit, in pipelined fashion, the partial results.

For all the algorithms, the PE's utilization tends to the maximum possible value (1/3) and the computing time to the minimum. Nevertheless, with an adequate implementation PE's utilization approaching 1 can be obtained.

## REFERENCES

/1/ H.T.Kung and C.E.Leiserson, "Systolic Arrays (for VLSI)", in I.S. Duff and G.W.Stewart (ed.), Sparse Matrix Proceeding 1978, pp. 256-282, 1979.

/2/ H.T.Kung, "Why Systolic Architectures?", Computer Magazine, Vol. 15, No. 1, Jan. 1982, pp. 37-46.

/3/ J.J.Navarro and V.Casares, "Systolic Implementation for Deconvolution Iterative Algorithm". Proc. of the ICASSP'86, Ap. 1986.

/4/ S.Y.Kung, "VLSI Array Processors". IEE ASSP Mag. Jul. 1985., pp. 4-22.

/5/ R.W.Hockney and C.R.Jesshope, "Parallel Computers: architecture, programming and algorithms". Adam Hilger Ltd. 1981.

/6/ K.Hwang and Y.H.Cheng, "Partitioned Matrix Algorithms for VLSI Arithmetic Systems". IEEE T.on C., C-31, No. 12, Dec.1982, pp. 1215-1224.

/7/ H.Y.H.Chuang and G.He, "A Versatile Systolic Array for Matrix Computations". 12th Anual Inter. Symp. on Comp. Architecture 1985. Conf. Proc. pp. 315-322.

/8/ D.Heller, "Partitioning Big Matrices for Small Systolic Arrays". Chap. 11 of VLSI and Modern Signal Processing. S.Y.Kung, et al.(ed.). Prentice-Hall, 1985.

/9/ H.T.Kung and C.E.Leiseron, "Algorithms for VLSI Processor Arrays" in Introduction to VLSI Systems, C.A. Mead and L.A. Conway, Eds. Reading, M.A. Addison-Weslay, 1980, pp. 271-292.

/10/ J.J.Navarro,J.M.Llaberia,M.Valero, "Computing Size-independent Matrix Problems on Systolic Array Processors". 13th.Int.Symp. on Computer Architecture, Tokyo, June 1.986.

/11/ R.W.Priester et al. "Signal Processing with Systolic Arrays". Proc. of the 1981 Inter. Conf. on Parallel Processing, pp. 207-215.

/12/ R.W.Priester et al. "Problem Adaptation to Systolic Arrays". SPIE Proc., Vol. 298: Real-Time Signal Processing IV, 1981.

/13/ G.Dahlquist, A.Bjorck. "Numericl Methods". Prentice-Hall. 1974.

# PARSOR: A PARALLEL PROCESSOR FOR SPARSE MATRIX SOLUTION BY SOR ITERATION

David A. Arpin* and Yongmin Kim

Department of Electrical Engineering
University of Washington
Seattle, Washington 98195

Abstract -- The need for fast solutions of a class of linear systems has led to the design of a special purpose parallel processor. The SOR (successive overrelaxation) iterative algorithm allows a high degree of parallelism in computations, and is well suited to certain types of problems. An architecture is described which executes this algorithm efficiently, and achieves parallel operation at several levels. A small scale prototype of this architecture has been constructed. Simulation results for the complete system and test results for the prototype are outlined.

## INTRODUCTION

There are many applications for linear systems which are large (involving thousands or tens of thousands of variables) and sparse, with a symmetric positive definite (SPD) matrix (since its entries represent some physical quantity such as resistivity or elasticity which is isotropic), for which solutions must be computed quickly. For example, Kim et al. [1] devised a finite element model of the human body for use in impedance imaging. In this technique, an iterative process consisting of the solution to Laplace's equation along with several other steps is repeated up to several hundred times. The finite element method [2] converts Laplace's equation to the type of linear system described above. Since hundreds of linear system solutions are required for image reconstruction, each instance of solving Laplace's equation must be done quickly in order to produce images in a reasonable length of time. A parallel processor can potentially provide the required speed.

Gauss-Seidel iteration with successive over-relaxation, known as the SOR method [3], is a good algorithm for this application. In this method, the system Ax = b is solved iteratively. Successive iterations of each component $x_i$ are computed from a combination of the previous value of that component, the corresponding component of vector b, and products of matrix entry $a_{i,j}$ and $x_j$. This algorithm allows for a high degree of parallelism of computation. For example, the computation for a variable can begin with the available values of $x_i$ and $b_i$. As values of $x_j$ become available, products involving them can be computed and added to the summation. With careful node ordering, such computations can be in progress for many nodes simultaneously.

## THE ParSOR ARCHITECTURE

We have devised an architecture which implements the SOR algorithm efficiently. We call this the ParSOR (for Parallel SOR solution) architecture.

Since the SOR algorithm allows computation to proceed for many of the variables at the same time, separate processors for each variable in the system can be utilized efficiently. The value $x_i$ computed in each processor is needed for the computations in several other processors, so the processors must be arranged to allow efficient inter-communication.

The "one processor per variable" assumption requires that each processor be inexpensive, so that the system can be applied to problems of reasonable size. Therefore, the architecture is designed for implementation in VLSI, with each processor requiring a single chip. This limits the number of I/O pins for each processor, as well as the die area which can be dedicated to various functions.

Since matrix A is sparse, most elements ($a_{i,j}$) are zeroes, so corresponding values $x_j$ need not be communicated to compute $x_i$. For example, in the finite element method, $a_{i,j}$ is non-zero only if i and j are node numbers of nodes in a common element. For typical two-dimensional finite element models with triangular or rectangular elements, each node is connected by common elements to 6 - 8 others. In such problems, each processor will almost always need to communicate with its nearest neighbors. In other cases, some longer communication paths will be required, depending on how well the problem can be mapped onto the grid of processors.

Thus, the communication scheme must be capable of routing data to distant processors, but should be optimized for communicating over short distances. To enhance parallelism, it should be possible to communicate with several nodes at the same time. Finally, since data required in a computation can arrive at the processor while another portion of the computation is in progress, it should be possible to perform communications without interfering with the computation. These requirements must be weighed against the limited number of I/O pins and die area available in the single chip processor.

Based on all these considerations, we selected a simple nearest neighbor rectangular grid for communication between PEs, as shown in

*David Arpin is now at the Dept. of Electrical Engineering, US Air Force Academy, CO 80840.

Fig. 1. Data to be sent to more distant PEs is routed through intermediate nodes. This arrangement provides very short data paths for regular finite element grids, and the flexibility to handle longer communications paths. In order to minimize the impact of inter-processor communication on the performance of the system, each processor is partitioned into an Arithmetic Unit (AU) and a Communications Unit (CU) (Fig. 2), which can operate independently.

The Arithmetic Unit at node i receives data consisting of values $x_j$ and associated matrix elements $a_{i,j}$ from the Communications Unit, performs the multiplications and additions necessary to compute the next iterated value of $x_i$, and sends this result back to the CU. To do these tasks, the AU needs a multiplier, an adder, a few registers, and a small amount of logic to keep track of when it has finished an iteration.

When an iteration is finished, the result $(x_i)$ is sent to the nodes which need that value. In order to use this value, the receiving nodes need to know which value it is, i.e., the subscript i, so this value is appended to the message. Some means of routing the message to its destination is also needed. In the SOR algorithm, the structure of matrix A and the mapping of variables onto processors determines where the messages must be sent, so a number of rows (R) and a number of columns (C) of displacement in the network can be precomputed to specify this routing. Thus, a complete message would consist of $x_j$, j, R and C.

For a message arriving at node i, two separate activities are required. The value of the index j must be examined to determine whether the value $x_j$ is needed for the iteration computation at node i. Also, the values of R and C must be examined to determine whether the message should be sent on to other nodes, and updated to reflect the fact that the message has moved one step closer to its ultimate destination. These two determinations are independent of each other, and so can be performed in parallel. To prevent these activities from interfering with the iteration computations, they are performed by the Communications Unit.

The CU is partitioned into four separate Communication Processors (CPs) (Fig. 2), each dedicated to communication with one neighboring node in the network. This allows messages to be sent to or received from several nodes simultaneously. Each CP can determine whether the data is needed by the AU and/or other nodes, and route the data on as required, with no intervention from the AU. Figure 3 shows the basic architecture of one CP. This processor is divided into six autonomous functional units:

External Bus Interface - This unit provides the communication with another processing element. Control lines are used to allow the processors to coordinate their use of the data lines efficiently and prevent conflicts. In order to reduce the number of input/output pins, the data lines are multiplexed.

Associative Memory - This unit examines the value of the index, j, of the incoming message to determine whether the data is needed by the AU. If the value of j, as received by the External Bus Interface, is stored in this memory, it signals that the data $(x_j)$ is needed by the AU for this node, and retrieves matrix entry $a_{i,j}$.

RC Update - This unit examines the values of R and C received in the message. If either of these is nonzero, the message must be sent to another node. This unit signals the fact that the message must be sent on, updates the value of either R or C to reflect the fact that the message has moved to this node, and determines which of the other 3 CPs in this node should send the message out to the next node in its path to the specified destination.

Internal Bus Interface - This unit provides communication with other CPs in the same processing element via a single shared bus. This communication is required when the message received by the External Bus Interface of a CP must be sent on to another node in the network.

AU Interface - This unit receives data consisting of an x value and a matrix entry from the Associative Memory, and sends this data to the Arithmetic Unit. All the CPs share a single data path to the AU, and the AU uses this same path to send results to the CPs. In order to send data over this bus, the CPs must request and be granted access. However, the AU need not be granted access, since it is not possible for the AU and the CPs to send data over this bus at the same time (this is caused by the data requirements for each step in the SOR iteration and the fact that the A matrix is symmetric). When the AU finishes an iteration, it sends the new value to the AU Interface of all four CPs.

Message Generator - When a result is received from the AU, this unit generates new messages to carry that value to the other processing elements which need it.

This architecture achieves parallelism at several levels - e.g., between computations at different nodes, between computation and communication, and between the various aspects of message routing and handling. Thus, it can implement the SOR algorithm with high efficiency.

## SIMULATION and PROTOTYPE

A finely detailed, parameterized model of this architecture was constructed using the SIMULA language. We used this model to measure the number of clock cycles needed to complete each iteration, in order to obtain a performance measure which is independent of the technology used to implement the system.

The performance of the system depends in part on the number of interconnections between variables. More interconnections require more computations at each node, and cause data to be communicated over longer paths. For most of our runs, we simulated a network of 15 nodes, arranged as a 3 x 5 finite element grid with triangular elements, with some additional connections between more distant nodes added. This model provided a reasonable degree of complexity, so the results would not just represent the best possible performance.

System performance also depends on assumptions such as the number of cycles required to send a message over the external bus, and the time required to complete multiplications. Assuming nominal values of such parameters, we determined that this system can complete each iteration in approximately 330 clock cycles. Processors in the interior of the grid achieve efficiencies (i.e., percent of time when the AU is performing computations) of over 70%.

A hardware prototype of the processing element was constructed to validate these results, and further refine the details of the architecture. The complexity of the design, and the size of the data paths, made it impossible to build an entire processing node, so the prototype was scaled down substantially. We eliminated the Arithmetic Unit completely, and constructed two CPs. The AU and neighboring nodes in the grid are simulated by a microprocessor (Motorola 68000). Extensive testing of this prototype verified the operation of the communication processors and supported the conclusions of the simulation programs.

## CONCLUSIONS

Problems in this architecture such as how to load data into the processors, control the stopping of the iteration, and retrieve the results have been addressed [4]. Evaluation of these results suggests that this architecture is capable of solving linear systems involving tens of thousands of variables in less than one second.

## REFERENCES

[1] Y. Kim, J.G. Webster and W.J. Tompkins, "Electrical Impedance Imaging of the Thorax", J. Microwave Power, V. 18, pp. 245-257, 1983.

[2] K.H. Huebner and E.A. Thornton, The Finite Element Method for Engineers, New York: Wiley, 1982.

[3] A. Jennings, Matrix Computation for Engineers and Scientists, London: Wiley, 1977.

[4] D.A. Arpin, A Parallel Processor for the Solution of Large Sparse, Symmetric Linear Systems, Ph.D. Dissertation, Univ. of Washington, Seattle, 1986.

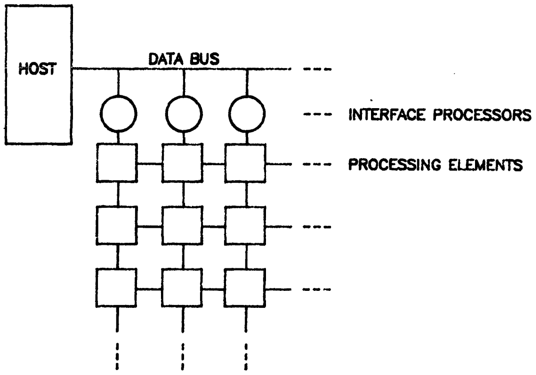


Figure 1 - Processor Interconnection

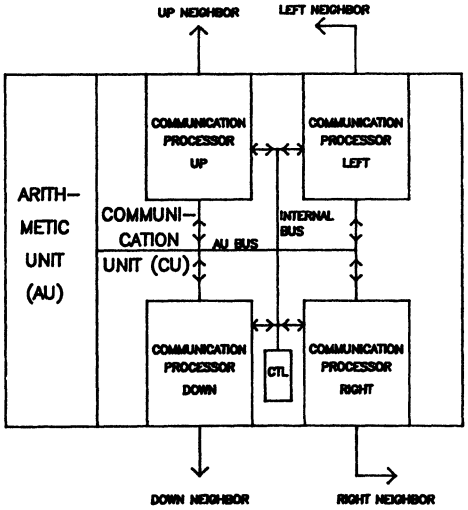


Figure 2 - Processing Element

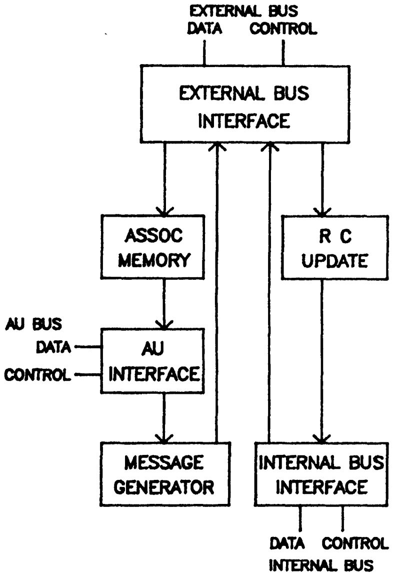


Figure 3 - Communication Processor

# Finding Test-and-Treatment Procedures Using Parallel Computation

Preliminary Report *

*Louis D. Duval* *Robert A. Wagner*
*Yijie Han* *Donald W. Loveland*

Department of Computer Science
Duke University
Durham, NC 27706

**Abstract**

A parallel algorithm to generate optimum test-and-treatment decision trees is presented. Constructing such trees is NP-hard. The algorithm is designed for a machine whose number of connections is $3p/2$, where $p$ is the number of processing elements(PEs), and where the PEs are simple enough such that a machine with $2^{20}$ PEs is currently implementable and a $2^{30}$ PE machine is feasible. A speedup of $O(p/\log p)$ over a sequential dynamic programming algorithm for this task is achieved, by paying careful attention to the communication problem. This algorithm is realized on the Boolean Vector Machine, a cube-connected-cycle system with $2^{20}$ PEs which is currently running in prototype form, with 512 PEs. The algorithm is concrete, in that all processor allocation and other control issues have been solved. The particular NP-hard problem is of independent interest; it generalizes the binary testing problem by introducing treatments on an equal basis with tests. Applications of this test-and-treatment problem are found in medical diagnosis, systematic biology, machine fault location, laboratory analysis and many other fields.

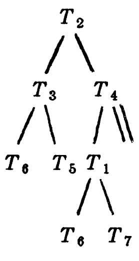
## 1. Introduction

### 1.1. The Test and Treatment Problem

The test-and-treatment(TT) problem originally defined by D.W. Loveland is a generalization of the binary testing problem studied by many researchers(see [1][2][5][6][7]). This problem is of independent interest since it finds applications in numerous real-world applications.

The test and treatment problem requests the selection of a "best" test and treatment procedure under a minimum expected cost criteria. The problem specification consists of a universe $U = \{0,1,...,k-1\}$ of $k$ objects, each with an associated weight $P_i$, and a set of tests and treatments $\{T_i, 1 \leq i \leq N\}$, each with an associated cost. The $T_i$, $1 \leq i \leq m$, denote tests, and the $T_i$, $m < i \leq N$, denote treatments. We assume that only one object is actually faulty, its identity is unknown, and each object $i$ has *a priori* likelihood $P_i$ of being the faulty object. Each test and treatment is specified by a subset of the universe; if the unknown object is in the test or treatment set then the test responds positively, or "is successful", or the treatment is successful. If the test is successful, the objects not in the test set are eliminated from consideration (and if negative, the test set of objects is eliminated), while a successful treatment ends the procedure. A failed treatment means the processing must continue. A successful TT procedure must provide for each object to be treated; a TT problem specification is ***adequate*** if there exists a successful TT procedure. With each test and treatment $T_i$ a cost $t_i$ of executing that test or treatment is given with the problem specification.

From the above description we see that a TT procedure is a binary decision tree, with both test and treatment nodes. A typical TT procedure is given in Fig. 1, where a single-line arc is used for both test outcomes (the positive outcome to the left by convention) and a treatment failure, and a double-line arc denotes a treatment set. (The double-line arc is for emphasis only since every branch of a successful TT procedure must terminate in a treatment set.)



$U=\{0,1,2,3,4,5\}$, $P_i=1/6$ for all $i$,
$T_1=\{2\}$, $T_2=\{0,4\}$, $T_3=\{0,1\}$, $T_4=\{0,1,3\}$,
$T_5=\{4\}$, $T_6=\{0,2\}$, $T_7=\{5\}$, $m=3$
$t_i=1$ for all $i$.

Fig. 1.

The TT procedure tree has an expected cost defined as: $Cost(Tree) = \sum_{i \epsilon U}$ (cost of all tests and treatments encountered if $i$ is the faulty object) $\cdot P_i$. The desired solution is the procedure which minimizes this cost. Thus $Cost = \min_{all\ trees} Cost(Tree)$.

It has been shown[3][5] that finding optimal solution to the binary testing problem is in general NP-hard. Since the test-and-treatment problem generalizes the binary testing problem, the test-and-treatment problem is also NP-hard. A parallel algorithm for this problem is presented which is implemented on the Boolean Vector Machine(BVM). We are able to achieve time complexity $O(ks(k+\log N))$ using $p=O(N2^k)$ processors, where $k$ is the size of the universe of objects containing the malfunctioned one, $s$ is the precision required, and $N$ is the total number of tests and treatments available. This result represents a speedup of $O(p/\log p)$, with regard to the known sequential algorithm which could be obtained by modifying the backward induction algorithm given by Garey[1].

### 1.2. The Boolean Vector Machine

The BVM is a CCC[8] parallel machine. Logically the BVM can be viewed as a bit array. Each row of bits forms a register. Each column forms a PE. The registers are denoted R[0], R[1], R[2],..., and are termed the PE's "memory". Each PE also contains 2 non-memory registers A and B which act like 1-bit accumulator. Let $r$ be a positive integer and $Q=2^r$, there are a total of $2^{r+Q}$ PE's, as required by a complete CCC network. The address of a PE can be represented by $(i, j)$ with the first component being the cycle number and the second the address within the cycle. Within cycle $i$ PE $(i, j)$ is only connected to its predecessor $(i,(j+Q-1)\%Q)$** and its successor $(i,(j+1)\%Q)$. In addition each PE $(i,j)$ is connected to its lateral neighbor $(i \char`^ 2^j, j)$, thus connecting the cycles together.

* Work reported herein is partially supported by the Air Force under grant number AFOSR 81-0221 and AFOSR 83-0205. This paper is a shortened version of a paper submitted for publication.

** As in the C language[4], %, /, &, |, ^ are the modulo, integer division, and, or, and exclusive-or operations respectively.

0190-3918/86/0000/0688 $01.00 © 1986 IEEE

The BVM is a bit-oriented machine. Only Boolean function operations are allowed. Each of its instructions involves possibly register A and B and at most one another register. Its instruction has the form:

{A or R[j]}, B = f, g(F, D, B) {IF or NF} <set>;

Two assignment operations will be simultaneously performed by executing this instruction. The first assigns f(F, D, B) to either A or R[j]; the second assigns g(F, D, B) to B. f and g are any Boolean functions of three arguments. F may be A or R[j]. D may be A.N or R[j].N. N denotes a neighbor PE of PE $(c ,p )$. It can be:

S: successor PE $(c ,(p +1)\%Q )$;

P: predecessor PE $(c ,(p +Q -1)\%Q )$;

L: lateral PE $(c\,\hat{}\,2^p ,p )$;

XS: even successor exchange PE $(c ,p\,\hat{}\,2^0)$;

XP: even predecessor exchange XP=P if $p$ is even; XP=S if $p$ is odd;

I: input one bit to PE (0, 0), PE $(2^Q -1,Q -1)$ outputs one bit at the same time. All other PEs get bits from their predecessors except PEs (., 0), which get bits from PEs $(.-1,\ Q -1)$.

The {IF or NF} <set> denotes the activate/deactivate set. <set> is a subset of $\{0, 1, ..., 2^r -1\}$. IF <set> means all the PE's $(i ,j )$, $0 \leq i < 2^Q$ and $j \epsilon$ <set>, will be activated while the remaining PE's will be deactivated. The meaning of NF <set> is just the opposite. If the part {IF or NF} <set> is not present in the instruction, then all the PE's are activated. There is also a special memory register, E, which is used as an enable/disable register. PE $i$ will be enabled or disabled according to whether its bit of the E register is 1 or 0. The E register itself is always enabled. The value of a PE's memory(except for register E) will not be changed while that PE is disabled. The entire state of a PE will remain unchanged by an instruction which deactivates that PE.

For further details of the BVM, the reader is referred to [9], [10].

### 1.3. ASCEND/DESCEND Algorithms

An algorithm is in the ASCEND(DESCEND)[8] form if it consists of a sequence of basic operations on pairs of data, where the addresses of the pairs differ successively in bit 0, bit 1, ..., bit $p$ -1 (bit $p$ -1, bit $p$ -2, ..., bit 0), here and henceforth bits are counted from the least significant bit.

Preparata and Vuillemin showed in [8] that the ASCEND/DESCEND algorithms can be simulated on a CCC at a slowdown of a factor of 4 to 6, regardless of the network sizes. Thus designing an ASCEND/DESCEND algorithm for a hypercube, and transforming it into a CCC algorithm seems to be a reasonable way of designing an efficient CCC algorithm. The algorithm presented in this paper is in the ASCEND/DESCEND form. The processor control is largely ignored here. We will address this issue in the expanded version of this paper.

## 2. A Parallel BVM Algorithm for the Test-and-Treatment Problem

Rather than enumerate all the possible TT trees and take the minimum cost directly as in section 1.1, we use the approach of dynamic programming and note that the optimal tree must apply the minimum cost action (test or treatment) to already optimal subtrees. The optimal subtrees are obtained by beginning with the empty tree and combining trees as just described. Thus we start with $C(\phi)=0$ and $C(S)=\infty$ for $S \neq \phi$. For an arbitrary nonempty set $S$ of objects we compute the cost $C(S)$ as follows:

$$C(S) = \min[\ \min_{1 \leq i \leq m}(t_i *p(S) + C(S \cap T_i) + C(S-T_i)),\ \min_{m < i \leq N}(t_i *p(S) + C(S-T_i))].$$

where $p(S) = \sum_{j \epsilon S} p_j$.

This definition is from first principles: the value $t_i$ is charged to each object subject to that action and the total weight of those objects to be charged is $p(S)$. For tests, one adds in the cost $C(S \cap T_i)$ of the set $S \cap T_i$ to which the test responds positively (the test set) plus the cost $C(S-T_i)$ of the set $S-T_i$ of objects not responding to the test. Treatments terminate action on the objects of $T_i$, $m < i \leq n$, (i.e., treat them) so the only objects needing further action are the objects in $S-T_i$; we add in the cost $C(S-T_i)$. The essence of an argument by induction that $C(S)$ is correctly computed uses the assumption that $C(S \cap T_i)$ and $C(S-T_i)$ are the correct costs for the subtrees and then we note from the above description that the correct minimum is taken to compute $C(S)$. We see that $C(U)=$ $Cost(tree)$ as desired.

In actual computation we will assign an array $M[S , k]$ to calculate $C(S)$: $M[S ,i]=t_i\, p(S)+C(S \cap T_i)+C(S-T_i)$, $0 \leq i \leq m$, and $M[S ,i]=t_i\, p(S)+C(S-T_i)$, $m < i < N$, therefore

$$C(S)=\min\{M[S ,i] \mid 0 \leq i < N\}.$$

The above observation is expressed in the following algorithm.

```
TT( )
{
  foreach $i$ : $0 \leq i < N$ do {
      $TP[S ,i]=t_i *p(S)$, if $|S| > 0$,
      $M[S ,i] = \begin{cases} 0, \text{ if } |S| =0, \\ \infty, \textit{ otherwise.} \end{cases}$
  }
  for($j$ =1; $j \leq |U|$ ; $j$ ++) {
    foreach $(S, i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ do {
        $M[S ,i]=M[S-T_i ,i]$;
        $M[S ,i]$+=$TP[S ,i]$;
        if($i \leq m$) $M[S ,i]$+=$M[S \cap T_i ,i]$;
    }
    foreach $(S ,i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ do
        $M[S ,i]=\min(M[S ,x] \mid 0 \leq x < N)$;
    }
  }
}
```

$|S|$ in the algorithm denotes the size of set $S$.

Can our parallel TT algorithm be transformed into the ASCEND/DESCEND form? Observe that if we assign a PE to each $(S ,i)$ pair with $M[S ,i]$ and $TP[S ,i]$ placed in different portion of that PE's memory, the instruction $M[S ,i]$+=$TP[S ,i]$ can be executed in parallel by all PEs at once.(The necessary bit-serial addition algorithm is a subroutine, executed in parallel on all PEs.) The minimization part of the algorithm can be transformed into the following ASCEND form:

```
for($t$ =0; $t < \log N$; $t$ ++)
   foreach$(S ,i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ do
      $M[S ,i]=\min(M[S ,i],M[S ,i\#t])$;
```

where $i\#t$ is the binary number obtained by complementing the $t$-th bit (from the right) of $i$.

Now consider the instruction:

foreach $(S ,i)$: $M[S ,i]=M[S-T_i ,i]$.

Can this operation be transformed into the

ASCEND/DESCEND form?

Let us begin by expanding it into its component operations as follows:

```
foreach $(S,i)$ do {
    $R[S,i]=M[S-T_i,i]$;
    $M[S,i]=R[S,i]$;
}
```

Consider now the operation

$$R[S,i]=M[S-T_i,i].$$

For each $S$ and $i$, this operation is well-defined, i.e.: each PE which receives information during this activity receives information from only one PE. However, one PE must send information to several PEs. In general, $M[S-T_i,i]$ must be broadcast to $R[(S-T_i)\bigcup V,i]$, for each $V$ such that $S\bigcap T_i \supseteq V$. The following loop accomplishes the required broadcast, for all $i$, $0\leq i<N$:

```
$R[S,i]=M[S,i]$;
for($e=0$; $e<k$; $e$++)
   foreach $(S,i)$: $U\supseteq S$ and $0\leq i<N$ and $e \epsilon S\bigcap T_i$ do
      $R[S,i]=R[S-\{e\},i]$;
$M[S,i]=R[S,i]$;
```

Let $I_t=\{j \epsilon U \mid j\leq t\}$. Using induction one can show that just before $e$ takes on value $t$, $R[(S-T_i)\bigcup(S\bigcap T_i\bigcap I_{t-1}),i]$ holds $M[S-T_i,i]$. After all iterations of the loop on $e$, $I_t=U$, and $S\bigcap T_i \bigcap I_t=S\bigcap T_i$. Thus, for all $S$ and $i$, $R[S,i]=M[S-T_i,i]$.

Similarly, the operation

$$\text{if } (i\leq m)\ M[S,i]+=M[S\bigcap T_i,i]$$

can be transformed into:

```
$Q[S,i]=M[S,i]$;
for($e=0$; $e<k$; $e$++)
   foreach $(S,i)$: $U\supseteq S$ and $0\leq i<N$ and $e \epsilon S-T_i$ do
      $Q[S,i]=Q[S-\{e\},i]$;
if ($i\leq m$) $M[S,i]+=Q[S,i]$;
```

The complete algorithm now appears as:

```
TT()
{
  foreach $i$: $0\leq i<N$ do {
      if( $|S|>0$) $TP[S,i]=t_i*p(S)$;
      $M[\phi,i]=0$;
      if( $|S|>0$) $M[S,i]=\infty$;
    }

    for($j=1$; $j\leq k$; $j$++) {
      foreach $(S,i)$: $P_1(S,i)$ {
        $Q[S,i]=R[S,i]=M[S,i]$;
      }
      for($e=0$; $e<k$; $e$++) {
        foreach $(S,i)$: $P_1(S,i)$ and $e \epsilon S\bigcap T_i$ {
          $R[S,i]=R[S-\{e\},i]$;
        }
        foreach $(S,i)$: $P_1(S,i)$ and $e \epsilon S-T_i$ {
          $Q[S,i]=Q[S-\{e\},i]$;
        }
      }

            foreach $(S,i)$: $P(S,i,j)$ {
              $M[S,i]=R[S,i]$;
              $M[S,i]+=TP[S,i]$;
              if($i\leq m$) $M[S,i]+=Q[S,i]$;
            }
            for($t=0$; $t<\log N$; $t$++)
              foreach $(S,i)$: $P(S,i,j)$ {
                $M[S,i]=\min(M[S,i],M[S,i\#t])$;
            }
          }
        }
```

Where $P_1(S,i)\equiv U\supseteq S$ and $0\leq i<N$, and $P(S,i.j)\equiv P_1(S,i)$ and $|S|=j$.

On the BVM each PE will stand for a pair $(i,j)$, where $i$ and $j$ are binary numbers and $ij$, the concatenation of $i$ and $j$, is the address of the PE. $|i|$, the number of bits in $i$, is $k$. The component $i$ denotes a subset $S$ of $U$, $a \epsilon S$ iff $a$-th bit of $i$ is 1. $j$ is the index of a test or a treatment.

The predicates $e \epsilon S\bigcap T_i$ and $e \epsilon S-T_i$ can be implemented by using the processor-ID. The Processor-ID is a bit pattern in which the memory of processor $(i, j)$ contains $ij$ [9]. The processor-ID bits will let each PE know the set $S$ it represents. $T_i$ should be input to the BVM. The most interesting part of the algorithm is the loop indexed by the variable $e$. Note that by imposing the conditions $e \epsilon S\bigcap T_i$ and $e \epsilon S-T_i$ the result becomes $R[S,i]=R[S-T_i,i]$ and $Q[S,i]=Q[S\bigcap T_i,i]$.

## 3. Conclusion

Many NP-complete problems can be solved on the BVM fairly efficiently, as we illustrate using the test-and-treatment problem. Indeed, the test-and-treatment problem itself is of real interest as it has many important applications. A parallel algorithm for this problem is presented and implemented on the Boolean Vector Machine. The communication problem and the PE allocation problem have been solved so that a speedup of $O(p/\log p)$ is achieved.

## References

[1] Garey, M.R. Optimal binary identification procedures. SIAM J. Appl. Math. Vol. 23, No. 2, Sept. 1972. pp. 173-186.

[2] Garey, M.R. and Graham, R.L. Performance bounds on the splitting algorithm for binary testing. Acta Informatica 3, 347-355(1974).

[3] Hyafil, L. and Rivest, R.L. Constructing optimal binary decision trees is NP-complete. Inf. Process. Lett. 5, 15-17(1976).

[4] Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall (1978).

[5] Loveland, D.W. Selecting Optimal Test Procedures from Incomplete Test Sets. Proc. First Int. Symp. Policy Anal. Inf. Sci., pp. 228-235. Duke University, Durham, NC, 1979.

[6] Loveland, D.W. Performance Bounds for Binary Testing with Arbitrary Weights. Acta Informatica 22, 101-114(1985).

[7] Payne, R.W. and Preece, D.A. Identification keys and diagnostic tables: a review. Jour. of the Royal Stat. Soc. (Series A) 143(3), 253-242 (1980).

[8] Preparata, F.P. and Vuillemin, J. The Cube-Connected Cycles: A Versatile Network for Parallel Computation. CACM 24,5 (May 1981), 300-309.

[9] Wagner, R.A. A Programmer's View of the Boolean Vector Machine, Model-2. CS-1981-8, Dept. of Computer Science, Duke Univ., Oct. 1981.

[10] Wagner, R.A. The Boolean Vector Machine [BVM]. IEEE 1983 Conf. Proc. of 10-th Ann. International Symposium on Computer Architecture, pp. 59-66.

# A DIFFUSING COMPUTATION FOR TRUTH MAINTENANCE

*Charles J. Petrie, Jr.*
*Microelectronics and Computer Technology Corporation*
*9430 Research Boulevard*
*Austin, TX 78759*
*(512) 343-0860*

## ABSTRACT

*"Truth maintenance" is an important artificial intelligence technique. Although some recent research has increased the efficiency of the status assignment process, it is still computationally expensive. Fortunately, there is a significant amount of parallelism inherent in the process. We present a distributed computation for status assignment in a Truth Maintenance System (TMS) where each node in the system is implemented as an independent processor.*

## 1 Introduction

Doyle first presented a formal Truth Maintenance System (TMS) in [2]. (This was later renamed a Reason Maintenance System [3], but TMS has come to mean a class of techniques within artificial intelligence and we use this generic term.) One of the major functions of such a system is to properly assign the status of IN or OUT to each node in a network. The nodes correspond to assertions and the statuses to belief in the assertions. We discuss below the structure of the network and the criteria for proper status assignment.

The status assignment process in a network of significant size requires a large amount of computation. This is evident not only from the algorithm but from general experience with systems that include a TMS, such as DUCK [6] and MRS [4]. Recent work has attempted to improve both the efficiency and functionality of the original algorithm [7] and [5]. However, the process is inherently expensive. Fortunately, status assignment in a typical network allows the status of many nodes to be determined based on only that of a relatively few other nodes. This allows a large degree of parallelism in status assignment.

Dijkstra has proposed a distributed computation called "diffusive computation" in [1]. In such a system, a computation proceeds by passing messages in a finite, directed graph of processors. A processor is activated only when it receives a message. It makes a calculation and sends the result in a message to its successor. The end of the computation is signaled by a system of replies. A TMS network of nodes can be implemented as a finite, directed graph of processors which seek to determine their status. We give a diffusive computation which implements status assignment in such a network of processors. In the discussion that follows, we assume that the reader is less familiar with the TMS proposed by Doyle than with Dijkstra's diffusive computation.

## 2 TMS Status Assignment

Each node in a TMS network is to be assigned a status of **IN** or **OUT**. A set of **justifications** is associated with each node. Each justification in the set consist of an ordered pair of sets of nodes. The first element of the pair is called the **Inlist** of the justification, and the second is called the **Outlist**. A justification is **valid** exactly when each element of the Inlist is IN and each element of the Outlist is OUT. An assignment of statuses to a TMS network is **consistent** when each node is assigned a status of IN iff it has at least one valid justification and OUT otherwise. The nodes form a directed graph where the union of the Inlists and Outlists of the justifications for a node are its predecessors, and those nodes which include it in a justification are its successors.

It is convenient to name certain subsets relative to a node. Its successors are also its **consequences**. If a node is IN, the nodes in one valid justification are designated its **supporters**. If a node is OUT, its supporters will include exactly one node from each justification: either an OUT node in the Inlist or an IN node in the Outlist. The **affected consequences** of a node are those consequences for which the node is a supporter. The transitive closure of the affected consequences constitutes the **repercussions** of the node. The **believed consequences** are those affected consequences which are IN. The transitive closure of these is the set of **believed repercussions**. The **ancestors** of a node are those in the transitive closure of its supporters. Any of the sets mentioned so far may be empty. If the set of justifications is empty, then the node has no justification and is OUT. If the Inlist and Outlist of a justification are both empty, the justification is valid and is said to be a **premise**.

We consider the problem of adding a justification to a node in a TMS network in which the status assignments are consistent and **well-founded**. This last means that no node is in its own believed repercussions. The status assignment computation should preserve the conditions of consistency and well-foundedness after the justification is added if it is possible to do so. As an example, consider Figure 1. In this graph and those which follow, each circle corresponds to a

0190-3918/86/0000/0691 $01.00 © 1986 IEEE

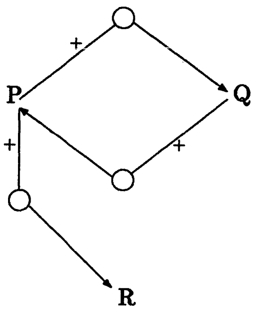


Figure 1: Unique Well-founded Consistent State

justification, with an arrow pointing to the justified node, positive arcs connected to the elements of the Inlist, and negative arcs to elements of the Outlist. The requirement of consistency is met if each node in this graph is assigned a status of IN. However, such a state of assigned statuses would not be well-founded. The only consistent well-founded state has each node labeled OUT.

We will show next how status assignment can be accomplished by a simple diffusive computation that generally follows Doyle's original algorithm. It is important to note that our computation, like that of Doyle's, is incomplete in some respects. A new justification may also introduce an **unsatisfiable circularity** into the system by creating a graph, such as that of Figure 2, for which no consistent assignment of statuses can be found. We improve upon the original algorithm by ensuring termination if a consistent labeling of statuses is not found. However, it may be possible to consistently assign statuses only if the supporters of some of the nodes are examined, and our computation only examines consequences of nodes. Our computation will also assign statuses if it is possible to do so consistently even though there is no assignment which is both consistent and well-founded. Russinoff has solved these problems in [7] with a more complex truth maintenance algorithm.

## 3 The Diffusive Computation

Imagine now that the nodes in a TMS network are processors connected to each other via two-way channels that carry messages in one direction and replies in the other. Each processor stores a set of justifications and its status. It is connected to other processors by channels which carry messages to its consequences and replies to the processors named in any of its justifications. We will describe below programs for the processors that implement the diffusive computation of status assignment.

Given a "root" processor which is initially OUT and which acquires a valid justification, we begin as Doyle does: the status of the processor and every element of its repercussions is set to NIL. This status assignment is special and only occurs during updating. The diffusive computation for this is quite simple. Each processor sends a message giving its status as NIL to each of its consequences. If a processor receives such a message from a processor which is not a supporter, or if the processor has already received a message from a supporter, it replies immediately. Otherwise, it sets its status to NIL, and sends the same message to each of its consequences. The originator of the first message is known to the processor as its "engager". When each one of the consequences of the processor, if any, has replied, then it replies to the engager. The root processor recognizes the termination of the NIL sweep when all of its consequences have replied because it marks itself as its own engager. At this point, we can attempt an assignment of IN or OUT to the designated processors.

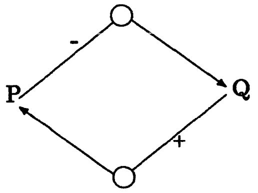


Figure 2: Unsatisfiable Dependencies

The basic idea of the diffusive computation for status assignment is that upon receiving a message for the first time, a processor records the sender as the engager and checks to see if if its own status is changed by the new information. This is determined by whether or not there is now a valid justification for the processor based on the new status of the engager. If the status of the receiving processor is unchanged, it replies immediately. If the processor's status is changed, then it sends a message containing its new status to each of its consequences. A processor also replies to any sender if it has already received another message to which it has not replied or if it has no consequences. It replies to the engager when it receives replies from all of its consequences.

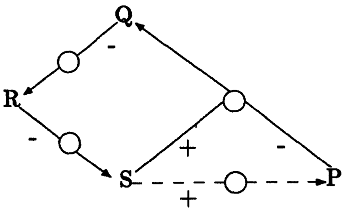


Figure 3: New Justification

We illustrate the use of such a diffusive computation with Figure 3. Suppose that initially P had no justification and was thus OUT. The justification being added is shown in the dotted lines. Processors Q and S were IN and R was OUT. Thus P is an OUT processor which acquires an initially valid justification and so becomes the root node in a diffusive computation to determine status assignments.

After the NIL sweep, both P and Q will each determine its status to be OUT because of the NIL status of S. Q will make its determination when it receives a message from P that it is OUT. Q then sends a message to R who is able to determine that its status is IN. R's message to S confirms the assumption that the NIL status of S would eventually result in an OUT status. S sends a message to Q and P. Since they are already engaged, they take note of the new status and reply immediately. Each node in the graph then replies to its engager after determining that its status is unchanged since it first received a message. In the final state, only R is IN and the other processors are OUT.

The above descriptions of diffusive computations would suffice for status assignment were it not for unsatisfiable circularities. Obviously, some graphs may not admit a consistent assignment of statuses. In such a case, the diffusive computation described above would not terminate. The simple fix for this problem proposed here is to pass a message which contains a string of processor names as well as a status. The string contains the ancestors of the engager. Thus a processor can determine if its current status depends on itself. If an engaged processor switches status twice as a result of being its own ancestor, then it is involved in an unsatisfiable circularity. (In [2], Doyle claims that an unsatisfiable circularity can be detected if a node is its own ancestor after finding a valid justification with a NIL status in the Outlist. Unfortunately, this is not the case.) We propose that a processor signals a special "trouble" reply to its engager when an unsatisfiable circularity is so detected.

Detection of an unsatisfiable circularity, given a particular state of assignments, is by itself insufficient. It may be that the diffusive computation has not yet terminated for some possible supporters of processors involved in the apparent circularity. We must specify what action is to be taken by a processor when it receives a trouble reply. Our solution is for the processor to wait until all replies have been received and then resend its status to its consequences if it has received the trouble report for the first time from a particular consequence. If a trouble reply is received twice from the same consequence, then the processor replies trouble to its engager.

For example, in Figure 4, only S is IN before P acquires its premise justification. After the NIL sweep, P determines that it is IN and sends its new status to Q and T with the message $< P, IN >$. Suppose that T is much slower than the other processors. While T is very slowly determining its status, Q determines that its own status is OUT since the status of S is NIL. R then receives the message $< QP, OUT >$. R then sends $< RQP, OUT >$ to S which then sends $< SRQP, IN >$ to Q. The latter records R's new status and replies immediately since it is already engaged. Eventually, Q receives its outstanding reply from R.

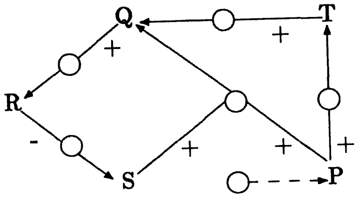


Figure 4: Apparent Circularity

Now Q finds that its status has changed to IN. Also, one of its supporters includes Q in its ancestors: S is recorded as having the status of IN due to the message $< SRQP, IN >$. Since Q is its own ancestor, it sets a circularity flag before it resends its status. It is easy to see that its status will change again due to its being its own ancestor if it does not receive a message from T before it receives its next reply from R. If T is so slow, then, since the circularity flag has been set, Q replies trouble to P.

Processor P does nothing until it hears from T. At that point, P rebroadcasts its status regardless of the fact that it has not changed. This time, T will be IN and Q will have a valid justification regardless of the status of S. Thus, the final determination of status assignments in this example will be the inverse of the state before P received a justification: only S will be OUT.

In the next section, we give the algorithm for a processor to carry out the diffusive computation described above. We omit the NIL sweep as trivial and assume that we start with all of the affected consequences of the root processor NIL and all of their consequences notified of their new status. We also assume that the root processor has already made itself its own engager and has initiated the message sending.

## 4 Algorithm for Processor $i$

Each processor $P_i$ has the following stores:

**Status:** IN, OUT, or NIL.

**Justification Set:** This is is the set of justifications for $P_i$ and will be abbreviated $\mathcal{J}$. Each Justification consists of a pair of sets of processor ids. For a particular justification $J_k$, we denote the first element of the pair as *Inlist*($J_k$) and the second element *Outlist*($J_k$). Both $\mathcal{J}$ or either element of any justification may be empty.

**Supporter-Status:** This is a vector of pairs indexed over the union of the processor ids in any set in $\mathcal{J}$. For any element $j$ of the vector, the first element of its pair is a set of processors called *Contributors(j)*. The second is *Update(j)*: the last known Status of $j$. Thus, each element of this vector has the same format as a message.

**Deficit:** The obvious integer required for a diffusive computation.

**New-Status:** IN or OUT.

**Circularity:** A boolean flag indicating a possible unsatisfiable circularity.

**Ancestors:** The set of processors upon which the current value of Status depends as determined by the procedure Status(i).

**Consequences:** The set of processors which mention $P_i$ in their justifications.

**Trouble:** The set of processors which have reported an unsatisfiable circularity with a trouble reply to $P_i$.

```
Define Processor(i):
begin
If receive message $< P_j\omega, T >$ then
  begin
  Supporter-Status(j) := $< P_j\omega, T >$
  If Engager = 0 then
    begin
    Engager := $P_j$
    New-Status := Det-Status(i)
    Circularity := false
    Trouble := {}
    If New-Status=Status then
      Send-Reply(i,'N')
      else Resend-Status(i)
    end
    else Send-Reply(i,'N')
  end
If receive normal reply from $P_j$ then
  begin
  Deficit := Deficit -1
  If Deficit= 0 then
    begin
      New-Status := Det-Status(i)
      If Trouble= {} then***No unsatisfiable circularities reported
        If New-Status = Status then *** No status change
          Send-Reply(i,'N') *** Send normal reply
          else If $i \in$Ancestors then***Possible unsatisfiable circularity
                If Circularity then***Definite unsatisfiable circularity
                  Send-Reply(i,'T') *** Trouble Reply
                  else
                  begin
                  Circularity := true
                  Resend-Status(i) *** Try to resolve possible problem
                  end
                else Resend-Status(i) *** $P_i$ is not its own ancestor
        else *** Trouble set is not empty
        If Circularity then Send-Reply(i,'T') *** Trouble Reply
          else Resend-Status(i) *** Problem may be resolvable
    end
  end
If receive trouble reply from $P_j$ then
  If $P_j \ni$ Trouble then
    Trouble := Trouble $\bigcup\{P_j\}$
    else Circularity := true *** Trouble Reply
end.
```

```
Define Send-Reply(i,b):
begin
if Engager = $P_i$ then halt
  else
  begin
  Engager := 0
  If b = 'N' then send normal reply
    else send trouble reply
  end
end.
```

```
Define Resend-Status(i):
begin
Status := New-Status
Deficit := size of Consequences
If Deficit = 0 then Send-Reply(i,'N')
  else send message $< \omega, S >$ to each consequence
    where $\omega := \{P_i\} \bigcup$ Ancestors
end.
```

```
Define Status(i):
If $\exists J_k \in \mathcal{J} \mid \forall n \in Inlist(J_k), Update(n)$ =IN
  and $\neg\exists o \in Outlist(J_k) \mid Update(o)$ =IN, then
  begin
  Ancestors := $\{j\} \bigcup Contributors(j) \forall j \in Inlist(J_k) \bigcup Outlist(J_k)$
  Return IN
  end
  else
  begin
  Ancestors := $\bigcup \forall J_k \in \mathcal{J} \bigcup Contributors(j)$
      for some $j$ | either $j \in Inlist(J_k)$ and $Update(j) \neq IN$
        or $j \in Outlist(J_k)$ and $Update(j) = IN$
  Return OUT
  end.
```

## 5 Conclusion

The amount of computation required and parallelism inherent in the truth maintenance process makes it a good candidate for a distributed computation. Dijkstra's diffusive computation is a good fit to at least Doyle's original algorithm for truth maintenance. In fact, if, like Doyle, we do not provide for termination given a graph which cannot be labeled consistently, the diffusive computations required are very simple.

Providing for such termination has the effect of complicating the final computation and significantly increasing the length of the necessary messages, as well as their number in some cases. And our algorithm does not properly identify just when it is possible to give a consistent and well-founded labeling of statuses to a graph.

However, the diffusive computation presented here is still relatively straight-forward and accomplishes as much or more than all but the most recent previous algorithms for this task. Furthermore, the remaining problems seem likely to be resolved by extensions of the current algorithm. For example, when the root processor halts with a trouble reply, it seems likely that a diffusive computation could be devised to examine its supporters as does Russinoff's algorithm. We conclude that diffusive computations are a good model for distributed implementations of truth maintenance.

## Bibliography

[1] Dijkstra W. and Sholten C. S., "Termination Detection for Diffusing Computations," *Information Processing Letters*, Vol. 11, no. 1, pp. 1-4, August 1980.

[2] Doyle J., "A Truth Maintenance System," *Artificial Intelligence*, Vol.12 No.3, pp. 231-272, 1979.

[3] Doyle J., "A Model for Deliberation, Action, and Introspection," AI-TR-581, Massachusetts Institute of Technology, AI Lab., 1980.

[4] Genesereth M., "Partial Programs," Memo HPP-84-1, Stanford Heuristic Programming Project, January 1984.

[5] Goodwin J., "An Improved Algorithm for Non-monotonic Dependency Net Update," LITH-MAT-R-82-23, Software Systems Research Center, Linkoping Institute of Technology, Sweden, August 1982.

[6] McDermott D., "DUCK: A Lisp-Based Deductive System," Yale University, Department of Computer Science, May 1983.

[7] Russinoff D., "An Algorithm for Truth Maintenance," TR AI-062-85, Microelectronics and Computer Technology Corp., July 1985.

# POTENTIALS FOR PARALLEL EXECUTION OF COMMON LISP PROGRAMS

*Patrick F. McGehearty and Edward J. Krall**

Parallel Processing Architecture
Microelectronics and Computer Technology Corporation
Austin, Texas 78759

*ABSTRACT*

Several Lisp application kernels were modified for parallel execution. Their simulated behavior showed speed-ups ranging from one to 850 times faster, suggesting that the technique is of no value to a few programs, of great value to a few programs, and of moderate value to most programs. The modifications were minimal and could be done mechanically.

## OVERVIEW

To assess how much faster Lisp programs would run on large multiprocessor systems, we modified several application kernels for parallel execution and simulated their behavior. We obtained speed-ups ranging from one to 850 times faster, suggesting that the technique is of no value to a few programs, of great value to a few programs, and of moderate value to most programs. The modifications were minimal and could be done mechanically.

An approximation of this technique is used to modify several application kernels by hand to run in parallel. The effective parallelism obtained is measured by means of simulation. These results are combined with results previously published [6] to identify areas of research needed for parallel execution of Lisp programs with minimal programmer intervention.

## REQUIREMENTS FOR "SAFE" PARALLEL EXECUTION

Unlike many languages, Lisp contains a viable "pure function" subset. Moreover, applications written in Lisp are typically composed of many functions which are "pure." Such functions can be rendered into parallel routines without fear of destroying the meaning of the program.

Unlike the functional languages, however, Lisp also supports applications which are not pure, which make use of "blackboards," global data structures, and side-effects in order to be useful. These side-effects must be carefully handled to insure correct parallel program execution.

### The Detection of Side-effects

Much of the effort in converting a program in Lisp (or any other language) to run in parallel comes in determining which parts may execute simultaneously without interference. We propose to study the restricted case of executing side-effect-free functions in parallel. Given a "black box" model of computation, a side-effect-free function may be viewed as a computational entity which accepts certain values as input parameters, and returns one or more values through the well-isolated interface of the function call. As such, the black box operates independently of its implementation -- it is behavior not structure that is important. Whether the function is implemented as software, firmware or hardware, nothing in the calling environment is altered, save the values returned through the function call interface.

A side-effect is therefore a change to the calling environment not associated with the function call interface. Note that this definition is usually weakened somewhat to exclude changes in internal addresses, memory allocation, timers, etc., which are beyond the access of the program.

There are two types of side-effect phenomena in Lisp: (1) functions which by their nature cause side-effects, and (2) functions which make use of global or free variables. Functions which use free variables have the potential for their computation being affected from any other part of the computation that uses those variables. Side-effecting functions include *rplaca, rplacd, set,* and the Input/Output functions. Moreover, any function which calls a side-effecting function may be considered side-effecting. In addition, any function which touches the property list of symbols (such as *get, putprop*) implicitly references globally available data.

*On Assignment to MCC from NCR Corporation

0190-3918/86/0000/0696 $01.00 © 1986 IEEE

Three answers may occur when performing side-effect analysis on a function: (1) no side-effects exist, (2) side-effects exist, and (3) don't know. We assume that only functions in the first category may be executed in parallel. We recognize that some algorithms exist which correctly use overlapping execution with side-effects, but we will not consider such cases here.

Static syntactic analysis should approach 100% accuracy in identifying the presence of global variables and side-effecting functions if they exist, but will be less accurate in establishing the absence of global variables or the lack of side-effecting functions.

In this determination, however, "don't know" may be the most common answer. Certain constructs are so suspicious (*(setq x (format ...))* or *(eval ...)* ) that we will flag them as side-effecting without further analysis. Others, such as *setq* will require further consideration. If a global variable is used as the assignee, then *setq* will cause side-effects. The use of a local variable as the assignee of a *setq* encapsulates the side-effect to the scope of the local variable. Hence the function itself acts as a black box, and can be executed in an applicative and parallel fashion.

### Obtaining Parallelism

After functions have been identified as safe candidates for parallel execution, some mechanism must be supplied for creating a separate task for their execution and observing the degree of parallelism obtained. We selected the *future* construct from Multilisp [5] as our primary construct for specifying parallel execution. The *future* is a form which spawns a new task for the evaluation of a given Lisp form. For example,

```
(setq x (future (add1 y))
(setq z x)
```

spawns a task to evaluate *(add1 y)*, assigns to *x* a pointer to the future of *(add1 y)*, and continues execution in line. When *(add1 y)* completes, the value of *x* becomes a pointer to the value of *(add1 y)*. If the value of *x* is needed before the future is completed, that task will suspend itself. However, if only *x* as a whole is needed (as in the subsequent assignment), the task is NOT blocked; instead *z* is set to point to the future also.

Note that the word *future* is used in two senses: as the name of the function which spawns the tasks, and as the (uncompleted) value of the task spawning. When it is necessary to distingush between the two, we will refer to the value of the task spawning as the *-future-* data-structure. In the example above, if *x* were inspected before the value was available, then something like the following might be seen:

```
*-FUTURE-*: (future (add1 y))
```

Two other forms are used in our study: *touch* and *pmapcar*. The function *touch* will return the value of its argument, blocking if necessary on an as-yet undetermined *future*. The function *pmapcar* is also built from futures. It is the analog of *mapcar*:

```
(pmapcar fn '(1 2 3 4 5))
```

will spawn tasks to evaluate (fn 1), (fn 2), ..., (fn 5) simultaneously, and return the list of values (or their futures).

It is important to note that a *future* does not automatically allow parallel execution. Consider the following:

```
(plus 10 (future (sin x)))
```

No useful parallelism is obtained. While (future (sin x)) does create a parallel task, the main line task tries to execute the plus instruction, and immediately blocks until the computation of (sin x) is completed. If *future* has any overhead, it would take less time to execute:

```
(plus 10 (sin x))
```

Therefore, a *future* should only be inserted when both paths of computation can be expected to have more computation to perform than the overhead of invoking a *future* before the invoker requires the value of the *future*.

### The Insertion Process

Because the *future* is a data structuring concept, and because it behaves like a function call, it is not only easy for the programmer to use, but it is also easy for a pre-processor to insert *futures* in a program to be parallelized. Since we do not currently have a program which detects side-effects or estimates computation requirements of code fragments, we inserted *futures* in the application kernels where we judged such programs would make those insertions.

As an example, consider the following:

```
(defun rewrite-args (lst)
  (cond ((null lst) nil)
        (t (cons (FUTURE (rewrite (car lst)))
                 (rewrite-args (cdr lst))))))
```

It is known that rewrite is side-effect-free; hence, the *future* can be inserted on the first argument of cons.

However, guaranteeing freedom from side effects is only a *sine qua non* for insertion of *futures.* The candidate function must also satisfy two other criteria:

1. The work being done by the spawned task must be sufficient in the light of the process scheduling overheads.
2. The work remaining in the spawning task until the future value is needed must also exceed the scheduling overhead.

For example, consider the following function:

```
(defun test (x) (+ (sin x) (cos x)))
```

One may insert futures indiscrimately as follows:

```
(defun test1 (x) (+    (future (sin x))
                       (future (cos x))))
```

Such insertion on the second argument of the addition is pointless -- the mainline task must block while it waits for the completion of computation of both arguments.

A better insertion is

```
(defun test2 (x) (+    (future (sin x))
                       (cos x)))
```

Now, after spawning the (sin x), the mainline will compute the (cos x) coincidentally with the computation of (sin x). The parallelism is now useful provided that (sin x) and (cos x) take longer to compute than a task takes to be spawned.

We note that Sharon Gray at MIT has developed a program for inserting *futures* in code that is alreadv known to be side-effect-free [4].

## MEASUREMENT TOOLS

Several tools were developed to allow measurement of the parallel execution of Lisp programs. Since a multiprocessor was not readily available to the authors, a simulator was developed to approximate the behavior of parallel Lisp programs on a multiprocessor. It generates a trace of events related to task scheduling and timing, such as task start, task end, and task blocking, using a value generated by another task. These trace events are used by a post-processing program to determine a running average degree of parallel execution and the overall average number of processors in use.

### The "Pure-Lisp" Simulator

The "pure-Lisp" simulator extends Common Lisp by implementing two interrelated primitives, *future* and *touch* to support parallel execution. As mentioned before, the *future* primitive returns a *-future-* data-structure. However, Common Lisp primitives do not support the *-future-* data type. This support is provided by means of a preprocessor for Common Lisp source code. The arguments to each system primitive such as *car* which requires access to their contents are enclosed in a *touch* primitive. The *touch* primitive treats normal Lisp values like the identify function. When its argument is a *-future-* data-structure, it simulates the appropriate delay, if necessary, and returns the value computed by the associated *-future-*. Thus, no changes are needed to the underlying Lisp system primitives.

During program execution, the various tasks are executed in a depth-first order, with the most recently created task being executed until it completes or starts another task. When it completes, then the task that started it is resumed. A simulated time is maintained throughout this execution. Whenever a *future* primitive is encountered, the current simulated time is recorded as the start-time for that *future.* Then the argument of that *future* is evaluated as a normal Lisp expression. The new simulated time is recorded as the end-time for that *future.* Then the simulated time is reset to the start-time of the *future.* Thus, the next instruction in the task that executed the *future* appears to occur during the time that the *future's* argument is being evaluated. When a task executes a *touch* of a *-future-* data structure, the simulated time is set to the greater of the current simulated time and the simulated completion time for that *-future-*. This adjustment represents waiting for a *future* to complete if necessary.

The "pure-Lisp" simulator described above has the advantage of relatively straightforward implementation. Further, so long as the arguments to the *future* primitive have no side-effects, the results computed will be correct and the estimated degree of parallelism and simulated timing will also be correct. However, if the arguments to the *future* primitive have side-effects, the simulator can be arbitrarily optimistic or pessimistic.

Suppose that a single computation is needed in several places in a parallel program. Further, suppose whatever portion of the parallel program needs that computation first sets a flag and computes the value while other portions of the program just use the result of the computation when they find the flag set. The *future* construct supports this kind of parallel programming, but the "pure-Lisp" simulator does not. If the shared computation first occurs inside of a *future*, then it could end up being used by another portion of the program at a simulated time earlier than it was computed. Or, if it is the value of a *future* which also includes a longer computation, another task could be delayed longer than necessary. Therefore, the "pure-Lisp" simulator used in these experiments is only appropriate for studying *futures* with no side-effects. Since the current study is only concerned with mutually independent sub-tasks, this limitation is not harmful.

### The Trace and Post-Processing Facility

The following events are traced: task start, task end, task block waiting on a *future*, and task unblock when a future is ready. Each trace event records the simulated event time and the event type. The first two events surround the *future* primitive, and the second two events surround the *touch* primitive. The act of recording the event data is hidden from the simulated timer. This series of events can readily be transformed into a time-weighted average of the number of active tasks and the number of blocked tasks. A small program computes the average number of active tasks and the maximum number of tasks that a parallel program uses. Also computed are the total computation for all tasks and the linear time required to execute the parallel program. These times represent the results that would be obtained if the program were executed on a ideal parallel processor that has as many processors as were needed, with no interference between the processors.

## SAMPLE APPLICATION KERNELS

This report examines four sample programs to assess the potentials of obtaining automatic parallel execution. Two of these, Mandelbrot and TAK are small programs, intended to exercise the primitives essential to parallel execution. The other two, EMY and REWRITE are kernels of major AI programs, intended to represent the computation of those programs.

### Mandelbrot

The Mandelbrot program determines, for each of a set of points on a square grid in the complex plane, whether the point is in the Mandelbrot set [2]. Thus, Mandelbrot can be considered a representative kernel of a class of numerical algorithms. The determination for a single point is iterative, but independent of all other points. In the sequential version of the program, the determination for each point done inside of a double loop. The outer loop controls which row in the grid is being evaluated, and the inner loop controls which column within the row is being evaluated. Given an assumption that at least five instructions should be performed by a parallel task to justify its creation overhead, the smallest computation that is suitable for execution as a separate task is the determination of whether a single point is or is not in the Mandelbrot set. Since each such determination is independent of the others, this unit of computation could be surrounded by a *future*.

To obtain experimental results, a grid of 50 by 50 points was selected, with an iteration limit of 100. Without *futures*, the program takes 27.8 seconds. Figure 4-1 shows the parallel execution profile with a single *future*. The y-axis is simulated time, and the x-axis is average number of executing tasks during a time window. With a total run length of 1.17 seconds, the speed-up is approximately 24.

| Time (ms.) | Active PE's | Each * = 2 Active Processors |
|---:|---:|:---|
| 0 | 23 | ************ |
| 100 | 19 | ********** |
| 200 | 19 | ********** |
| 300 | 26 | ************* |
| 400 | 31 | **************** |
| 500 | 31 | **************** |
| 600 | 34 | ***************** |
| 700 | 34 | ***************** |
| 800 | 28 | ************** |
| 900 | 25 | ************ |
| 1000 | 9 | **** |
| 1100 | 4 | ** |

Figure 4-1: Mandelbrot - One Future Case.

Since 2500 tasks were created and executed, this speed-up is disappointing. Careful consideration of program behavior shows that the problem is the rate at which tasks are created. On average, a delay of 360 microseconds occurs between each task startup, mostly due to loop overhead. Since each task takes an average of 11,000 microseconds, only 30 tasks can be started before some tasks complete. Variation in the length of each task causes the variation in the number of active tasks.

As the program is operating on a square grid, the iteration can easily be setup as a loop computing each point on a row, within a loop computing each row. Since there will now be a task for each row, 50 tasks will be starting tasks in parallel, and the average number of running tasks should rise to 1500. Figure 4-2 shows the parallel execution profile of the Mandelbrot program with two *futures.*

| Time | Active | Each * = 50 |
|---|---|---|
| (ms.) | PE's | Active Processors |
| 0 | 39 | * |
| 2 | 249 | ***** |
| 4 | 666 | ************* |
| 6 | 1281 | ************************** |
| 8 | 1581 | ******************************** |
| 10 | 1587 | ******************************** |
| 12 | 1476 | ****************************** |
| 14 | 1413 | ***************************** |
| 16 | 1386 | **************************** |
| 18 | 1376 | **************************** |
| 20 | 1317 | *************************** |
| 22 | 963 | ******************** |
| 24 | 485 | ********** |
| 26 | 147 | *** |
| 28 | 21 | . |
| 30 | 2 | . |
| 32 | 1 | . |

Figure 4-2: Mandelbrot - 50x50 Two Futures Case.

While 1500 tasks do execute simultaneously, the average speedup is approximately 850. The remaining sequential startup time for the 50 rows and 50 points in each column is responsible for average speedup being lower than the peak number of tasks.

A fairly simple program for obtaining automatic parallelism would find the two locations for inserting futures that were used here. A more advanced program might even use three or four levels of futures to speed the startup further. As very substantial amounts of speedup were obtained with minimal overhead, we conclude that programs similar to the Mandelbrot program can be automatically converted to effective parallel execution.

## TAK

TAK is a six line program used in the Gabriel benchmarks [3] intended to test the efficiency of Lisp implementations in calling recursive functions. For the purposes of this report, TAK provides opportunity for rapidly increasing parallelism provided that efficient primitives for fine grain task computation are available. Each recursive call to TAK spawns four more calls to TAK, three of which can execute in parallel. Thus, the potential parallelism of TAK grows exponentially with the call depth. Since the benchmark requires a call depth of 18 to complete, massive parallelism is available. However, a typical recursive call has fewer than ten instructions not counting those required by the deeper levels of recursion. If the overhead of creating a separate task is great, then the effective speedup will be poor. Therefore, TAK requires an efficient implementation of the parallelism primitives. TAK requires 3.05 seconds of sequential time to execute without *futures.* Figure 4-3 shows the results of *futures* on each of the three arguments to the recursive call to TAK.

The total processing time increases to 5.95 seconds in the parallel execution of TAK, almost doubling. The overhead of creating tasks is almost as much as that of executing. However, most of these task creations occur in parallel, so an effective speedup of approximately 85 is obtained. Indeed, if it were not for the long tail of low processor count, the speedup would have been greater. This tail is caused by tasks which block on their futures resuming and continuing the recursive computation. Therefore, even with very fine grain parallelism, significant speedup is possible.

## EMY

| Time | Active | Each * = 50 |
|---|---|---|
| (ms.) | PE's | Active Processors |
| 0 | 1060 | ******************** |
| 2 | 1414 | *************************** |
| 4 | 260 | ***** |
| 6 | 108 | ** |
| 8 | 45 | * |
| 10 | 25 | * |
| 12 | 15 | . |
| 14 | 10 | . |
| 16 | 6 | . |
| 18 | 3 | . |
| 20 | 3 | . |
| 22 | 2 | . |
| 24 | 6 | . |
| 26 | 7 | . |
| 28 | 2 | . |
| 30 | 4 | . |
| 32 | 5 | . |
| 34 | 4 | . |
| 36 | 1 | . |

Figure 4-3: Tak with three *futures.*

EMY is an implementation of the inference kernel of the EMYCIN expert system [1]. It represents approximately 600 lines of Lisp code. It was discussed in detail in [6]. Briefly, it uses backward chaining reasoning to answer queries against its database. During the process of making deductions, it updates a global database, to save conclusions for use by other rules. These frequent operations on a shared database or "blackboard" prevent much parallel operation without use of synchronization primitives. Note that the "blackboard" model of execution is used in other AI programs such as Hearsay.

Nine locations were found for inserting *futures* in the EMY program. The effective speedup obtained was negligible. The lack of speedup reflects the small granularity of parallel tasks between operations on the shared database. These results are particularly noteworthy, as the EMY program has been shown to allow significant speedup (by factors of ten to twenty) when synchronized operations are allowed on the global database. At the current time, it is not well understood how to insert automatically synchronization around the access to a shared data structure.

## REWRITE

REWRITE is a simple theorem prover developed by Boyer and Moore to evaluate Lisp implementations. It is also included in the Gabriel benchmarks. It represents approximately 100 lines of Lisp code and 350 lines of data consisting of 104 lemmas. It was written with the intent of providing behavior that was representative of the much larger Boyer and Moore Theorem Prover. Theorem proving is an important part of the planning section of many AI programs. While the authors (Boyer and Moore) now have some doubts about the representative nature of REWRITE, it is the best available sample theorem prover. As it is shown in the Gabriel benchmarks, there are two places in the code which use global variables to pass extra values back from a function. To allow parallel execution, the code was changed to pass the extra values back as part of the function. After this change, most of the program is without side-effects, allowing automatic parallel execution.

The following theorem was selected for our measurement experiment:

```
(IMPLIES
   (AND (AND P1 Q1)
        (AND (IMPLIES P1 P)
           (IMPLIES Q1 Q)))
   (OR P Q))
```

| Time | Active | Each * = 5 |
|---|---|---|
| (ms.) | PE's | Active Processors |
| 0 | 10 | ** |
| 5 | 9 | ** |
| 10 | 17 | *** |
| 15 | 27 | ***** |
| 20 | 30 | ****** |
| 25 | 37 | ******* |
| 30 | 36 | ******* |
| 35 | 42 | ******** |
| 40 | 56 | *********** |
| 45 | 75 | *************** |
| 50 | 84 | ***************** |
| 55 | 85 | ***************** |
| 60 | 99 | ******************** |
| 65 | 136 | *************************** |
| 70 | 150 | ****************************** |
| 75 | 134 | *************************** |
| 80 | 143 | ***************************** |
| 85 | 139 | **************************** |
| 90 | 99 | ******************** |
| 95 | 56 | *********** |
| 100 | 72 | ************** |
| 105 | 76 | *************** |
| 110 | 34 | ******* |
| 115 | 26 | ***** |
| 120 | 15 | *** |
| 125 | 11 | ** |
| 130 | 2 | . |
| 135 | 2 | . |
| 140 | 1 | . |
| 145 | 1 | . |
| 150 | 1 | . |

Figure 4-4: REWRITE with three *futures.*

Figure 4-4 shows the results of hand-insertion of three *futures* on the selected theorem. Sequential execution requires 7.1 seconds. A speedup on the selected test case of approximately 48 was obtained. More complex theorems would yield more opportunities for parallel execution, so we would expect greater speedups. We expect that similar results could be obtained from programs similar to theorem provers.

## Summary of Results

Table 4-1 summarizes the results of each experiment. The Mandelbrot program is listed in both the 1-*future* and 2-*future* form. We have observed speedups based on automatic syntactic insertion of parallel constructs ranging from 1 to 850. We draw the obvious inference that the amount of parallelism to be obtained is strongly dependent on the application selected.

| Program | Parallel Time (secs.) | Sequential Time (secs.) | Parallel Factor |
|---|---|---|---|
| Mandelbrot-1 | 1.168 | 27.8 | 24 |
| Mandelbrot-2 | 0.032 | 27.8 | 850 |
| TAK-3 | 0.036 | 3.1 | 85 |
| EMY | 0.500 | 0.5 | 1 |
| REWRITE | 0.150 | 7.1 | 48 |

Table 4-1: Summary

## CONCLUSIONS

### Style

Programming style is a major determinate of the effectiveness of these mechanical parallelizing techniques. The EMY program achieved no gain in performance due to frequent reading and writing of global data structures. The Mandelbrot program achieved massive parallelism because a well-defined separable unit of computation was incapsulated in a function. TAK and REWRITE achieved their parallel execution by having recursive functions with several arguments, each requiring substantial computation. Other programming styles may be expected to aid or hinder automatic parallel execution in similar ways.

### Implications

The approach adopted here — (simulated) automatic syntactic insertion of parallel constructs — has potential both for research into parallel execution of some Lisp programs, and also for "real" Lisp application programming. We expect to continue investigation of Lisp applications to broaden our understanding of the relationship between programming style and achievable parallelism.

Software engineering theorists have stressed that an applicative, non-side-effecting style of programming produces clearer and safer code. It can also be run in parallel with the same clarity and safety.

The requirements of many applications force a "blackboard" model for efficient operation. To the degree that the blackboard permeates the program design, to that degree will automatic syntactic insertion of parallel constructs be hindered. This style of programming shows that the automatic parallelization by syntactic pre-processors is not panacea for all programs, but just a part of a total solution.

### Approach to a Solution

A more complete solution would include: (1) syntactic pre-processors for modifying existing programs, (2) interactive analyzers to guide programmers in modifying existing programs for parallel execution, and to guide them in the initial construction of new applications, and (3) research into semantic analyzers to allow automation of the parallel execution of a broader class of programming styles.

In addition, it is imperative that a "cost analyzer" be incorporated into the pre-processor. The purpose of the analyzer is to predict the processing cost (time, memory references, bus delays, etc.) in order to estimate the benefits of simultaneous execution.

## REFERENCES

[1] Buchanan, B., and Shortliffe, E. *Rule-Based Expert Systems*, Addison-Wesley, 1984.

[2] Dewdney, A. K. "Mandelbrot Sets" (in the column "Computer Recreations"), *Scientific American*, July 1985.

[3] Gabriel, Richard. *Performance and Evaluation of Lisp Systems*, The MIT Press, Cambridge, 1985.

[4] Gray, Sharon. "Using Futures to Exploit Parallelism in Lisp," Master's Thesis, Massachusetts Institute of Technology, February 1986.

[5] Halstead, Robert. "Implementation of Multilisp: Lisp on a Multiprocessor," *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[6] Krall, E. and McGehearty, P. "A Case Study of Parallel Execution of a Rule-Based Expert System", *International Journal of Parallel Programming*, (accepted for publication).

# REPRESENTING S-EXPRESSIONS FOR THE EFFICIENT EVALUATION OF LISP ON PARALLEL PROCESSORS

W. Ludwell Harrison, III
David A. Padua

Center for Supercomputing Research and Development
University of Illinois
Urbana, Illinois 61801

Abstract -- Present methods for exploiting parallelism in Lisp programs perform poorly upon lists (long, flat s-expressions), as such structures must be both created and traversed sequentially. While such a serial operation may be masked by overlapping it with other computation (by virtue of process spawning, or by the use of a mechanism such as *futures*), it represents a lost (and potentially large) source of parallelism. In this paper we describe the representation of s-expressions employed in PARCEL (Project for the Automatic Restructuring and Concurrent Evaluation of Lisp), which faciliates the creation and access of lists, without compromising the performance of functions which manipulate s-expressions of a more general shape. Using this representation, the PARCEL compiler translates Lisp programs written in a subset of the Scheme dialect (which allows for global variables and atom properties) into code for a large, tightly coupled shared memory multiprocessor.

## Introduction

Conventional methods of evaluating Lisp programs on parallel architectures are heavily skewed in favor of s-expressions which assume the shape of (relatively balanced) trees. In particular, it is frequently pointed out that much parallelism may exist in a program, written in an applicative subset of Lisp, which makes a symmetrical traversal of such an s-expression. By the use of a construct such as the *future* (see [6]) the time to traverse and operate upon a long, flat s-expresssion may be masked to a degree (if several processes are traversing the list, their operations upon it may be pipelined), but the essentially serial nature of such operations cannot be overcome by such a mechanism. In this paper we present the details of the storage scheme employed by PARCEL (Project for the Automatic Restructuring and Concurrent Evaluation of Lisp). This representation is intended to allow the fast creation and access of s-expressions which have the shape of lists, while not detracting from the performance of code which traverses s-expressions of other shapes (such as the trees mentioned above). The representation has some properties which make it interesting apart from its implications for parallelism in Lisp.

PARCEL is an investigation of the problem of compiling Lisp for evaluation on a large, tightly coupled shared memory multiprocessor. The project consists of a restructuring compiler, which produces from a program written in a subset of the Scheme dialect of Lisp an object code containing various directives for parallelism, and a run-time system, including parallel algorithms invoked by the compiled code, and a parallel garbage collection mechanism. The subset treated by PARCEL includes the side-effect causing mechanisms of free variables, atom properties, etc., as well as downward and upward funargs.

The PARCEL compiler employs novel algorithms for the compile-time transformation of Lisp programs. Some of these are extensions of ideas developed in the PARAFRASE project (see [8], [12]), while others are specific to the problem of compiling Lisp. For instance, an algorithm for creating loops from a recursive function is employed, which requires neither that the function be tail-recursive, nor that it be side-effect free (the algorithm is described briefly in this paper). A complete description of the design of the compiler is presented in [7].

The machine for which we are compiling is assumed to have a large number of identical processors connected to an interleaved memory via a fast interconnection network. Examples of such machines are the NYU Ultracomputer (see [4]), and the CEDAR machine (see [5]). We assume that the processors operate asynchronously, but at nearly identical rates of processing. Delays due to memory read conflicts will be ignored, in estimations of the performance of algorithms presented in this paper.

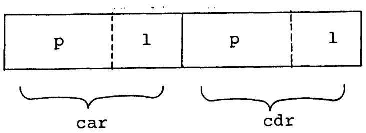


Figure 1. The Cons Cell Employed in PARCEL

## Pointers and Cons Cells

The PARCEL compiler employs an unusual cons cell, illustrated in Figure 1.[a] This cell consists, of course, of two pointers, the `car` and the `cdr`. The `p` field within each is an address in memory; for simplicity, let's assume that the fundamental unit of memory (machine word) is a single cons cell. The `l` field is an integer whose use will be explained shortly.

Throughout this paper, the word ***pointer*** will indicate a pair `[p, l]` such as the `car` and `cdr` in the illustrated cell. (Brackets will be used when treating a pointer explicitly as an ordered pair.) When we mean the location in memory of a data object, we will use the word ***address***; an address is but one of the two components of a pointer. If `z` is a pointer, then we will use `z.p` to mean the `p` component of `z`, and `z.l` to mean its `l` component. If `z` points to a cons cell, then we will use `z.p->car` to indicate the `car` pointer within the cell, and `z.p->cdr` to mean its `cdr` field. Accordingly, the components of the `car` pointer would be `z.p->car.p`, and `z.p->car.l`, and likewise for the `cdr` pointer. (This is meant to mimic the conventions of a language such as C or pascal.)

Consider Figure 2. The list `x = (a b c d e f)` is illustrated. Beside each arc (`p` component) in the figure is shown the value of the corresponding `l` component. For "variable" pointers (such as `x`) and `car` pointers, the `l` component indicates the length of the pointed-to s-expression. The list `x` has length six; beside the arc leaving `x` is shown a 6. A length of zero is associated with pointers to atoms. The `l` component of a `cdr` pointer has a very different meaning. Suppose that we take a double vertical line separating a pair of adjacent cons cells to indicate that the cells occupy adjacent locations in memory. In this example, the sublist `(a b c d)` consists of a block of four cons cells in contiguous memory locations (`x.p` through `x.p + 3`), and the sublist `(e f)` occupies two consecutive memory locations. If `y` is

[a] For the purposes of illustration, we have omitted other tag bits used to indicate, for example, a numeric atom (integer), etc.

[b] We will see shortly that it is also possible for a list to terminate "within" a contiguous block, i.e., at a cell `y` such that `y.cdr.l` ≠ 0.

0190-3918/86/0000/0703 $01.00 © 1986 IEEE

a cons cell, then y.cdr.l indicates the number of cons cells "to the right" of y in the contiguous block of cells containing y. In other words, if one is traversing a list beginning at y, y.cdr.l is the number of adjacent locations following y that will be visited before a cdr pointer leads to a non-adjacent memory location, or the end of the list is reached.[b] This is very similar to the *cdr-coding* scheme for list storage; see [1], [10]. Cdr-coding is a technique for reducing the average space consumed by a cons cell, and is based on the observation that the cells in the top level of a list frequently occupy contiguous memory locations. The storage scheme presented here has properties not available from cdr-coding; the motivation for its development is to facilitate the parallel creation and access of lists, although we will see that it allows a more flexible sharing of sublists than conventional list representations, which may result in greater than usual efficiency of storage. For the moment we will assume that, unlike in the cdr-coding scheme, all cdr pointers are present and properly assigned, even within contiguous blocks of cells.[c]

## The Primitive Operations car, cdr, cons and length

Figure 3 shows our definitions of four primitive operations upon s-expressions: car, cdr, cons, and length.

Let's consider these definitions in turn. car(x) simply returns the value of the cons cell to which x points. cdr(x) is a bit more complex. Notice first that we do ***not*** use the reading of a null pointer as an indication of list termination; if the length of x is one, then cdr(x) returns nil, under the assumption, which we will make throughout this paper, that all lists are nil-terminated.[d] nil is, like all pointers, an ordered pair consisting of the address of the atom nil and an l field of zero. If x.l > l, then cdr(x) returns a list whose length is one less than that of x.

Now consider the function cons(y, x). If the newly allocated cell happens to immediately precede x in memory, then the l component of the cdr pointer of this new cell is set to indicate that it is part of the same contiguous block of cons cells as the first cell in x; otherwise, this l component is set to zero, indicating that the new cell is the rightmost in a new contiguous block of cells. This is an unsatisfactory method of creating large contiguous blocks, as it depends upon only one list at a time being created by calls to cons. In other words, if calls made to cons for the allocation of cells to be used in building a list x are interspersed with calls for cells to be used in building y, then neither x nor y will contain long contiguous blocks. We will return to this matter shortly.

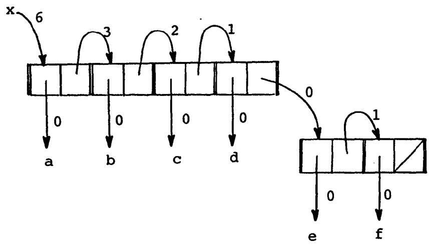


Figure 2. The List (a b c d e f)

Finally, consider the trivial definition of length(x). It goes without saying that if we associate the length of an s-expression with every variable and car pointer, that we may find the length of a list in constant time.

## Some More Primitive List Operations

Now let's consider some slightly more complex list operations as they might be defined to take advantage of our storage scheme. Consider the operation nthcdr, as defined in Figure 4. nthcdr(x, n) is equivalent to cddd ... dr(x) (n d's). It works by first deciding if the cell for which it is searching is among those in the contiguous block containing the first cell in x; if so, it jumps immediately to that cell, and returns a list n cells shorter than x. Otherwise, it skips directly to the end of this contiguous block, and follows the last cdr pointer to the next contiguous block, repeating the process. The important point is that any cell within x may be reached within time proportional to the number of contiguous blocks comprising x, and not proportional to the length of x. A great deal of effort is spent by the PARCEL compiler and run-time system in assuring that the contiguous portions of a list are as long as possible; that is, that each list is comprised of as few contiguous segments as possible. We will return to a discussion of some of the means of doing so.

The operation firstn(x, n) reveals an essential characteristic of our list representation. As its name indicates, firstn returns a list consisting of the first n cells of x. Given a conventional representation for lists, there are two ways of achieving this effect. The first (the usual definition of firstn) is simply to copy the first n cells of x, and return a pointer to this new list. This has no ill-effect upon the storage system as a whole, but necessitates the expenditure of n cells and O(n) time to copy them. The second means is to march out to the nth cell in x, and rplacd the value nil into the cell's cdr field. This has the virtue of requiring no additional storage, but the disadvantage that the original list, along with every other list sharing the cell, is altered, and that it, too, takes O(n) time.

Notice how easily this operation may be performed given our representation. This is entirely due to the fact that the end of a list is located, not by finding a null cdr pointer, but rather by successively decrementing the list's length as it is traversed. In particular, notice that we may remove the last element of a list in constant time, with the operation firstn(length(x) - 1), which makes no examination of the list x at all. This allows us to replace a destructive, side-effect causing operation (rplacding the next-to-last cell of a list) with a functional one, and to reduce the computation time by a factor of n at the same time.

Finally, consider the definition of lastn(x, n) which takes, like nthcdr, time proportional to (at most) the number of contiguous blocks of which x is comprised.

The operations illustrated in this section are very useful when performing traversals (especially parallel access) of an s-expression which is essentially list-like; that is, one that is long in the "cdr" direction. But notice that the speed of a non-linear traversal (one which follows, say, car and cdr pointers in a symmetrical way) has not been compromised; the functions car and cdr have remained quite inexpensive. This is important, for, just as it is not always the case that s-expressions describe balanced trees, so it is not always the case that they describe long and flat lists; we wish to use a representation which does not provide fast access for the one shape at the expense of the other.

[c] As the reader may have guessed, this is unnecessary, but the details of conditionally omitting these fields are not treated in this paper. This system is being implemented experimentally for the CEDAR machine [5], which has a 64 bit word; each of our pointers will occupy one machine word.

[d] It is a simple matter to extend the representation to accomodate improper lists; see [7].

```
nthcdr(x, n) ≡
  if n = 0 then return(x)
  else
     dist := x.p->cdr.l;
     if dist ≥ n
        then return([x.p + n, x.l - n]);
        else return(nthcdr(
                          cdr([x.p + dist,
                               x.l - dist]),
                          n - dist - 1));

firstn(x, n) ≡ return([x.p, x.l - n]);

lastn(x, n) ≡ return(nthcdr(x, x.l - n));
```

Figure 4. Definitions of nthcdr, firstn, and lastn

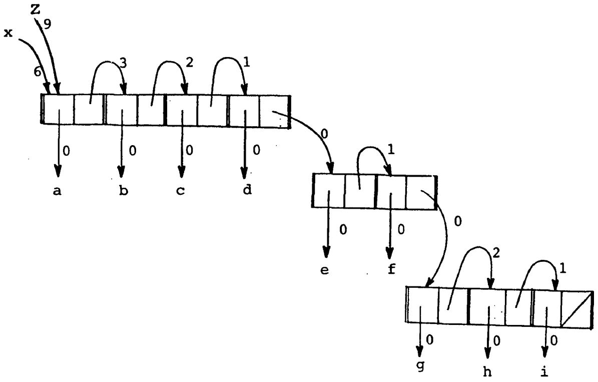


Figure 5. After Performing append(x, y)

## rplaca, rplacd, append, and nconc

The operations rplaca and rplacd are forbidden in PARCEL. While the operation rplaca is no different to perform using our representation than when using a conventional one, the increased level of sublist sharing causes a concomitant increase in the likelihood of rplacaing a shared cell, and makes it practically impossible to give a precise semantic definition of the construct. This combines with the fact that rplaca may be used to violate the call-by-value mechanism in Lisp to make for a very sticky problem. The operation rplacd is far more difficult to perform using this representation. The reason is that the usual semantics of rplacd cause it to affect the length of every list sharing the altered cell. Even if we could determine the location of every variable and car pointer to a list sharing each cell (a very expensive affair), we would have to know the exact position within the list of the cell to determine its effect on the list's length. The philosophy of PARCEL is to forbid the rplac constructs until (unless) they can be incorporated without a crippling effect on the parallelism of the resulting code.

Two of the most important uses of rplacd are to remove elements from the end of a list (delete a suffix) and to add elements to the end of a list, in both cases without causing any of the unaltered portion of the list to be copied. We have already seen that our representation provides a much faster means of doing the former (via firstn); let us now turn to the latter. Consider Figure 2 again, and notice that the cdr pointer of the last cell of the list is never used; non of the primitives we have defined so far examine or alter its contents.

Suppose that we wish to form the list z = append(x, y),, where x is as shown in Figure 2 and y = (g h i). We may perform this operation by rplacding the last cell in x;[e] the p component of this cdr pointer is set to the address of the first cell in y, and its l component is set to zero. We return a pointer to the first cell in x, with an l component equal to x.l + y.l. Figure 5 shows the state of affairs after this operation. Notice that this operation has the semantics of append, and not of nconc; this is, as we have said, because the extent of x is determined using its length and not by searching for a null cdr pointer. The reader may wish to verify that the primitives defined to this point still apply to the list x.

Figure 6 shows a definition of the operation append(x, y). It works as follows. First we find the last cell in x. If the cdr pointer of this cell is nil, then we simply place the address of y in the p component of this cdr pointer, and a zero in its l component, indicating that y is not part of the contiguous block containing the last cell in x. In this case, the entire operation takes only time proportional to the number of contiguous blocks of which x is composed. If the cdr pointer of the last cell in x is non-null, then we must use the old (conventional) definition of append (called old-append here), which copies x to a new location. If, for instance, after performing z = append(x, y) as above, we attempt to form q = append(x, y), we will find that the cdr pointer of the last cell in x is non-null, forcing us to perform the old version of append. Notice that if x is represented in a small (constant) number of blocks,[f] that it may be copied in parallel with linear speedup, to a new (contiguous) block of cons cells. Even if this is done sequentially, by copying it to a contiguous block, we guarantee that later accesses to x will be made faster. (It is easy to allocate a contiguous block, since we know x's length.)

```
append(x, y) ≡
    lastcell := last(x);
    /* lastcell is a pointer */
    if lastcell.p->cdr := nil then
        lastcell.p->cdr := [y.p, 0];
        return([x.p, x.l + y.l]);
    else return(old-append(x, y));
```

Figure 6. A Definition of append

This definition of append affords many of the advantages of nconc, without the side-effects to other lists for which that operation is well-known. Of course, there are occasions for wishing to alter several lists, using "back-door" side-effects, via such constructs as rplacd and nconc; but, as we have said, our primary

[e] Ironically, forbidding the use of rplacd has allowed us to make use of it in a way that is impossible when it is permitted.

[f] A possibility worth exploring is to add to each car and cdr pointer a field indicating the number of contiguous blocks of which the referenced s-expression is composed. The ratio of the length of the list to this value is an estimate of the potential speedup (number of useful processors) of a traversal of the list, such as that done during copying. It is easy to incorporate this information into the return value from cons, append, etc. One difficulty comes when using firstn as we have defined it (there is no way to see the resulting number of blocks without examining the list), but if the number of blocks in a list is used for purposes of estimation, then we may assume that the ratio of length to number of blocks for a particular list remains constant, and adjust the latter accordingly. Such a number might also be used to determine when garbage collection should be applied (our garbage collection algorithm puts every list in the system into contiguous form, space permitting). But we digress.

motivation in developing this representation of s-expressions is to facilitate the parallel reading and writing of lists. It is hoped that the advantage of being easily able to do so will outweigh the advantage of permitting such aliased side-effects.

The definition (of append) is not yet satisfactory however; as given in Figure 6, it decides dynamically whether its operation can be done using "nconc", or must be done by copying its first argument. If, however, there is a sufficient number of processors at hand, it may be best to copy ***both*** arguments to a single contiguous block, than to copy only one or none. In the extreme case, suppose we have two lists of length n/2 (which are, say, in fully contiguous form), and n processors with nothing to do. Clearly it would be best to put these to use by copying these lists (in one time step) to form a single contiguous list.[g] Experimentation is needed in designing a well-behaved heuristic for deciding when such copying should be done; for the moment, let us say that if the number of available processors P is greater than n/k, where k is some small constant and n is the sum of the lengths of the arguments to append, then both arguments will be copied; otherwise the definition of Figure 6 will be applied. (This seems an ideal application for information concerning the number of blocks composing a list - see footnote (f) above.)

## Parallel Creation and Access of Lists

In this section we will show, in an abstract setting, how some recurrences involving list operations may be solved in parallel; the examples will seem, no doubt, removed from the realities of Lisp programming. Afterwards, we will suggest briefly the way our compiler arranges for the detection and solution of such recurrences in the context of real Lisp programs. The example of Figure 5 suggests the possibilities for sublist sharing allowed by this representation. Notice that we have done something impossible with ordinary representations: two lists may share a cell without sharing the entire subexpression rooted at that cell. It is this important property which will allow us to solve some (ordinarily very costly) recurrences involving list operations with greater than linear speedup (when compared to conventional representations and solutions).

Figure 7 shows a simple recurrence, consisting of a loop in which x is traversed, cell by cell. When f(i) is true, x is advanced to the next cell, otherwise it is unaltered. Let us suppose that we wish to know the value of x after every iteration of this loop (say, for input to another parallel loop). We may treat x as an array x[] of length n (a technique called ***scalar expansion*** - see [12], [7]). x[0] will be synonymous with the starting value of x, and x[i] will be the value of x after iteration i of the loop in Figure 7. In this case the code of Figure 8 will suffice. (A doall loop is one in which all iterations may be executed simultaneously; that is, given n processors, where n is the number of iterations of the doall loop, the loop will complete in the time required by its longest running iteration. See [2]). After loop (2), d[i] will indicate the number of times cdr has been applied to x in producing x[i]. This computation of the vector d[] embodies a familiar technique called ***parallel prefix*** - see [8]. Assuming that x is contained in a small (constant) number of contiguous blocks, the speedup obtained is O(n/*lg* n). We can do much better than this if n is large relative to the number of processors (P) allocated to this loop; in fact, if n > P·*lg* P, then we can achieve constant efficiency (a running time of O(n/P)) by operating on the vector d[] in subsequences of length n/P on each processor, before and after the "recursive doubling". Figure 9 shows the improved version of the algorithm in loop (2), which will now have a running time of $T_P$ = O(n/P + *lg* P). The vector D[] has length P + 1; the algorithm works in the following way.

```
do i = 1 to n
    if f(i) then x := cdr(x);
```

Figure 7. A Simple Recurrence Involving cdr

```
(1) doall i = 1 to n
       if f(i) then d[i] = 1;
               else d[i] = 0;
    dist = 1;
(2) do j = 1 to ⌈lg n⌉
      doall i = 1 to n
         if i > dist then
            d[i] := d[i] + d[i-dist];
      dist := dist * 2;
(3) doall i = 1 to n
      x[i] := nthcdr(x[0], d[i]);
```

Figure 8. A Parallel Version of Figure 7

```
doall i = 1 to P
   do j = (i-1) ·⌈n/P⌉ + 2 to i·⌈n/P⌉
      d[j] := d[j] + d[j-1];
   D[i] := d[i·⌈n/P⌉];
do j = 1 to ⌈lg P⌉
    doall i = 1 to P
       if i > dist then
          D[i] := D[i] + D[i-dist];
    dist := dist * 2;
D[0] := 0;
doall i = 1 to P
    do j = (i-1)·⌈n/P⌉+1 to i·⌈n/P⌉
       d[j] := d[j] + D[i-1];
```

Figure 9. An Efficient Algorithm for Parallel Prefix

Processor i, 1 ≤ i ≤ P, forms the sum d[(i-1)·⌈n/P⌉+1] + d[(i-1)·⌈n/P⌉+2] + ... + d[i·⌈n/P⌉]. Then, in the middle loop in the figure, we form the partial sums D[i] = d[1] + d[2] + ... + d[i·⌈n/P⌉]. Finally, we add the value D[i-1] into each of d[(i-1)·⌈n/P⌉+1] through d[i·⌈n/P⌉], in the final doall loop. This is a well-known method for folding the parallel prefix solution onto limited processor; see [8].

Now consider the example in Figure 10. Here, we are adding the list x[i] to the end of y, during each iteration of the the loop. (x[i] might, for example, be a value such as f(car(nthcdr(z, i-1))), for some other list z, as computed by a loop such as that in Figure 8. Notice how the computation has been broken into a series of parallel steps; this is the result of ***loop distribution*** - see [8], [12], [7]. Figure 11 shows the parallel version of this code, in which the value of y after every iteration of the original loop (y[i] is the value after the ith iteration) is produced. In loops (1) and (2), we compute d[i], 1 ≤ i ≤ n, which equals the number of cells added to (the original) y after iteration i of the original loop. Next, we allocate a contiguous block of cons cells of length d[n], i.e., of length equal to the total number of cells added to y. This block is pointed to by z. In loop (3), we copy each appended sublist to its destination within z. The function copy_to takes as arguments a list (pointer) and a destination address; it copies each cell in the top level of the list into contiguous locations beginning at the destination address. In addition, it causes the cdr pointer of the last cell in the duplicate sublist to point to the next location in memory (the destination address of the next sublist to be copied.) After explicitly setting the cdr pointer of the last cell in z to nil, we append z to y (see discussion of append above). Finally, we compute the value of y[i], for 1 ≤ i ≤ n; this is simply a pointer to y (recall

[g] This has no effect upon the semantics of append; however, it (once again) affects the amount of sublists sharing (this time, by decreasing it) which, in turn, affects the use of rplaca.

```
do i = 1 to n
    y := append(y, x[i]);
```

Figure 10. A Simple Recurrence Involving append

```
(1) doall i = 1 to n
        d[i] := x[i].l
    dist := 1;
(2) do j = 1 to ⌈lg n⌉
        doall i = 1 to n
            if i > dist then
                d[i] := d[i] + d[i-dist];
        dist := dist * 2;
    z := allocate(d[n]);
    d[0] := 0;
(3) doall i = 1 to n
        copy_to(x[i], z.p + d[i-1]);
    (z.p + d[n] - 1)->cdr = nil;
    y := append(y[0], z);
(4) doall i = 1 to n
        y[i] = [y.p, y[0].l + d[i]];
```

Figure 11. A Parallel Version of Figure 10

that we are adding cells to the end of y), with length equal to the sum of the lengths of (the original) y, and the number of cells added during iterations 1 through i of the original loop.

To make this procedure more tangible, let's suppose that n = 4, that y (initially) equals (a b c), and that x[0] = (d e), x[1] = (f g h), x[2] = nil, and x[3] = (i j). The initial state of affairs in shown in Figure 12, and the state after execution of the code in Figure 11 is shown in Figure 13.

In the general case, let us assume that each of the x[i]'s to be appended, along with the original y, has length of approximately m, and that (the original) y is represented using a small (relative to n) number of contiguous blocks. Then the time to perform the operation of Figure 11 using P processors is $T_P = O\left(\frac{n \cdot lg\, n}{P}\right) + O\left(\frac{m \cdot n}{P}\right)$. With the improved version of parallel prefix described above, it is $T_P = O\left(\frac{n}{P} + lg\, P\right) + O\left(\frac{m \cdot n}{P}\right)$. The number of cells used in representing the resulting lists is O(nm).

Using a conventional representation and means of evaluation, this operation would take $T_1 = O(mn^2)$.[h] Apparently in this (arguably anomalous) case, we have achieved a speedup of O(nP), suggesting that considerable improvement could be made in the usual (sequential) means of evaluating such a recurrence.

It is straightforward to extend this type of parallel list operation to the operator cons, or to the case of appending onto the front of a list; in these cases, we must be satisfied with (at best) linear speedup. Notice that if, within a loop such as that in Figure 7 or 10, we decrease the length of a list by a constant number of cells per iteration (via the operation cdr) or increase its length by a constant number of cells per iteration (with cons or append), that there is no need to use parallel prefix to compute the lengths of the intermediate values; in such a case, the length of the list behaves as an induction variable (i.e, it describes an arithmetic sequence).

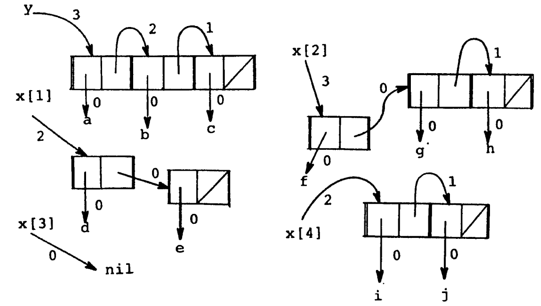


Figure 12. Some Lists to Be Appended as per Figure 11

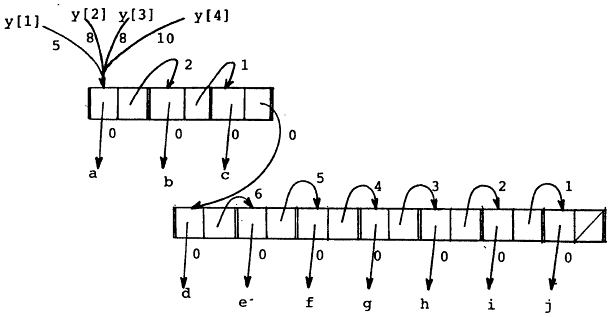


Figure 13. The State After Performing Figure 11

It is an important property of the code generated by the PARCEL compiler that every parallel list-creating operation results in a single, contiguous block of cells, such as z in the above example. This is because such operations make a single call to the function allocate. As mentioned, the performance of code in which the elements of a list must be accessed in parallel depends upon the existence of large, contiguous blocks; consequently, it may be useful to apply techniques which increase the contiguity of lists created sequentially, by repeated calls to cons. See [1], [7].

### The Use of the Storage Scheme in Compiled Lisp Code

Consider the definition of union in Figure 14. The function is recursive, but not (strictly) tail-recursive, as cons may be called after returning from a recursive invocation. The first step taken by our compiler is to build a representation of the function, which is, for the most part, a standard flow-of-control graph. See Figure 15. The variable r is a local temporary variable, used to hold the return value of an invocation of the function. The first optimization to be performed by the compiler in this case is called *recursion splitting*. Recursion splitting is a very general technique for forming loops from recursive functions. It does not require that the function be tail recursive, nor that it be side-effect free. The basic idea is to select a set of *recursive nodes* of the function (nodes in which there is a recursive call to the function) such that there is at most one member of the set along any path through the graph of the function. Then the graph is "split" into two loops (called the forward and backward loops), using these nodes as the points of separation. See Figure Figure 16, in which an abstract function graph is shown. The recursive nodes are shown as double circles, labeled 1, 2, 3, and 4. Suppose that we choose to split using nodes 1 and 2;[i] the result would look something like Figure 17. The first loop in this figure performs the operations of the function as recursive calls are made from members of the selected set of recursive nodes (that is, the portion of the computation of the original function which occurs as the stack becomes deeper due to these calls); the second loop performs the operations of the function as those calls unwind. Recursion splitting works hand-in-hand with our techniques for recurrence recognition and solu-

[h] Note that this is true even if nconc were used; however, as nconc is destructive, only the final value of y could be constructed in this manner. It is arguably rare to repeatedly append to the end of a list; yet it may be that this is because it is ordinarily so expensive (for it seems a natural enough thing to do).

```
(def union (lambda (a b)
   (cond [(null a) b]
         [(member (car a) b)
             (union (cdr a) b)]
         [t (cons (car a)
             (union (cdr a) b))])))
```

Figure 14. The Function union

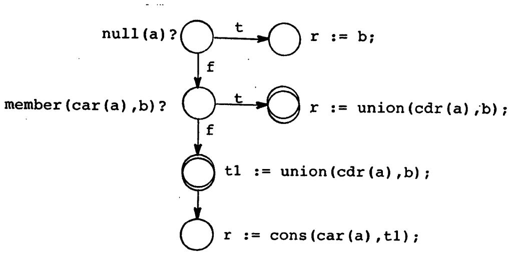


Figure 15. The Graph of union

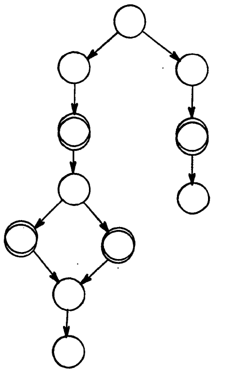


Figure 16. A Function Graph, with Recursive Nodes Indicated

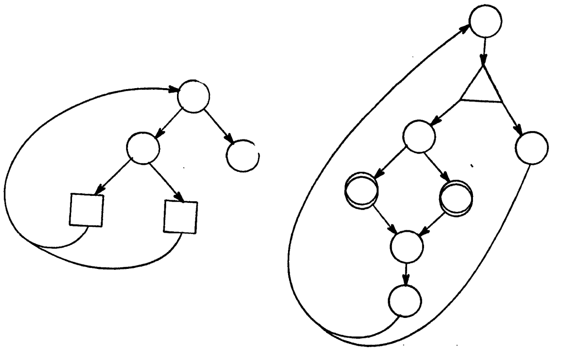


Figure 17. The Graph of Figure 16, After Recursion Splitting

tion. For the transformation to be successful, we must be able to form a recurrence which is used to determine, prior to entering the loops we have created, the number of iterations of each (they have the same number of iterations, just as, in the original function, we must have as many returns from, as we have calls to, the function). This number is equal to the number of consecutive calls made from the nodes used to split the graph. Expanded variables (in the case of Lisp, single pointer variables turned into vectors of pointers) simulate the action of the stack upon the local variables of the function; that is, instead of using a stack to record the state of variables before and after those recursive calls made from those nodes used to split the graph, we represent each variable as a vector, and write the value of the variable before the ith such recursive call into the ith position of the vector. As a consequence, it is much easier to access and manipulate these values in parallel, than if they were dispersed across the run-time stack.

Note that, in this abstract example, the recursive calls in nodes 3 and 4 remain in the modified graph of the function. They become recursive calls to the transformed function, which means that there will be dynamically nested parallel loops if the forward and backward loops are amenable to further optimization. If not, that is, if the bulk of the computation in these loops remains sequential, then we would probably be better off with a more direct translation of the original function definition. The purpose of recursion splitting is to introduce parallelism, and not to remove recursion per se.

Figure 18 shows what union might look like after recursion splitting. Each of a, b, and r is turned into a vector of pointers (in the subsequent transformations, we will turn around and restore b and r to single pointers). The vector e[] is used to record, for each iteration of the forward loop, which of the recursive nodes is reached. This information is used by the corresponding iteration of the backward loop to allow the computation to resume at the correct point. In terms of the original function, this corresponds to the fact that, before a recursive call, a record is made on the stack of the point from which the call is being made, so that execution may proceed from that point when the stack frame is restored. Notice that, intuitively, the backward loop runs "in the opposite direction" of the forward loop; that is, increasing the iteration number of the forward loop corresponds to moving deeper on the run-time stack, while the opposite holds true for the backward loop.

The code finally produced for union, after performing

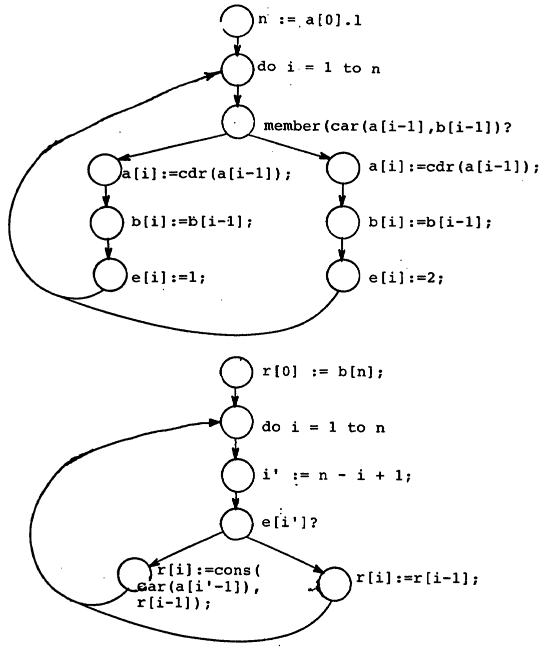


Figure 18. The Graph of union, After Recursion Splitting

(j) The choice of nodes with which to split a function depends upon the recurrences which will result among the variables in the created loops. We wish to find a set of nodes such that the resulting recurrences may be solved in parallel. (We speak of recurrences, but it may be that the variables describe a very simple pattern, such as an arithmetic sequence, or that their values from one iteration to the next are entirely independent; recurrences are the most general case treated by the technique.)

many other manipulations of its graph would look something like that shown in Figure 19. We will describe each of the `doall` loops in turn, assuming that we have $P \leq n$ processors with which to perform the function, where n is the length of (the starting value of) a. (As is frequently the case, no recurrence solution is needed to compute the number of iterations of these loops, as it is simply the length of a list; and because this length is recorded within the pointer to the list, the time to "compute" the loops' upper bound is constant.) Loop (1) assigns all of the intermediate values of a in the loop. As we saw, if a is represented in a constant number of blocks, this will take $T_P = O(n/P)$. The compiler recognizes that the length of a behaves as an induction variable, and knows therefore that its intermediate values may be computed without the use of a recurrence solution technique. Loop (2) assigns the value of `e[i]`, which records the number of the recursive node reached during iteration i of the forward loop. This is also very striaghtforward for the compiler, as it sees that the value of the variable e (which we have "expanded" into a vector) during each iteration is independent of its value in all others. Loops (1) and (2) were created by "loop distribution" from the forward loop (see [12], [7]). (In reality, these loops would be "fused" into a single `doall` loop; see [11].) Loops (3), (4), and (5) were created by distributing the backward loop. Loop (3) places an integer (1 or 0) into each location in the vector `d[]`, indicating whether a cons cell is added to r during a particular iteration of the backward loop. The compiler actually translates this in a more general way, as the length of the sublist appended to the front of r during each iteration. This allows the process of recurrence recognition to made quite formal, as the compiler maps all associative recurrences involving list operations onto arithmetic recurrences involving the intermediate values of the length of the recurrence variable. Loop (4) is the familiar and inefficient version of parallel prefix used to form the values `d[1], d[1] + d[2]`, etc. In the code generated by the compiler, this would be a function call and not written explicitly into the code. Before entering loop (5), space is allocated for the cons cells added by `union` to the front of b. In loop (5), each added cell is copied into its final resting place within this contiguous block of cells. Again, the compiler translates this in a more general way; a `doall` loop is generated, in which every sublist appended to the front of the return value of the function is placed at its destination position within z; a sublist of length zero causes no such placement to occur. We append the block z to the front of b, and return the result. Using an efficient algorithm for parallel prefix, the whole process will take $T_P = O\left(\frac{n \cdot T(\text{member})}{P}\right) + O(lg\, p)$, where T(member) is the time for a call to `member`. Now, `member` is also quite easily made parallel; it, in turn, calls `equal`, which is as well. However, in reality a function such as `union` is likely to appear near the "bottom" of the calling graph of a program, meaning that unless we have either a very large machine, or only procedures which we cannot make parallel "above" `union` in the calling graph, `union` is unlikely to receive a large number of processors. If it is always evaluated sequentially, a translation closer to the original would be better, as (for one thing) the code produced will be smaller than for the transformed version.

Finally, let's examine the definitions of `quicksort` and `split` and given in Figure 20. First consider the driving routine, quicksort. Recursion splitting fails here, as the recurrence defined by the operation `split` cannot be solved in parallel by our methods. (In effect, the compiler sees a recurrence such as `l[i+1] := car(split(cdr(l[i]), car(l[i]), nil, nil))` which must be solved sequentially.) That attempt failing, the compiler inserts a forking directive, causing the two recursive calls to quicksort to be executed simultaneously, resulting in a process tree.

```
     n := a[0].l;
(1)  doall i = 1 to n
         a[i] = nthcdr(a[0], i);
(2)  doall i = 1 to n
        if member(a[i], b) then e[i] := 1;
                                else e[i] := 2;
(3)  doall i = 1 to n
         i' := n - i + 1;
         if e[i'] = 1 then d[i] := 1;
                           else d[i] := 0;
     dist := 1;
(4)  do j = 1 to ⌈lg n⌉
        doall i = 1 to n
           if i > dist then
              d[i] := d[i] + d[i-dist];
        dist := dist * 2;
     z := allocate(d[n]);
     d[0] := 0;
(5)  doall i = 1 to n
         i' := n - i + 1;
         if e[i'] = 1 then
             (z.p + d[i'] - 1)->car :=
                 car(a[i'-1]);
             (z.p + d[i'] - 1)->cdr :=
                 [z.p + d[i'], z.l - d[i']];
     r := append(z, b);
     return(r);
```

Figure 19. A Parallel Translation of `union`

```
(def quicksort (lambda (l)
 (cond
   [(null l) nil]
   [t (let ((l_and_r (split (cdr l) (car l)
                               nil nil)))
        (append (quicksort (car l_and_r))
                (list (car l))
                (quicksort (cadr l_and_r)))
      )]))))

(def split (lambda (l)
 (cond [(null l) (list left right)]
       [(lessp (car l) partition)
          (split (cdr l) partition
                 (cons (car l) left) right)]
       [t (split (cdr l) partition
                 left (cons (car l) right))]
    )))
```

Figure 20. The Function `quicksort`

Now consider the definition of `split`. Recursion splitting succeeds here (trivially), resulting in a function which takes $T_P = O\left(\frac{n}{P} + lg\, P\right)$, on P processors, where n is the length of the first argument to `split`. The fact that `split` is tail recursive means that the backward loop will be empty (non-existent).

Now let's consider how the transformed versions of the functions would work together. Assume that n processors are available to quicksort, where n is the length of the list to be sorted; all n participate in the operation `split`, which thus takes $T_P = O(lg\, n)$, given an input list represented in a constant number of blocks. In the recursive calls to quicksort, assume that the processors are again divided according to the lengths of the sublists to be recursively sorted. Upon the return (join) of these two recursive calls, all n processors are available to perform the append operation; hence the two sorted sublists, along with the list consisting of the single partitioning element, are copied to a single contiguous block of storage (see the discussion above, concerning parallel copying during append operations). Assuming that we have input lists in random order, it can be shown that the average time to perform the entire sort will be $T_P = O(lg^2 n)$. In the case in which there are $P < n$ processors available, the average running time is $T_P = O\left(\frac{n\,lg\,n}{P}\right) + O(lg\,P \cdot lg\,n) - O(lg^2 n)$.

There is an interesting point here, concerning append. Each recursive call to quicksort returns a list which is in a single contiguous block. Because the complexity of split is $O(n/P + lg\,P)$, the time to copy the two results returned by recursive calls into a single contiguous block ($O(n/P)$) will always be less than the time to split the original list which was passed to those calls, meaning that the complexity of the algorithm is not affected adversely by the use of the "copying" version of append. On the contrary, it is only possible to achieve this running time when the copying version of append is used, for otherwise the recursive calls will return lists consisting of (within a constant factor) the same number of blocks as elements, causing append to require linear time.

## Conclusion

We have presented a storage scheme for s-expressions which facilitates the parallel creation and use of lists. The representation may be inexpensively used to represent s-expressions of more general shape, and allows a more flexible sharing of sublists than conventional representations. The PARCEL compiler, assuming this representation, produces code for a shared memory multiprocessor from programs written in a subset of the Scheme dialect of Lisp.

## References

[1] D.W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, January 1979.*

[2] J.R. Beckman-Davies, "Parallel Loop Constructs for Multiprocessors," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-81-1070, May 1981.

[3] D. Friedman, C. Haynes, E. Kohlbecker, M. Wand, "Scheme 84 Interim Reference Manual", Technical Report No. 153, Computer Science Department, Indiana University, Bloomington Indiana, January 1985.

[4] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared-Memory Parallel Machine," *IEEE Trans. on Computers*, Vol. C-32, No. 2, pp. 175-189, February 1983.

[5] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "CEDAR -- A Large Scale Multiprocessor," *Proc. of International Conference on Parallel Processing*, pp. 524-529, August 1983.

[6] R.H. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, October 1985, pp 501-538.

[7] W.L. Harrison, "Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor", CSRD Report No. 565, Center for Supercomputing Research and Development, University of Illinois, Urbana, March 1986

[8] D.J. Kuck, *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York, 1978.

[9] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference*, October, 1980.

[10] G.L. Steele, "Destructive Reordering of CDR-Coded Lists," AI Memo No. 587, Massachusetts Institute of Technology Artificial Intelligence Laboratory, August 1980.

[11] A. Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," Ph.D. Thesis, University of Illinois at Urbana-Champaign, CSRD Report No. 520, 1985.

[12] M.J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-82-1105, 1982.

# The Classifier System: A Computational Model That Supports Machine Intelligence

Stephanie Forrest

Teknowledge Inc.
1850 Embarcadero Road
P.O. Box 10119
Palo Alto, California 94303

Abstract -- The Classifier System is a parallel computational model that supports two important aspects of machine intelligence: machine learning algorithms and operations on high-level "knowledge" structures such as those frequently used in artificial intelligence research. The Classifier System formalism is presented and previous research using the Classifier System in conjunction with two learning algorithms is reviewed. A Classifier System implementation of the knowledge representation language KL-ONE is described, and analytical results are described that show that the implementation exploits the parallelism of the Classifier System.

## Introduction

Researchers in artificial intelligence (AI) are exploring various models of parallelism, both for the purpose of improving performance of AI programs and to express models of intelligent behavior. The possibility that parallel architectures could implement AI models directly has encouraged the development of parallel models that might have both efficient hardware implementations and plausibility as intelligent systems.

Performance improvements can be obtained by using parallelism to improve the execution speed of current languages in which most AI programs are implemented. Projects to construct parallel implementations of Lisp [12], [10] and Prolog [4] fall into this category. Parallel hardware is also being designed to help traditional AI programs execute more efficiently. The Butterfly machine under development at Bolt Beranek and Newman, and the FAIM [6] and NON-VON [19] machines are three examples of such hardware projects.

A second approach is to consider new intermediate representations out of which intelligent systems can be built. Projects such as ABE [7] provide computational units that can be readily composed into large knowledge systems and are amenable to implementation on many different multi-processing systems.

Yet another approach is to build machines that implement parallel models of intelligent behavior more or less directly. The most prominent example of this is the Connection Machine architecture [13] being developed by Thinking Machines Inc. However, there are several other proposed computational models for which hardware implementations do not yet exist that claim to express some aspect of intelligence. These include the Classifier System [18], the Boltzmann machine [14], and the NETL machine [8]. These models all have some features in common, such as massive parallelism, processors with limited computational power, and a high degree of connectivity between processors. The Classifier System and the Boltzmann machine are notable in that the models have been designed expressly to support learning; however, it has not been clear how well these two "fine-grained" models will be able to represent and process high-level symbolic structures.

The research described in this paper explores the question of how well the Classifier System can represent and process high-level symbolic knowledge structures. To explore this issue, I selected a high-level knowledge representation language (KL-ONE) that has well-defined algorithms for accessing and manipulating information stored in knowledge structures, and studied how well the Classifier System could support its facilities. The remainder of the paper is divided into four sections: The Classifier System, Methodology, Results, Discussion, and Conclusions.

## The Classifier System

The Classifier System [18] is a massively parallel model developed by John Holland in which each computational unit (classifier) has very limited processing power, and in which there is no global supervisor. Every processor can communicate with every other processor, and, as will be shown, the extent of this communication can be controlled easily. Each processing element is highly standard, which makes the Classifier System a promising candidate for implementation in hardware. The Classifier System is well suited for problems in artificial intelligence because (1) it possesses a high degree of parallelism, (2) it allows processors to communicate with one another flexibly, (3) learning mechanisms can be applied at several levels within the system, and (4) high-level symbolic representations can be expressed within the formalism.

The Classifier System is modeled on the idea of production systems [5]. It contains a data base of production rules, called classifiers. Each classifier can perform one action, that of adding messages (bit strings) to the short term memory (called the message list). At any instant the state of the message list determines which classifiers are eligible to write information to the message list at the next time step.

Each classifier, or production rule, consists of a condition part and an action part. The action part specifies exactly one action, while the condition part may contain many conditions (pre-conditions of activation). Rules with more than one condition are referred to as "multiple-condition classifiers." A multiple-condition classifier must have each of its pre-conditions matched in a single time step for it to be activated (although it is not necessary that the same message match each condition). The conditions and actions are fixed-length strings over the alphabet (1,0,#) where # denotes "don't care" and 1 and 0 are literals. The determination of whether or not a specific message matches a condition is a

0190-3918/86/0000/0711 $01.00 © 1986 IEEE

logical bit comparison on the defined (1 or 0) bits. If a "not" condition is used, the condition is fulfilled just in the case that no message on the message list matches it. The #'s in the condition part designate "don't care" positions in the sense that they match either 1 or 0. The action part of the classifier determines the message to be posted to the message list: if the i-th symbol, $A_i$, of an action is 0 or 1, the i-th symbol of the posted message will be $A_i$. Otherwise, $A_i$ is #, and the i-th message bit will equal the bit matched by the i-th bit of the distinguished condition.[a]

Each classifier may be regarded (functionally) as a separate processor that takes messages as input and produces messages as output. The configuration of the individual classifier determines which messages are accepted as input and how accepted messages are transformed into output messages. A classifier is activated at a given time step if its pre-conditions are satisfied by at least one message which appeared on the message list at the previous time step. The message list is completely rewritten once per time step so that each message has a duration of exactly one time step. Thus, the primary action of the system is a loop in which all of the classifiers access the current message list, each determining if its pre-conditions have been met, and if so, posting its own output message(s) at the next time step. This process continues until the system has iterated a fixed number of times or, in some systems, until the message list remains unchanged (quiescent) for two successive time steps. All external communication (input and output) is via the message list. As a result, all internal control information and external communication reside in one data structure.

As a simple example, consider the following four bit (n = 4) classifier system:

```
#00# => 1101

#101
###1 => ##1#

~1111 => 1111.
```

This classifier system has three classifiers. The second classifier illustrates multiple-conditions, and the third contains a negative condition. If an initial message, "0000" is placed on the message list at time T0, the pattern of activity shown in Figure 1 will be observed on the message list:

| Time Step | Message List | Activating Classifier |
|---|---|---|
| T0: | 0000 | external |
| T1: | 1101 | first |
| | 1111 | third |
| T2: | 1111 | second |
| T3: | | |
| T4: | 1111 | third |

**Figure 1:** Example Classifier System Behavior

The final two message lists (<empty> and 1111) will continue alternating until the system is turned off. At T1, one message (1101) matches the first (distinguished) condition and both messages match the second condition. Pass through is performed on the first condition, producing one output message for time T2. If the conditions had been reversed (###1) distinguished), the message list at time T2 would have contained two identical messages (1111).

One apparent drawback of the Classifier System is that each classifier reads (examines in its entirety) a potentially large message list on which most of the messages may not be relevant. Having each classifier read the entire message list introduces a time consuming search. However, it is possible to arrange the system in such a way that the messages are routed directly to the classifiers that they will activate. Because the format for expressing the condition parts of a classifier is so constrained, it is possible to sort the conditions of any given list of classifiers so that messages are routed efficiently. If this were done, each classifier would only have to read the relevant messages. Messages that were relevant to many classifiers would still be effectively global, but messages that were only relevant to one classifier would only be read by that classifier. This data-flow approach would not change the overall behavior of the system although it would affect the system's efficiency. Since in most applications, the size of the message list remains small with respect to the total number of classifiers, the actual speedups might not be significant.

In large scale parallel systems such as the Classifier System, the issue of design is central. Design issues arise in two ways for the Classifier System: in deciding which external classifiers are to be generated, and in deciding which external messages are to be placed on the message list and when. As the number of classifiers increases, it quickly becomes impossible to do this by hand. Two automatic approaches have been explored: "learning" and "compiling." The process of compiling can be viewed as mapping high-level structures onto lower-level operations ("top down"). Likewise, some kinds of learning (for example, genetic algorithms) can be viewed as the gradual emergence of higher-level structures from a random assortment of low-level processes; systems using these kinds of learning organize themselves from the "bottom up."

[a] For multiple-condition classifiers, this interpretation of an action is ambiguous since it is not clear what it means to simultaneously perform "pass through" on more than one condition. The ambiguity is resolved by distinguishing one condition to be used for pass through. By convention, this will always be the first condition. Another ambiguity arises if more than one message matches the distinguished condition in one time step. Again by convention, in my system I process all the messages that match this condition. The example illustrates this procedure.

Learning algorithms have been used with Classifier Systems by Holland [18], Smith [20], [21], Booker [1], Wilson [22], and Goldberg [11]. Each of these systems has been designed for a specific purpose and relies on adaptive algorithms to control system behavior. In each case, the researcher was able to demonstrate learning for a complex task: Holland's system learned control routines for operating in a two-dimensional environment, Smith's system learned poker-playing strategies, Booker's and Wilson's systems simulated a hypothetical organism that learned to locate resources and avoid noxious stimuli in an uncertain environment, and Goldberg's system learned to control gas pipeline operations.

Access to the message list can be limited by choosing an upper bound for the number of active messages at any one time. In current systems this is typically a small number (for example, thirty-two). The classifiers that are potentially active (those whose conditions are matched by messages on the current message list) then bid to put their messages on the list, and those with the highest bids are allowed to do so. The bid of a classifier is dependent on at least two components: (1) the specificity of the classifier's conditions and (2) the strength of the classifier.[b] There are two principal ways in which learning is used to control a Classifier System whose message list is of limited size. (1) by controlling write access to the message list and (2) by controlling which classifiers are in the data base of rules. Two different algorithms are used for each of these functions.

The first, the "bucket brigade" [16], adjusts the strengths of the classifiers over time, rewarding those classifiers that have contributed to good solutions and punishing those that do not prove useful. Classifier strength is increased when the system produces a "good" external response. Thus, reward is ultimately dependent on the system's performance in its external environment.

The choice of which classifiers are in the data base is controlled by the Genetic Algorithm [15]. The algorithm is used periodically throughout the operation of the Classifier System to evaluate which classifiers are doing well (contributing to useful solutions) and which ones are not doing well. The evaluation is based on the current strengths of each classifier. Based on this evaluation, weak classifiers are eliminated from the data base, strong classifiers are retained, and new classifiers are generated by applying "genetic operators" to previously successful classifiers in the hopes of generating even more successful recombinations.

## Methodology

In order to study the relation between the bit-level operations of the Classifier System and higher-level symbolic operations, I selected one knowledge representation language, KL-ONE [3], and showed how both its data structures and accessing algorithms can be implemented using the Classifier System.

A knowledge representation system consists of data structures (facts known to the system and the relations between the facts) and interpretive procedures used to access them in an orderly fashion. KL-ONE is a knowledge representation system in which information is represented as a directed graph (called a semantic network).

The implementation takes the form of a compiler, mapping "high-level" semantic network definitions onto the Classifier System. In this context, the Classifier System is properly viewed either as a lower-level target language or as a specification for an abstract parallel machine. An external command processor runs the Classifier System, providing input (and reading output) from the "classifier program." The parallel algorithms are formulated as a sequence of queries to the Classifier System representation of a KL-ONE network. The queries are initiated by placing a set of messages on the message list, allowing the system to iterate for a fixed number of cycles, and then reading the new set of messages from the final message list. Figure 2 illustrates the organization of the implemented system.

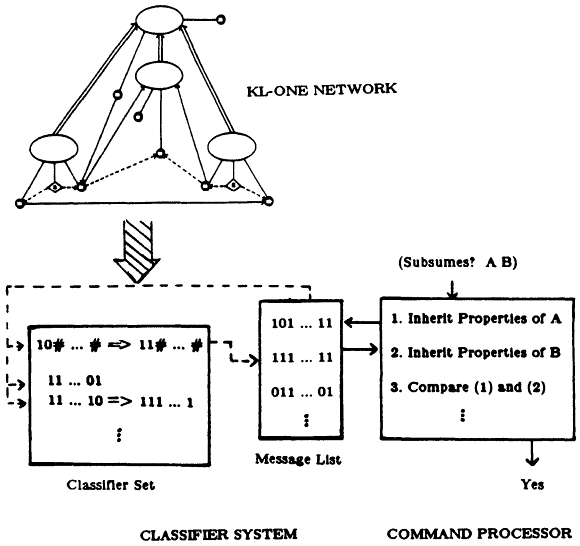


**Figure 2:** Classifier System Implementation of KL-ONE

KL-ONE is one of the few knowledge representation systems for which both the data structures and the operations are well-defined. The central operation in KL-ONE is that of "classification." In its most general formulation, classification is the problem of how to relate new information to an existing knowledge base. In network-based systems, this becomes the problem of deciding which links to add between new and old nodes when incorporating new structures into the network. Of the various knowledge representation paradigms in use today, the KL-ONE family has focused on the issue of classification most precisely.

[b] Recently, a third component has been added. This is called "support," and it corresponds roughly to the number of previously active classifiers that thinks the bidding classifier should be active now.

KL-ONE organizes descriptive terms into a multi-level structure which allows properties of a general concept, such as "mammal," to be inherited by more specific concepts, such as "zebra." This allows the system to store properties that pertain to all mammals (such as "warm-blooded") in one place but to retain the capability of associating those properties with all concepts that are more specific than mammal (such as zebra). In KL-ONE, the multi-level structure is easily represented as a graph, where the nodes of the graph correspond to concepts and properties, and the links correspond to relations between nodes.

In KL-ONE, classification is the process of deciding where a new term (a subgraph) should be located in an existing network. The term may be a single concept, or more likely, a complex description built out of many concepts. The classification procedure can be expressed as the procedure of deciding which concepts are "above" (more general than) and "below" (more specific than) an incoming concept. If a concept A is more general than another concept B (based its definitions), then A is said to *subsume* B. The procedure for deciding whether one concept subsumes another is computationally expensive; the procedure has been shown to be NP-Complete for a related knowledge representation formalism [2].

## Results

The development of the system was divided into several phases: (1) implementing the Classifier System in software, (2) designing and implementing a program that translates KL-ONE networks into production rules for the Classifier System, (3) developing the parallel algorithms to determine subsumption, (4) validating the algorithms, and (5) analyzing the complexity of the system. The Classifier System simulation program, the compiler, and the command processor are all implemented in Franz Lisp, and run on a Vax 11-780 using the Unix Operating System.

The algorithms that have been developed can be divided into two classes: high-level (with respect to the Classifier System) constructs that should be of general utility to other users of the Classifier System, and special-purpose algorithms for classification in KL-ONE. The first class provides a set of general-purpose operations that proved useful for developing KL-ONE-specific algorithms. The general-purpose operations include boolean operations on sets of messages (intersection, complementation, etc.), stack operations (push and pop), and some numerical operations (find the maximum or minimum of a set of numbers, compare two numbers, and addition). For KL-ONE, algorithms for two major operations were developed: (1) a decision procedure for subsumption, and (2) a search procedure that finds the set of *Most Specific Subsumers*[c] (MSS). The sequential version of these algorithms is described in [17] and the parallel version in [9].

The successful implementation of KL-ONE using the Classifier System demonstrates that the Classifier System is capable of representing complex data structures and that complex computations can be performed within the Classifier System formalism. However, this alone does not demonstrate that the implementation is "reasonable." The reasonableness of the implementation has been evaluated using formal complexity measures.

Four measures of complexity were considered: length of computation (in simulated time steps), the size of each processor, the number of processors, and the amount of inter-processor communication. For the Classifier System, these four measures are interpreted as the number of time steps required to complete a computation, the number and length of conditions for each classifier, the number of classifiers used to represent a KL-ONE network, and the maximum size of the global message list. A reasonable implementation is one in which data are not stored redundantly (the number of classifiers and the number and length of conditions grow linearly with the size of the network), the size of the global message list remains small with respect to the total number of classifiers (inter-processor communication should not increase disproportionately to the size of the network), and the parallel algorithms are computationally more efficient than their sequential counterparts (the speed-ups are expected to be related to the fan-out of the underlying structure).

A detailed analysis of the parallel algorithms using these four measures of complexity appears in [9]; in summary, the analysis shows that the implementation meets the above criteria. The number of classifiers required to represent each KL-ONE network is directly proportional to the size of the network definition. The size of each classifier grows as the log of the number of nodes in the network. The maximum size of the message list is related to the topology of the KL-ONE network being processed; for networks in which the fan-out is greater than the fan-in (the expected case), the maximum size of the message list grows with the depth of the network. The Subsumption algorithm and the search for Most Specific Subsumers each run in time proportional to the depth of the network.[d] Because the search for Most Specific Subsumers invokes the Subsumption test, the overall complexity for the algorithms taken together is proportional to the $(\text{depth})^2$.

[c] A is a Most Specific Subsumer of B iff
A SUBSUMES B, and
there does not exist a C, such that
A SUBSUMES C AND C SUBSUMES B.

[d] The implemented version of the search for Most Specific Subsumers may run somewhat slower than this for some networks. However, it would be straightforward to add one field (a small number of bits) to each classifier and guarantee that the algorithm always ran in time proportional to the depth of the network (with a larger message list size). Since the performance of the algorithms is highly dependent on the conformation of the networks it processes and there is very little data available about the conformation of typical networks, it was not clear at the time of implementation whether this was a worthwhile optimization.

## Discussion

In the current implementation, the algorithms rely to some extent on the host language (Lisp) in which the command processor is implemented. That is, some of the algorithms take advantage of the control structures and data management facilities of Lisp beyond using it to formulate queries to the Classifier System. This is an important issue as it would be easy to hide significant amounts of complexity in the processing of the host language and make it appear that the parallelism was accomplishing more than it really was. On the other hand, it seems unreasonable to force intrinsically sequential operations into the parallel formulation when there is an adequate sequential language immediately available. In the extreme, it would be possible to generate a set of classifiers that could answer the subsumption question with one query (with no interference from the host language)[e], but the number and length of classifiers and the time to translate KL-ONE networks into the classifier representation would be unreasonably large.

The parts of the algorithm that have been implemented in Lisp are either natural components of the host language (such as invoking the Classifier System) or they are operations that could be made parallel by adding a constant number of bits (to be used as tags) to each classifier. In particular, the embedding language (Lisp) is used in four ways: (1) to translate symbolic queries such as "(Subsumes? A B)" into binary messages and translate them back after the query, (2) to invoke the Classifier System, (3) to store the results of queries (for example, a list of messages) that will be needed again later, and (4) to control the highest-level sequences of queries through the use of conditional and iterative constructs. The first two of these are appropriate for the command processing role that is played by the embedding programs. The second two are only a matter of convenience for the current simulation and could be implemented in the Classifier System.

The implementation revealed a set of techniques that are useful for controlling the Classifier System. These are tagging, the use of negative conditions as triggers, computations that process one bit position at a time, and synchronization.

Tagging, in which one field of the classifier is used as a selector, is used to maintain groups of messages on the message list that are in distinct states. Tagging allows the use of specific operators that are defined for particular states. This specificity also allows additional "layers" of parallelism to be added by processing more than one operation simultaneously. In these situations, the messages for each operation are kept distinct on the global message list by the unique values of their tags.

Negative conditions activate and deactivate various subsystems of the Classifier System. Negative conditions are used to terminate computations and to explicitly change the state of a group of messages when a "trigger" message is added to the list. When the trigger message appears, it violates the negative condition and that classifier is effectively turned off.

Computations that proceed one bit at a time illustrate two techniques: (1) using control messages to sequence the processing of a computation, and (2) how to collect and combine information from multiple independent messages into one message. Sequencing will always be useful when a computation is distributed over multiple time steps instead of being performed in one step. Collection is important because in the Classifier System it is easy to "parallelize" information from one message into many messages that can be operated on independently. This is most easily accomplished by having many classifiers that match the same message and operate on various fields within the message. The division of one message into its components takes one time step. However, the recombination of the new components back into one message (for example, an answer) is more difficult. The collection process must either be conducted serially (one bit position at a time) or one classifier must be allocated for each possible message combination (a potentially huge number). Intermediate solutions are also possible.

Synchronization techniques allow one operation to be delayed until another has finished. Synchronization can be achieved by combining tagging with negative conditions.

## Conclusions

The implementation of KL-ONE demonstrates that the Classifier System formalism is a powerful and flexible representation system. This result combined with previous work that demonstrates how the Classifier System can learn suggests that the Classifier System is a promising low-level organization from which intelligent systems can be constructed.

In particular, the implementation is one in which reasonable computation time improvements can be obtained without an unreasonable increase in the number and size of processing units or in the degree of inter-processor communication. The approach that was demonstrated for KL-ONE could easily be extended to other similar inheritance networks.

This particular implementation does not include those features that are specific to the use of adaptive algorithms, such as bidding, support, etc. However, previous experiments that combined learning algorithms with the Classifier System worked with rather small sets of classifiers (usually less than two hundred and often less than one hundred). Further, it is very difficult to "decompile" these systems after significant learning has occurred to determine what role is played by individual classifiers. Using a compilation technique allows the construction of much larger sets of classifiers and it allows complex behavior (such as synchronization) to be identified and studied. One promising direction for further research is to try combining the two approaches to see if the kind of complex behavior demonstrated by the KL-ONE implementation can be learned by the bucket brigade and genetic algorithms.

[e] It is straightforward to show that for any finite function, a Classifier System can be constructed that computes it.

Acknowledgements

The inspiration and the original designs of the Classifier System, the Bucket Brigade, and the Genetic algorithm are John Holland's. In addition, he made many helpful suggestions about the KL-ONE implementation and its significance for which I am very grateful. This research was supported by National Science Foundation Grant DCR-8305830 while I was at The University of Michigan.

## References

[1] Booker, L. *Intelligent Behavior as an Adaptation to the Task Environment.* PhD thesis, The University of Michigan, 1982.

[2] Brachman, R. J. and Levesque, H. J. The Tractability of Subsumption in Frame-Based Description Languages. In *The Proceedings of the National Conference on Artificial Intelligence.* 1984.

[3] Brachman, R. J. and Schmolze, J. G. An Overview of the KL-ONE Knowledge Representaton System. *Cognitive Science* 9(2), 1985.

[4] Campbell, J. A. *Implementations of Prolog.* Ellis Horwood Limited, 1984.

[5] Davis, R. and King, J. An Overview of Production Systems. *Machine Intelligence* (8), 1977. pp. 300-331.

[6] Davis, A. L., and Robison, S. V. The Architecture of the FAIM-1 Symbolic Multiprocessing System. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* AAAI, 1985.

[7] Erman, L., Fehling, M., Forrest, S., and Lark, J. ABE: Architectural Overview. In *Proceedings of the Workshop on Distributed Artificial Intelligence.* 1985.

[8] Fahlman, S. E. *NETL: A System for Representing and Using Real-World Knowledge.* M.I.T. Press, Cambridge, Ma., 1979.

[9] Forrest, S. *A Study of Parallelism in the Classifier System and Its Application to Classification in KL-ONE Semantic Networks.* PhD thesis, The University of Michigan, 1985.

[10] Gabriel, R. P. and McCarthy, J. Queue-based Multiprocessing Lisp. In *1984 ACM Symposium on Lisp and Functional Programming.* ACM, 1984.

[11] Goldberg, D. *Computer-aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning.* PhD thesis, The University of Michigan, 1983.

[12] Halstead, R. H. Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *1984 ACM Symposium on Lisp and Functional Programming.* ACM, 1984.

[13] Hillis, D. W. *The Connection Machine.* M.I.T. Press, Cambridge, Ma., 1985.

[14] Hinton, G. E. and Sejnowski, T. J. Analyzing Cooperative Computation. In *Proceedings of the Fifth Annual Conference of the Cognitive Science Society.* 1983.

[15] Holland, J. H. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Mi., 1975.

[16] Holland, J. H. Properties of the Bucket Brigade. In Grefenstette, John J. (editor), *Proceedings of An International Conference on Genetic Algorithms and Their Applications.* Texas Instruments, Inc. and NCARAI, 1985.

[17] Lipkis, T. A. *A KL-ONE Classifier.* Technical Report Consul Note #5, USC/Information Sciences Institute, Marina del Rey, Ca., 1981.

[18] Michalski, R.S, Carbonell, J.G., and Mitchell, T.M. (eds.). Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-based Systems. *Machine Learning.* Morgan Kaufmann Publishers, Inc., 1986, pages 593-623, Chapter 20.

[19] Shaw, D. E. *The NON-VON Supercomputer.* Technical Report, Columbia University, Dept. of Computer Science, 1982.

[20] Smith, S. *A Learning System Based on Genetic Algorithms.* PhD thesis, The University of Pittsburgh, 1980.

[21] Smith, S. Flexible Learning of Problem Solving Heuristics Through Adaptive Search. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* AAAI, 1983.

[22] Wilson, S. *Adaptive 'Cortical' Pattern Recognition.* Technical Report, Research Laboratories, Polaroid Corporation, 1982.

# CONTINUOUS SPEECH RECOGNITION ON A BUTTERFLY ™ PARALLEL PROCESSOR

Lynn Cosell, Owen Kimball, Richard Schwartz, Michael Krasner

BBN Laboratories
10 Moulton St.
Cambridge, MA 02238

## ABSTRACT

This paper describes the implementation of a continuous speech recognition algorithm on the BBN Butterfly$^{TM}$ Parallel Processor. The implementation exploited the parallelism inherent in the recognition algorithm to achieve good performance, as indicated by execution time and processor utilization. The implementation process was simplified by a programming methodology that complements the Butterfly architecture. The paper describes the architecture and methodology used and explains the speech recognition algorithm, detailing the computationally demanding area critical to an efficient parallel realization. The steps taken to first develop and then refine the parallel implementation are discussed, and the appropriateness of the architecture and programming methodology for such speech recognition applications is evaluated.[1]

## INTRODUCTION

This paper describes research to investigate the uses of parallel computation for continuous speech word recognition. Our goal in this work is to determine the extent to which continuous speech recognition algorithms can make use of parallel processing to achieve real time speeds. Our approach has been to develop parallel versions of an existing recognition algorithm on BBN's Butterfly Parallel Processor.

## BUTTERFLY

The Butterfly Parallel Processor [1] is composed of multiple (up to 256) identical nodes interconnected by a high-performance switch. Each node contains a processor and memory. The switch allows each processor to access the memory on all other nodes. Collectively, these memories form the shared memory of the machine, a single address space accessible to every processor. All interprocessor communication is performed using shared memory. Typical memory referencing instructions accessing local memory take about 2 microseconds to complete, whereas those accessing remote memory take about 5 or 6 microseconds.

The Butterfly Parallel Processor is a multiple instruction multiple data stream (MIMD) machine in which each processor node executes its own sequence of instructions, referencing data as specified by the instructions. Processor Nodes are tightly coupled by the Butterfly switch. Tight coupling permits efficient interprocessor communication and allows each processor to access all system memory efficiently. The Butterfly Parallel Processor is expandable to 256 Processor Nodes. Each Processor Node contains a Motorola MC68000 microprocessor, an optional floating-point coprocessor, from 1 to 4 MBytes of main memory, a co-processor called the Processor Node Controller, memory management hardware, an I-O bus, and an interface to the Butterfly switch. The particular machine that was used for development in this project was a 16 processor machine with 1 Mbyte of memory on each Processor Node. It did not have hardware support for floating point arithmetic.

## WORD RECOGNITION ALGORITHM

The problem of speech recognition requires that we map an analog signal onto a sequence of words that comprise sentences. However, the identification of particular speech sounds (phonemes) from this signal is made difficult due to the variability that occurs in speech production. This variability is due to the effect of neighboring speech sounds (coarticulation), and a variety of other effects that all combine to make the same speech unit appear differently each time it occurs.

Our recognition algorithm is based on the explicit modeling of variability in speech through the use of probabilistic Hidden-Markov Models (HMMs) of speech sounds (phonemes) in various phonetic contexts [2]. Each phoneme consists of three states, which are associated with acoustic events corresponding roughly to the beginning, middle and end of the phoneme. There is, associated with each pair of states, a transition probability $a(j|i)$ which is the probability of going to state $j$ given that the process is in state $i$. Unlike a Markov chain, in which each state has associated with it a single output, each HMM state has an output probability density function (pdf) $P(x|i)$ that gives the probability of each possible output symbol $x$, given that the process is in state $i$.

Rather than use actual segments of the speech signal as output symbols, we can represent the speech signal as a sequence of spectra that occur at discrete

[1] This work was sponsored by the Defense Advanced Research Projects Agency and was monitored by the Naval Electronics System Command under Contract No. N00039-85-C-0313.

time intervals. Furthermore, each of these spectra can be approximately characterized as one of a small number of spectral types (256 in our system), which are determined using a clustering procedure. The spectral characterizations, each a single number, then become the possible output symbols. Words can be modeled as concatenations of phoneme HMMs that have been modified to take into account the contextual effects of the word.

One approach to understanding HMMs is to imagine them in a synthesis role, where they are used to produce spectral sequences. Starting from the initial state of the model, we randomly choose the next state according to the transition probabilities on the arcs leaving the initial state. Whenever the process is in an acoustic state, we randomly pick an output symbol according to the state's output pdf. As the process moves from state to state in this manner, it produces a sequence of symbols (speech spectra) as output.

The recognition problem can be viewed as the inverse of the synthesis problem: given a sequence of input spectra and a set of models of all possible words, we wish to find the sequence of models that is most likely to have produced the spectral sequence. An adaptation of the Viterbi algorithm solves just this problem, and is the basis for our recognizer.

The basic recognition algorithm finds the path through the states that is most likely to have produced the spectral sequence to be recognized. The algorithm does this by finding the best path to every state at every time given that the path to the previous state was also "best". Each step along these paths has associated with it a "score", which reflects the probability of the step given the spectral sequence and the transition probabilities of the model. The scores are accumulated along the paths, so that, at every time, the best path to any state has a single score. The best path to a particular state, S, at time t, is determined by considering all possible predecessor states (states which have transitions to S) at time t−1 and the best path to each of these. The score for a path to S is then the combination of the path score to the best predecessor state and the score for the step from that predecessor state to S.

The central computation in the algorithm is: for each time interval, update the scores for all states. Figure 1 schematically illustrates the scoring procedure for a single state in a word. In this figure, the score for state n at time t (the $\alpha_n(t)$ in the lower right corner) is being computed based on the scores for three states computed at time t−1 (the three circles on the left of the figure) each multiplied by the corresponding transition probability of going to state n. The new score for state n is just the maximum of the entering scores multiplied by the probability of the spectrum $x_t$ at time t, at state n, $p(x|n)$.

This scoring procedure is applied to all states in each word for all time frames in the utterance. The scores computed for terminal states (ends of words) are special. They are compared and only the maximum terminal score is saved, as are the word that produced it and the start time for the word. The largest terminal score for a time frame is used as the score for all word-initial states in the next time frame. In addition, the current state score, $\alpha_n(t)$ is compared against the largest state score, $\alpha_{Best}(t)$, encountered so far for the current time frame, and replaces it, if appropriate. This is used to derive a normalization factor ($NF = 1/\alpha_{Best}(t)$), which is used to prevent arithmetic underflow.

When all the time frames in the utterance have been processed in this way, the best sequence of words is determined. The maximum terminal score at the end of the utterance specifies the last word of the utterance. The start time of this last word is the end time for the previous word, so the maximum terminal score at this time indicates which word should be selected as the second-last word in the theory, and so on, back to the beginning of the utterance.

## SINGLE PROCESSOR IMPLEMENTATION

The first step toward a parallel implementation was to bring up the speech recognition program on a single processor of the Butterfly Parallel Processor. The existing VAX implementation depended on the file system to store the large amounts of data which included the transition probabilities and the pdfs for the word models. This data totaled 1.5 Mbytes. Storing this amount of data on the Butterfly required using some parallel memory management techniques to allocate shared memory on multiple nodes.

The VAX (and the first Butterfly implementation) used floating-point arithmetic. Because floating-point arithmetic is performed in software in our Butterfly Parallel Processor, it seemed likely that the time required for the floating-point arithmetic would hide any overhead and task granularity problems we encountered. We decided to investigate using fixed-point arithmetic.

For storage efficiency, most of the data had been represented as indices into a table of probabilities. This table contained 256 entries, ranging between zero and unity, quantized logarithmically. Therefore, the indices themselves were scaled log probabilities. Multiplication of probabilities in the original program could, of course, easily be converted to addition of corresponding log probabilities. The Viterbi algorithm finds the maximum of the path probabilities to a node, which is equivalent in both log and linear domains. We converted the program to use the indices directly as log probabilities, and obtained the same results as before, but the execution time remained disappointingly long. Measurement tools allowed us to discover that most of the time was being squandered in two ways. First, using double indexing into two-dimensional arrays was quite slow because the 68000 compiler available at the time used double precision multiplies for this kind of address calculation. We converted to register pointers to avoid this situation. Secondly, much time was being spent in the call to the subroutine that performed the word scoring. We simplified the calling sequence by defining many of the arguments globally. These modifications caused the execution time to drop to about two minutes for a 3.5 second utterance, (about the same speed as our optimized VAX program). This execution time seemed low enough for reasonable parallelization measurements.

## UNIFORM SYSTEM

We used the Uniform System approach to obtain a parallel implementation. The Uniform System is a programming methodology supported by a library of high-level functions [3]. It exploits the uniform environment provided by the architecture of the Butterfly Parallel Processor to simplify the problem of load balancing for the memory as well as for the processors. Memory accesses must be organized to avoid memory contention. The load on the processors is balanced when all processors are equally busy and no processor is waiting for another to finish.

Balancing the load on memory is accomplished by spreading out the data evenly across the different physical memories in the machine, under the assumption that this will also spread the accesses fairly evenly, reducing the inefficiency that results when many processors attempt to access the same memory simultaneously. Functions for allocating storage in the shared memory are included in the Uniform System, as are functions that perform block transfers between shared memory and local memory.

The philosophy behind the Uniform System processor management methodology views the processors as a uniform pool of workers, all of which know how to execute the same tasks. Using this methodology, the programmer is only required to supply code that operates correctly when multiple processors execute it at the same time. The processor management is accomplished by first copying the program to each of the processors. In most cases, the program will begin with a section of serial code that is executed on a single processor. To begin executing a section of code on multiple processors -- a FOR loop, for example -- the programmer can use a "task generator" to replace the FOR statement and a "worker routine" to replace the body of the FOR loop. The task generator makes a task descriptor available to all processors, which use it, as they become free, to generate calls to the worker routine. Processors, using this descriptor, execute the routine repeatedly for different index values, until the index has run its range. When all processors have finished, the program, once again serial, continues executing on a single processor.

We decided to use the Uniform System for several reasons. First, the speech recognition algorithm is essentially a single task, executed many times. This fits the Uniform System paradigm very well. Second, being novices, we were attracted by the simplicity of use of the Uniform System. Third, functions in the Uniform System allow automatic timing of the same program run on various numbers of processors, and this provided an easy way of evaluating the performance of the parallel implementation. Finally, because the same program can be run on one or many processors, we believed that debugging the parallel implementation would be simplified.

## PARALLEL IMPLEMENTATION AND RESULTS

In our system, both the training and spectral analysis tasks are performed "off-line" and the results are stored. The recognition system begins by reading in the word models for a speaker. Then, for each utterance, the spectral parameters for the utterance are read and stored. At this point in the program, the actual recognition search task begins. This was the only portion considered for parallel implementation. Our execution time measurements began here and continued until the input utterance had been recognized, that is, a theory for the complete utterance had been obtained. The pertinent portion of the speech recognition program can be abstracted as follows:

```
a. FOR all frames
b.     initialize frame
c.     FOR all words
d.         initialize word
e.         FOR all states
f.             compute state score
g.             IF (new max score)
h.                 replace max score
i.         IF (new max terminal)
j.             replace max terminal
k.     determine normalization
l.     FOR all words
m.         get terminal score
n. determine theory
```

The first parallel version combined lines d) through f) and parts of g) and h) into a single task and used the generator GenOnIndex, which includes a prologue task and an epilogue task in addition to the main task. The prologue task is executed only once by each processor before that processor executes the main task for the first time. In this version, the prologue included line b). Similarly, the epilogue task is executed once by each processor after all main tasks have been completed by that processor. For this program, the central task determined the maximum state score and the maximum terminal score seen by each processor. The epilogue task compared these local maxima against global maxima, replacing the global maxima if necessary. The remainder of the program (lines k-m), including the second FOR loop was executed sequentially, on a single processor. Approximately 9 seconds was spent in the sequential portion of the program when it was run for a 3.5 second utterance. When run on 15 processors, this resulted in less than 50% utilization of the processors.

The next step was to attempt to reduce the sequential portion of the program. We noticed that the second FOR loop (lines l and m), which propagates the best word's score in the current frame to word-initial states for the next frame, could be incorporated into the first FOR loop, effectively changing the program to:

```
a. FOR all frames
b.     initialize frame
c.     FOR all words
d.         get previous frame terminal score
e.         initialize word
f.         FOR all states
g.             compute state score
h.             IF (new max score)
i.                 replace max score
j.         IF (new max terminal)
k.             replace terminal
l.     determine normalization
m. determine utterance.
```

This revision substantially reduced the time spent executing serial code. For 15 processors, the execution time dropped from 15 seconds to 11 seconds for a 3.5 second utterance, and the effective number of processors rose from 6.9 to 11.2, or approximately 75% utilization. The processor utilization is shown in Figure 2.

## CONCLUSIONS AND FUTURE RESEARCH

Our work on this project has shown that the Butterfly architecture is suitable for continuous speech word recognition. The decomposition of the algorithm into tasks that match one word to one frame of input speech provided a granularity that made efficient use of the processors. The memory and processor management functions of the Uniform System made parallelization of the algorithm surprisingly easy and rapid.

In the near future, we hope to extend the current research to include a grammar and larger vocabulary tasks. The grammar will require search of a much larger space that is too large to search exhaustively. The search will have to be pruned, thus presenting a more challenging parallel implementation task.

## REFERENCES

1. R. Thomas, R. Gurwitz, J. Goodhue and D. Allen, "Butterfly Parallel Processor Overview", BBN Laboratories Incorporated, March 6, 1986, pp 9-14.

2. Y.-L. Chow, R. Schwartz, S. Roucos, O. Kimball, P. Price, F. Kubala, M. Dunham, M. Krasner and J. Makhoul, "The Role of Word Dependent Coarticulatory Effects in a Phoneme-Based Speech Recognition System", IEEE Int. Conf. Acoust., Speech, Signal Processing, Tokyo, Japan, April 1986.

3. Thomas, Robert, "The Uniform System Approach to Programming the Butterfly Parallel Processor", BBN Laboratories Incorporated, March 6, 1986.

## FIGURES

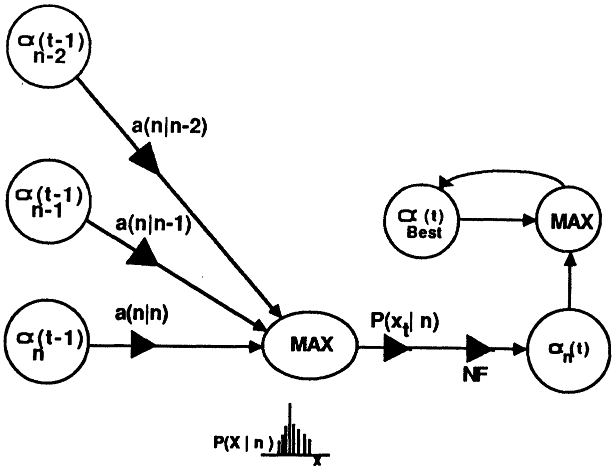


**FIG. 1. STATE SCORE COMPUTATION**

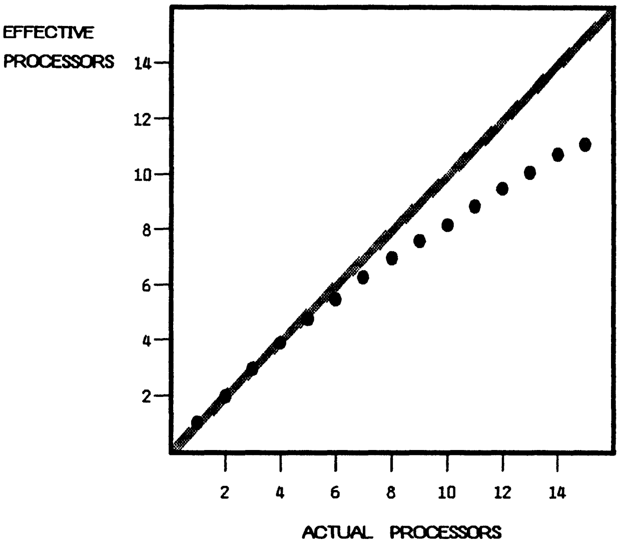


**FIG. 2. PROCESSOR UTILIZATION**
**The gray line is 100% utilization**

# A Hyperconcentrator Switch for Routing Bit-Serial Messages
(Extended Abstract)

Thomas H. Cormen
Charles E. Leiserson

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

## Abstract

In highly parallel message routing networks, it is sometimes desirable to concentrate relatively few messages on many wires onto fewer wires. We have designed a VLSI chip for this purpose which is capable of concentrating bit-serial messages quickly. This hyperconcentrator switch has a highly regular layout using ratioed nMOS and takes advantage of the relatively fast performance of large fan-in NOR gates in this technology. A signal incurs exactly $2\log_2 n$ gate delays through the switch, where $n$ is the number of inputs to the circuit. The architecture generalizes to domino CMOS as well.

## 1 Introduction

The problem of concentrating relatively few communications on many input lines onto a lesser number of output lines must be solved in communication networks of all kinds. In many parallel computing systems, communications are packaged into messages which are routed among the processors. This paper presents a design for a VLSI implementation of a fast concentrator switch suitable for routing bit-serial messages in a parallel supercomputer.

An *n-by-m concentrator switch* has $n$ input wires $X_1, X_2, \ldots, X_n$ and $m < n$ output wires $Y_1, Y_2, \ldots, Y_m$. The switch can establish $m$ disjoint electrical paths from any set of $m$ input wires to the $m$ output wires. A concentrator switch always routes as many messages as possible. Specifically, whenever $k$ out of the $n$ input wires of an $n$-by-$m$ concentrator switch carry messages, one of the following is true:

- If $k \leq m$, then an electrical path is established from each input wire which contains a message to an output wire.
- If $k > m$, then each output wire has an electrical path established from an input wire which contains a message.

When $k > m$, some messages cannot be successfully routed, in which case we say the switch is *congested.* Typical ways of handling unsuccessfully routed messages in a routing network are to buffer them, to misroute them, or to simply drop them and rely on a higher-level acknowledgment protocol to detect this situation and resend them. The switch design in this paper is compatible with any of these congestion control methods.

One way to create a concentrator switch is with a hyperconcentrator switch. An *n-by-n hyperconcentrator switch*[1] has $n$ input wires $X_1, X_2, \ldots, X_n$ and $n$ output wires $Y_1, Y_2, \ldots, Y_n$. The switch can establish disjoint electrical paths from any set of $k$ input wires, for any $1 \leq k \leq n$, to the first $k$ output wires $Y_1, Y_2, \ldots, Y_k$. In other words, we route the $k$ messages to the first $k$ output wires. We can make any $n$-by-$m$ concentrator switch from an $n$-by-$n$ hyperconcentrator switch by simply choosing the first $m$ output wires of the hyperconcentrator switch, $Y_1, Y_2, \ldots, Y_m$, as the $m$ output wires of the concentrator switch.

We can use a sorting network to implement a hyperconcentrator switch. The inputs to the sorting network are 1's and 0's, representing the presence or absence of messages on the input wires to the switch. The sorting of the 1's and 0's, with 1's before 0's, causes the $k$ input messages to occupy the first $k$ outputs.

Many sorting networks, such as Batcher's bitonic sort [6], employ the technique of recursive merging.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract N00014-80-C-0622. Tom Cormen is supported in part by a National Science Foundation Fellowship. Charles Leiserson is supported in part by an NSF Presidential Young Investigator Award.

[1] The terminology is drawn from [11].

0190-3918/86/0000/0721 $01.00 © 1986 IEEE

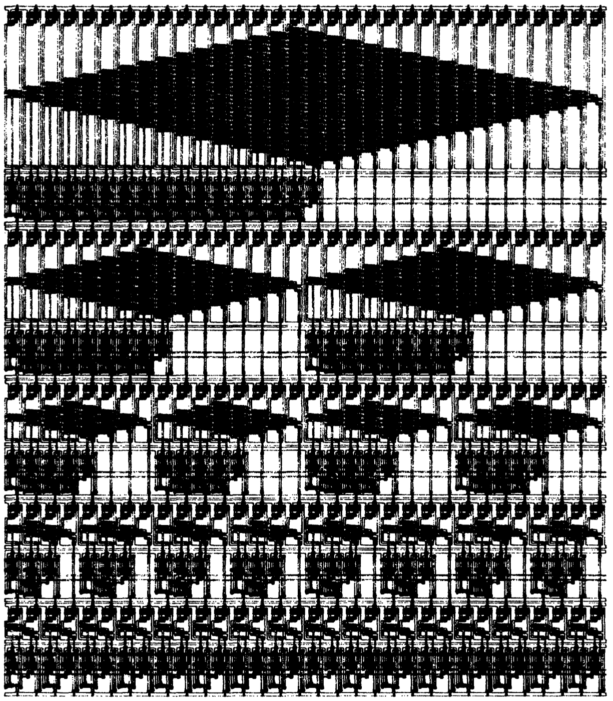


**Figure 1:** An nMOS layout of a 32-by-32 hyperconcentrator switch. The recursive nature of the switch can easily be seen. This implementation includes superbuffers where needed to provide enough drive for high fan-out signals.

A problem of size $n$ is divided into two problems of size $n/2$, which are recursively solved in parallel. The two sorted sets are then merged to produce the solution to the original problem. The recursion requires $\lceil \log_2 n \rceil$ levels, and since each merge step can be performed in $O(\log n)$ time in parallel, the total time to sort $n$ values is $O(\log^2 n)$. Sorting networks of depth $O(\log n)$ are known [1], but they are impractical to use as hyperconcentrator switches because of the large associated constant.

The $n$-by-$n$ hyperconcentrator switch presented in this paper also uses recursive merging, but by taking advantage of the relatively fast performance of high fan-in NOR gates in nMOS technology, each merge takes only 2 gate delays. A signal therefore incurs exactly $2\lceil \log_2 n \rceil$ gate delays in passing through the switch. The switch has a simple design and a regular layout in both ratioed nMOS and domino CMOS technologies. Unlike many concentrator switches in the literature [8,9,10], our switch sets itself up "on-line" when messages are presented to it.

The remainder of this paper is organized as follows. Section 2 covers some basic terminology and describes the message format and timing model upon which the switch is based. Section 3 discusses the merge box, how to use merge boxes to implement the hyperconcentrator switch and describes an nMOS implementation. Section 5 covers some applications which benefit from the switch. Finally, Section 6 contains further remarks about the switch.

## 2 Preliminaries

In this section, we define some basic terminology and notational conventions and present the message format and timing model assumed by the hyperconcentrator switch design.

Bit and boolean values are denoted by "1" and "0", or by "high" and "low", for TRUE and FALSE respectively.

We assume that the hyperconcentrator switch routes *bit-serial messages*. Each message is formed by a stream of bits arriving at a wire at the rate of one bit per clock cycle. The first bit of each message that arrives at an input wire is the *valid bit*, indicating whether subsequent bits arriving on that wire form a valid message or an invalid message. The bit sequence following a valid bit of 1 forms a *valid message*, which we would like to be routed from an input wire to an output wire of the switch. From there it may pass through the remainder of the routing network. A valid bit of 0 indicates an *invalid message*, which does not need to be routed to an output wire. We assume that in an invalid message, not only is the valid bit 0, but so are all the remaining bits in the message.[2]

The valid bits all arrive at the input wires of the hyperconcentrator switch during the same clock cycle, which we call *setup*. An external control line signals setup. Message bits entering through input wires at cycles after setup follow the electrical paths in the switch which are established during setup.

We shall adopt some notational conventions to ease the exposition in the following sections. Uppercase symbols denote wire names and lowercase symbols denote integer values. We shall also use uppercase symbols to denote bit values on the wires they name when the usage is unambiguous. Wire names will usually have subscripts.

## 3 The Merge Box

This section presents the design of the merge box, the key portion of the hyperconcentrator switch architec-

[2] This assumption is easy to enforce — just AND the valid bit into each subsequent bit of the message.

ture. The hyperconcentrator switch consists of many merge boxes, of various sizes, connected as shown in the next section. The design exploits the fast performance of large fan-in NOR gates in nMOS technology, much the same as does a PLA, to merge two sets of messages of any size with two gate delays. The merge box design presented in this section uses ratioed nMOS technology and no pass transistors.

A merge box merges two sets of messages, each set sorted by their valid bits, into one sorted set of messages. A *merge box of size* $2m$, where $m$ is a power of 2, has two sets of input wires $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_m$ and one set of output wires $C_1, C_2, \ldots, C_{2m}$. We require that the lower-numbered wires of both the $A$ and $B$ input sets carry valid messages and that the higher-numbered wires of both the $A$ and $B$ input sets carry invalid messages. That is, if we let $p$ and $q$ be the number of valid messages entering the $A$ and $B$ wire sets respectively, we require that the valid bits appear on the input wires during setup as follows:

$$\begin{aligned} A_1, A_2, \ldots, A_p &= 1 \\ A_{p+1}, A_{p+2}, \ldots, A_m &= 0 \\ B_1, B_2, \ldots, B_q &= 1 \\ B_{q+1}, B_{q+2}, \ldots, B_m &= 0 \end{aligned}$$

During setup, the merge box establishes disjoint electrical connections between the $p + q$ input wires with valid messages and the $p+q$ lower-numbered output wires $C_1, C_2, \ldots, C_{p+q}$ in a combinational fashion, as shown in Figure 2. The connections $C_1 = A_1, C_2 = A_2, \ldots, C_p = A_p, C_{p+1} = B_1, C_{p+2} = B_2, \ldots, C_{p+q} = B_q$ are established, and valid bits appear on the output wires as follows:

$$\begin{aligned} C_1, C_2, \ldots, C_{p+q} &= 1 \\ C_{p+q+1}, C_{p+q+2}, \ldots, C_{2m} &= 0 \end{aligned}$$

These connections are maintained during subsequent cycles for the remaining bits in the message streams to follow.

Figure 3 is a schematic diagram of a merge box for which $m = 4$. This merge box includes eight NOR gates, with diagonal output wires labeled $\overline{C}_1, \overline{C}_2, \ldots, \overline{C}_8$. Each of these NOR gate outputs is inverted to produce the merge box outputs $C_1, C_2, \ldots, C_8$, so we may view the pulling down of a diagonal wire $\overline{C}_i$ to be equivalent to the corresponding output $C_i$ being 1. The NOR gates have fan-ins ranging from just one pulldown circuit (e.g. the gate with output $\overline{C}_8$) to 5 pulldown circuits (e.g. the gate with output $\overline{C}_4$). In general, the NOR gates have fan-ins of up to $m + 1$ pulldown circuits. Each pulldown circuit consists of just one or two transistors, regardless of the size of the merge box, making for fast NOR gates and low-area pulldowns, even with minimum-sized pullups. As can be verified from Figure 3, a merge box of size $2m$ implements the following function:

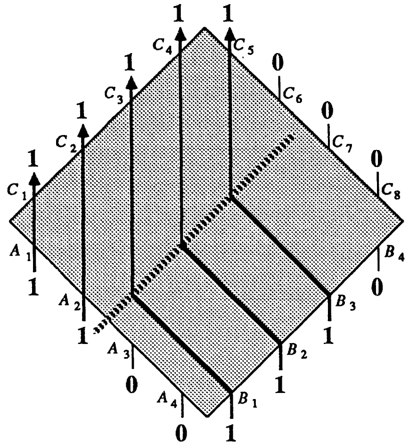


**Figure 2**: The paths taken by valid messages in a merge box. Valid bits are shown as they enter and leave the merge box. The $p$ valid messages arriving at input wires $A_1, A_2, \ldots, A_p$ are routed to output wires $C_1, C_2, \ldots, C_p$ respectively. Here, the only $A$ wires with valid messages are $A_1$ and $A_2$. These valid messages are routed to $C_1$ and $C_2$ respectively. The $q$ valid messages arriving at input wires $B_1, B_2, \ldots, B_q$ head toward $C_1, C_2, \ldots, C_q$ but are steered to $C_{p+1}, C_{p+2}, \ldots, C_{p+q}$. Here, the valid messages entering through $B_1$, $B_2$, and $B_3$ are steered to output wires $C_3$, $C_4$, and $C_5$ respectively.

$$C_i = \begin{cases} A_i \vee \left( \bigvee_{j=1}^{i} (B_j \wedge S_{i+1-j}) \right) & \text{if } 1 \leq i \leq m \\ \bigvee_{j=1}^{2m+1-i} (B_{m+1-j} \wedge S_{i+j-m}) & \text{if } m < i \leq 2m \end{cases}$$

The *switch settings* $S_1, S_2, S_3, S_4, S_5$ are computed and stored in registers during setup, based on the valid bits appearing at the $A$ and $B$ input wires. In general, a merge box of size $2m$ has switch settings $S_1, S_2, \ldots, S_{m+1}$. These stored settings continue to be used during subsequent cycles. These switch settings establish the electrical connections throughout the entire hyperconcentrator switch. Other than the storing of the switch settings, the operation of the merge box is purely combinational.

Let us look at the operation of the merge box during setup. The lower-numbered $A$ and $B$ input wires have valid bit values of 1, and the higher-numbered $A$ and $B$ input wires have valid bit values of 0. If

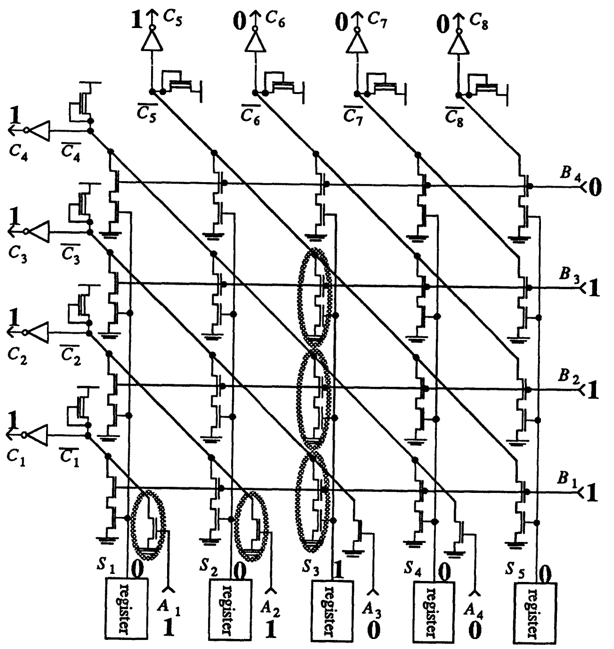


**Figure 3:** A merge box of size 8. The input wires are $A_1, A_2, A_3, A_4$ and $B_1, B_2, B_3, B_4$. The output wires are $C_1, C_2, \ldots, C_8$. The switch settings are stored during setup in registers $S_1, S_2, S_3, S_4, S_5$. Here we have $p = 2$ and $q = 3$ during setup. The valid bit values on each $A$, $B$, and $C$ wire are shown, as are the $S$ switch settings. All conducting paths to ground are circled.

input $A_i$ is 1, then the NOR gate output $\overline{C}_i$ is pulled down by the single transistor whose gate is $A_i$. The inverter causes output $C_i$ to be 1. Having input values $A_1, A_2, \ldots, A_p = 1$ thus causes the outputs $C_1, C_2, \ldots, C_p$ to be 1.

The switch settings $S$ act as steering signals, sending the $B$ values $B_1, B_2, \ldots, B_m$ to the output wires $C_{p+1}, C_{p+2}, \ldots, C_{p+m}$. The $S$ values are computed and stored during setup so that only the setting $S_{p+1}$ is 1, corresponding to input $A_{p+1}$ being the lowest-numbered $A$ with a valid bit of 0. (If no input wire $A_i$ is 0, then we have $p = m$, and only switch $S_{n+1}$ is set to 1.) The $S$ values are defined by the valid bits on the $A$ wires as follows:

$$\begin{aligned} S_1 &= \overline{A}_1 \\ S_i &= A_{i-1} \wedge \overline{A}_i \quad \text{for } 1 < i \leq m \\ S_{m+1} &= A_m \end{aligned}$$

Of the two-transistor pulldown circuits, only column $p+1$ may possibly pull a diagonal wire down to 0, since only switch setting $S_{p+1}$ is high. Similarly, a diagonal wire $\overline{C}_i$ may be pulled down only by input wire $A_i$ or the conjunction $B_{i-p} \wedge S_{p+1}$. The only NOR gate which may be pulled down by input $B_1$ has output wire $\overline{C}_{p+1}$, and in general the only NOR gate which may be pulled down by input $B_i$ has output wire $\overline{C}_{p+i}$.

For example, suppose that, as in Figure 3, the input wires have the following valid bits during setup:

$$\begin{aligned} A_1, A_2 &= 1 \\ A_3, A_4 &= 0 \\ B_1, B_2, B_3 &= 1 \\ B_4 &= 0 \end{aligned}$$

Then we have $p = 2$, $q = 3$, $S_3 = 1$, and all other $S_i = 0$. There are five valid messages passing through the merge box, and there are exactly five conducting paths to ground, circled in Figure 3, one for each of the first five diagonal wires, $\overline{C}_1, \overline{C}_2, \overline{C}_3, \overline{C}_4, \overline{C}_5$. These paths to ground cause output values of 1 on the corresponding output wires $C_1, C_2, C_3, C_4, C_5$. The remaining three diagonal wires, $\overline{C}_6, \overline{C}_7, \overline{C}_8$, are not pulled down to ground by these input values, and the output wires $C_6, C_7, C_8$ all have the value 0.

Now we look at the message bits that arrive after setup. The switch settings $S$ were computed and stored during setup, and they remain unchanged in their registers. Just as during setup, a bit with the value 1 which enters through input wire $A_i$ directly pulls down the diagonal wire $\overline{C}_i$, regardless of the $S$ values. A bit with the value 1 which enters through input wire $B_i$ may pull down only the diagonal wire $\overline{C}_{p+i}$ because the only switch setting which is 1 is $S_{p+1}$. The only difference in the merge box operation between setup and later cycles is that the $S$ values, which are always used to steer the $B$ values to the appropriate $C$ outputs, are computed and stored only during setup. In the cycles following setup, the merge box is a combinational circuit, reading the registers holding the switch settings $S$.

Recall that in Section 2 we required that all bits in an invalid message must be 0. We now can see the reason for this restriction. Suppose that in our above example, in which we had $A_3 = 0$ and $S_3 = 1$ during setup, we had that at some cycle following setup $A_3 = 1$ and $B_1 = 0$. We would expect $C_3$ to be 0 in this case, since $B_1$, which is routed to $C_3$, is 0. Since $A_3$ is 1, however, $\overline{C}_3$ is pulled down, and $C_3$ becomes 1. The requirement that $A_3$ and $A_4$ be 0 after setup eliminates spurious pulldowns.

## 4 The Hyperconcentrator Switch

In this section, we give the recursive construction for assembling merge boxes into a hyperconcentrator switch. We also show that a signal incurs exactly $2\lceil \log_2 n \rceil$ gate delays through the switch.

A hyperconcentrator switch with input wires $X_1, X_2, \ldots, X_n$, of which $k$ contain messages, and output wires $Y_1, Y_2, \ldots, Y_n$ routes the $k$ valid messages to the first $k$ output wires $Y_1, Y_2, \ldots, Y_k$. Since valid messages are identified by a valid bit of 1 during setup, a hyperconcentrator switch may be viewed as a network that sorts 1's and 0's, with 1's before 0's in the output. The switch is set during setup, with subsequent bits following these established electrical paths.

We use recursive merging to sort the messages, solving the subproblems at each level of the recursion in parallel. By knowing in advance the size of the problem, we know in advance exactly how sets will be divided and merged. We can thus build the division process into the hardware and successively merge larger sets of bits through cascades of parallel merge boxes.

Figure 4 shows the organization of a 16-by-16 hyperconcentrator switch. There are four stages through which the bits cascade, from bottom to top in the figure. By this construction, an $n$-by-$n$ hyperconcentrator switch, composed of $\lceil \log_2 n \rceil$ stages of combinational merge boxes, is itself a combinational circuit. Signals incur exactly $2\lceil \log_2 n \rceil$ gate delays in an $n$-by-$n$ hyperconcentrator switch. During setup, the $S$ switches in each merge box are computed and stored in registers. These switches establish electrical paths for messages in each merge box. Since there are no other switches between merge boxes, the $S$ switches actually establish the paths through the entire hyperconcentrator switch. As in the individual merge boxes, message bits that enter after the valid bit follow the established paths through the hyperconcentrator switch.

Figure 1 shows the layout of a 32-by-32 hyperconcentrator switch, using 4$\mu$m nMOS MOSIS design rules. In order to provide enough drive for the pulldown transistors of the next stage, the inverters following the NOR gates in each merge box are actually inverting superbuffers. Timing simulations have shown that the propagation delay through this circuit is under 70 nanoseconds in the worst case.

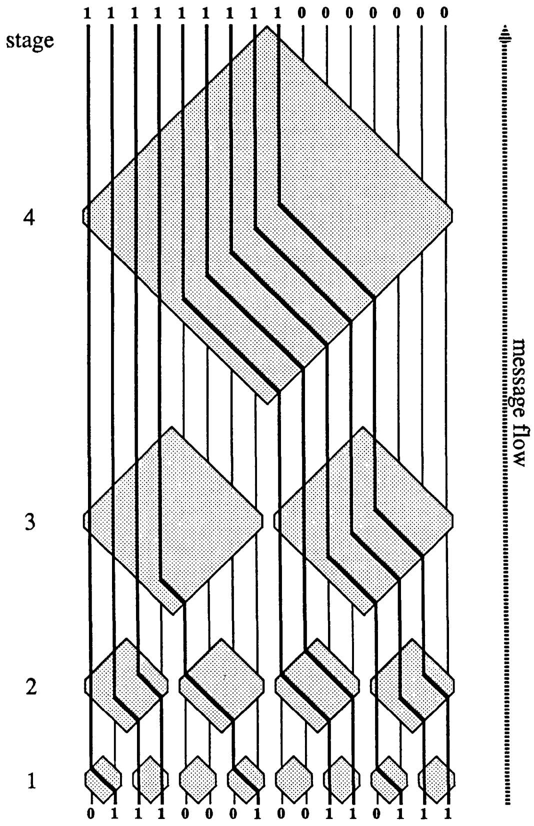


**Figure 4**: A 16-by-16 hyperconcentrator switch with four stages of merge boxes. Individual merge boxes are oriented as in Figure 2, with input wires $A$ entering the bottom left, input wires $B$ entering the bottom right, and output wires $C$ leaving the top left and top right. Messages flow from bottom to top. The output wires $C_1, C_2, \ldots, C_m$ of a merge box of size $m$ are the input wires of a merge box of size $2m$, either $A_1, A_2, \ldots, A_m$ or $B_1, B_2, \ldots, B_m$. Valid bit values are shown entering the first stage and leaving the last stage of the cascade. The electrical paths established within the merge boxes and the switch during setup are shown in heavy lines.

## 5 Applications

This section discusses two applications of the hyperconcentrator switch. One application, the one for which the switch was designed, allows us to use the available clock period more efficiently in bit-serial routing networks. Another application is its use in building a superconcentrator switch.

We can replace small, simple switches in a bit-serial routing network by concentrator switches to successfully route more messages in a single clock cycle, thus using the available clock period more efficiently. The routing network switches we shall consider route valid messages either left or right, based on an address bit immediately following the valid bit. Such a routing scheme is used, for example, in a butterfly network. An address bit of 0 indicates that the valid message should be routed to a left output of a switch, and an address bit of 1 indicates that the valid message should be routed to the right.

Consider the 2-input, 2-output butterfly node shown in Figure 5. A single level of a routing net-

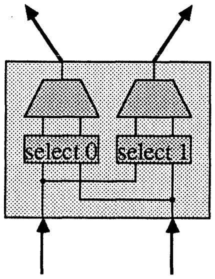


**Figure 5**: A 2-input, 2-output butterfly node. The selectors and 2-by-1 concentrator switches ensure that both input messages reach output wires if their address bits specify that they are going in different directions. If the messages contend for the same output wire, the concentrator switches ensure that one makes it through. With randomly chosen address bits, we expect $3n/4$ of the $n$ messages to be successfully routed through this node.

work such as a butterfly would typically have several such nodes side-by-side. The node contains two simple 2-by-1 concentrator switches, depicted as trapezoids, one with outputs going left and one with outputs going right. Each simple concentrator switch is preceded by a selector circuit that, given an input valid bit and an address bit, produces a new valid bit which is 1 if and only if the input valid bit is 1 and the address bit matches the output direction of the concentrator switch. If two valid messages with equal address bits enter a butterfly node, only one is successfully routed.

The problem with this scheme is that it does not use the available clock period efficiently. The simple node uses only a few levels of logic, so the delay through it is only a few nanoseconds. But because of the time required to get signals on and off chips in current technologies, we cannot distribute a clock with a frequency high enough to match the short delay of this node. In fact, the clock period we can distribute is at least an order of magnitude greater than the delay through this node. This node therefore performs no useful work in at least 90 percent of each clock cycle.

Now consider the generalized $n$-input, $n$-output butterfly node shown for $n = 8$ in Figure 6. Like four simple butterfly nodes of Figure 5 laid side-by-side, it has a total of 8 input wires and 8 output wires, with 4 outputs going left and 4 outputs going right. But here we use two $n$-by-$n/2$ concentrator switches, one with outputs only going left and one with outputs only going right.

The advantage held by the larger node is that at the same clock speed as the simple nodes, it can successfully route more valid messages in each clock cycle. The clock speed remains the same because the additional delay introduced by the larger concentrator switches is just soaked up by the unused portion of the clock period. These nodes use a larger portion of the available clock period. Since the simple nodes leave so much of the available clock period unused, we can even scale these concentrator switches up considerably before the delay introduced exceeds the original clock period. To see that more valid messages are routed by the larger node, assume that a valid message arrives at each input wire of each switch and that the address bit is 0 with probability 1/2, independent of the address bits of other messages. A simple calculation shows that we can expect the simple nodes to successfully route $3n/4$ of the $n$ arriving messages. A more complex calculation [2] shows that the expected number of valid messages successfully routed by the larger nodes with $n$ input wires is $n - O(\sqrt{n})$. Intuitively, the larger nodes successfully route more valid messages because they have more freedom in mapping inputs to outputs.

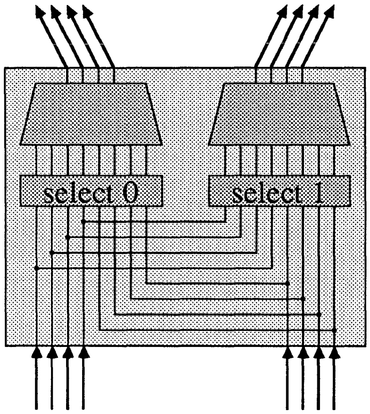


**Figure 6**: A generalized butterfly node with $n$ inputs and $n$ outputs, shown here for $n = 8$. There are two $n$-by-$n/2$ concentrator switches. With randomly chosen address bits, we expect $n - O(\sqrt{n})$ messages to be successfully routed through this node.

The approach of replacing many small routing nodes by fewer nodes with larger concentrator switches is used by the cross-omega network [5]. Part of the cross-omega network is based on a truncated butterfly network. Single wires of the butterfly network are replaced by bundles of 32 wires, and the simple butterfly network nodes are replaced by nodes like that of Figure 6, but with 32 inputs, 32 outputs, and two 32-by-16 concentrator switches.

Fat-trees serve as an example of a class of routing networks that makes use of concentrator switches whose pin counts increase with the network size. The interested reader is referred to [4] and [7] for details.

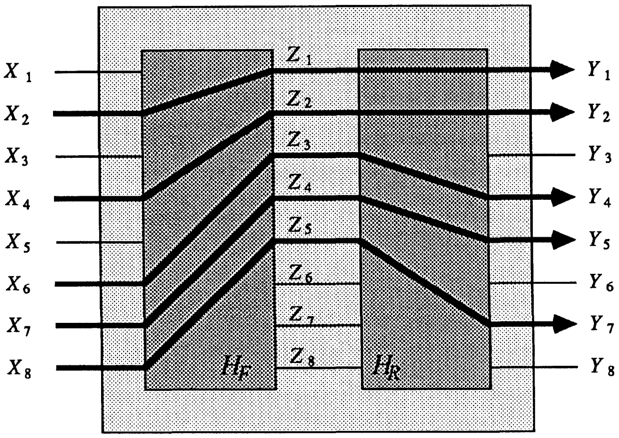


**Figure 7**: A superconcentrator switch built out of two hyperconcentrator switches. The hyperconcentrator switch $H_R$ is set up the connect the first $l$ reverse input wires $Z_1, Z_2, \ldots, Z_l$ to the $l$ good reverse output wires, which also serve as the output wires of the superconcentrator switch. The $k$ valid messages are then routed by the hyperconcentrator switch $H_F$ to the wires $Z_1, Z_2, \ldots, Z_k$ and through the switch $H_R$ to the first $k$ good output wires.

Another application of hyperconcentrator switches is in building superconcentrator switches. An *n-by-n superconcentrator switch* has $n$ input wires $X_1, X_2, \ldots, X_n$ and $n$ output wires $Y_1, Y_2, \ldots, Y_n$. For any $1 \leq k \leq n$, disjoint electrical paths may be established from any set of $k$ input wires to any arbitrarily chosen set of $k$ output wires. Superconcentrator switches are useful in fault-tolerant systems. If some of the output wires of a concentrator switch may be faulty, we can use a superconcentrator switch that routes signals to only the good output wires.

We can build a superconcentrator switch out of two full-duplex hyperconcentrator switches $H_F$ and $H_R$, as shown in Figure 7.[3] (After setup in a full-duplex hyperconcentrator switch, signals can travel along the established paths simultaneously in both forward and reverse directions. Extending the design of the hyperconcentrator switch to make it full-duplex is straightforward.) The output wires of the switch $H_F$ (a "forward" hyperconcentrator switch) feed directly into the reverse input wires of the full-duplex hyperconcentrator switch $H_R$ (a "reverse" hyperconcentrator switch). Suppose there are $l$ good output wires of the superconcentrator switch. Before setup of the superconcentrator switch, the switch $H_R$ sets up electrical paths from its first $l$ reverse input wires $Z_1, Z_2, \ldots, Z_l$ to the $l$ good reverse output wires. These paths are established by assigning a 1 to each forward input wire of the switch $H_R$ that corresponds to a good output wire, assigning a 0 to the forward input wires corresponding to faulty output wires, and running a setup cycle of the switch $H_R$.

Setup of the superconcentrator switch is then just setup of the hyperconcentrator switch $H_F$. The $k$ valid messages are routed through the switch $H_F$ to the wires $Z_1, Z_2, \ldots, Z_k$ and then along the reverse paths through the switch $H_R$ to the first $k$ good output wires.

## 6 Concluding Remarks

In this section, we describe a circuit containing the hyperconcentrator switch which has been fabricated. We also discuss how the switch can be implemented in domino CMOS, and how the switch can be used to build larger concentrators. Finally, we pose some open questions.

We have implemented a 4$\mu$m nMOS 16-by-16 hyperconcentrator switch, which was fabricated by MOSIS. The chip contains programmable selector circuitry preceding the hyperconcentrator switch so that an independent routing decision can be made for each input, as in Figures 5 and 6. Each of the 16 selectors includes a UV write-enabled PROM cell, described in [3]. The bit value stored in each PROM cell is compared with an address bit in the input message to determine whether the message is going in the correct direction. The device is currently under test.

The hyperconcentrator switch can be implemented using precharged design methodologies, such as domino CMOS, instead of ratioed nMOS technology. Although the output inverters used in domino CMOS design are already present in the merge box circuit, the conversion from ratioed nMOS to domino CMOS design is not as straightforward as it may seem at first glance. The functions which define the switch settings $S$ are not monotonic increasing, a violation of domino CMOS design principles. This problem occurs only during setup.

One solution involves an alternative computation of the $S$ values during setup, as follows:

$$\begin{aligned} S_1 &= 1 \\ S_i &= A_{i-1} \quad \text{for } 2 \leq i \leq m+1 \end{aligned}$$

It can be verified that these $S$ values produce the correct valid bit values at the merge box output wires during setup. These $S$ values, however, are not the switch settings stored in the registers. To ensure that subsequent bits are correctly routed through the merge box, we compute the switch settings as in the

[3] This construction is shown in [11].

ratioed nMOS design and store them in the registers during setup, without connecting them to any pulldown circuits. After setup we connect the switch settings to the pulldowns, as in the ratioed nMOS case, and the routing of subsequent bits takes place just as in the ratioed nMOS circuit.

The hyperconcentrator switch can be used as a building block in large concentrators. For example, replacing the comparators in an arbitrary sorting network by $n$-by-$n$ hyperconcentrator switches yields a large hyperconcentrator. (Actually, only the first level of comparators must be replaced by hyperconcentrator switches; merge boxes suffice at all subsequent levels.) We have also found that efficient partial concentrator switches can be built from hyperconcentrator chips. An $(n, m, \alpha)$ *partial concentrator switch* has $n$ inputs, $m$ outputs, and a fraction $\alpha$ such that any set of $k \leq \alpha m$ valid messages is successfully routed to output wires. Using $3\sqrt{n}$ hyperconcentrator chips, each with $\sqrt{n}$ inputs, we can build an $(n, m, \alpha)$ partial concentrator switch with $\alpha = 1 - O(n^{3/4}/m)$ [2].

It is natural to wonder whether a simple design for a concentrator switch exists when we relax the constraint that all the valid messages arrive at the same time. A crossbar switch has the capability of allowing valid messages to come and go at any time, but switch setup can be expensive. It may be that a concentrator switch can be designed that allows new messages to be routed in batches while preserving old connections.

## Acknowledgements

Thanks to Eric Tenenbaum and Forrest Thiessen of M.I.T. for their help in the nMOS implementation of the hyperconcentrator switch and to Chris Terman of M.I.T. for his help in performing the timing analysis.

## References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ parallel steps," *Combinatorica*, Vol. 3, No. 1, (1983), pp. 1–19.

[2] T. H. Cormen, "Concentrator switches for routing messages in parallel computers," Masters thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., (1986), 66pp.

[3] L. A. Glasser, "A UV write-enabled PROM," *Proceedings, 1985 Chapel Hill Conference on VLSI*, (May 1985), pp. 61–65.

[4] R. I. Greenberg and C. E. Leiserson, "Randomized routing on fat-trees," *26th Annual IEEE Symposium on Foundations of Computer Science*, (Oct. 1985), pp. 241–249.

[5] T. F. Knight, unpublished manuscript.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. 3*, Addison-Wesley, (1973), 723 pp.

[7] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, Vol. C-34, No. 10, (Oct. 1985), pp. 892–901.

[8] G. M. Masson, "Binomial switching networks for concentration and distribution," *IEEE Transactions on Communications*, Vol. COM-25, No. 9, (Sept. 1977), pp. 873–883.

[9] M. S. Pinsker, "On the complexity of a concentrator," *Proceedings 7th International Teletraffic Conference*, Stockholm, (1973), pp. 318/1–318/4.

[10] N. Pippenger, "Superconcentrators," *SIAM Journal on Computing*, Vol. 6, No. 2, (June 1977), pp. 298–304.

[11] L. G. Valiant, "Graph-theoretic properties in computational complexity," *JCSS*, Vol. 13, No. 3, (Dec. 1976), pp. 278–285.

# HARDWARE EXTRACTION OF LOW-LEVEL CONCURRENCY FROM SERIAL INSTRUCTION STREAMS[a]

by

Augustus K. Uht[b]
and
Robert G. Wedig[c]

Department of Electrical and Computer Engineering
Carnegie-Mellon University
Pittsburgh, PA. 15213

## Abstract

*A hardware solution to low-level (semantic) concurrency extraction is considered. The focus is on the reduction of data-flow inhibitors of concurrency in sequential instruction streams. The reduced procedural dependency techniques of prior work are combined with new high-speed reduced data dependency techniques to yield a new machine model executing standard sequential code in a data-flow-like manner. Also considered is a conceptually simple form of branch prediction. The model is simulated both with and without the branch prediction technique, using a wide range of benchmarks as input. The results are presented and analyzed, showing the model's functionality and performance improvement.*

## 1 Introduction

Computer system performance may be improved via application of *concurrent* or *parallel* processing techniques at many levels. This paper explores the use of *low-level* or *semantic* concurrency extraction techniques applied at the implementation stratum of processor design; the concurrency is extracted directly from the machine code, with no software pre-processing. It is assumed that the machine code is a sequential instruction stream, consisting solely of branches and assignment statements[d], in which each instruction takes one cycle to execute. The specific problem addressed by this work is increasing the concurrency extractable from sequential instruction streams, thereby improving performance, while keeping the hardware extraction techniques used both high-speed and efficient.

Low-level concurrent operation of a computer allows machine-level instructions to execute independently when the semantic restrictions between them allow; e.g., two assignment instructions may execute concurrently if they share no operand sources and sinks. Using low-level concurrency allows multiple instructions to be issued at once, typically more than one per execution cycle. Low-level concurrency was first exploited in the CDC 6600 [10] and the IBM 360/91 [1, 12], although neither machine issued more than one instruction per cycle. Also, neither computer employed significant techniques to reduce the ill effects of branches. Since these machines were built, some research progress has been made by Tjaden [11] in the quantity of low-level concurrency extractable. More recently, with the advent of lower-cost hardware, the utilization of low-level concurrency is being re-examined [4, 5, 7, 13, 16], resulting in new concurrency extraction techniques. In [15] Uht examines improved concurrency extraction techniques in which the negative effects of branches are minimized (this did not include branch prediction methods). In this paper the techniques are extended to include both reduced assignment statement constraints and a form of branch prediction. The prediction technique requires no backtracking or state restoration upon discovery that a branch target prediction was wrong.

The initial applications for this work are both super- and mainframe-(large general purpose) computers, aiding in the speedup of both scientific and general-purpose instruction streams. Future applications are widespread. With the continuing decline of hardware costs, such techniques may be realizable in single-chip machines.

In the next section the problem is further described and refined, previous work is reviewed, and the approach of the solution is outlined. In Section 3 details of the new approach are given. In Section 4 both simulation and hardware estimate data are presented and analyzed. Lastly, a summary is given in Section 5, overviewing the achievements of this work.

## 2 Previous Work and New Approach Outline

### 2.1 Dependency Types

Concurrency extraction starts as *dependency* detection. Two instructions are dependent if their execution must be ordered, due to either *semantic* or *resource* dependencies.

A semantic dependency exists between two instructions if their execution must be serialized to ensure correct operation of the code. This type of dependency arises due to ordering relationships occurring in the code itself. There are two forms of semantic dependencies, *data* and *procedural*.

### 2.1.1 Data Dependencies

Data dependencies arise due to instructions sharing source and sink names in certain combinations. Examples of data dependencies are shown in Figure 1. Referring to Relation 1, a data dependency exists between instructions 1 and 2 since instruction 1 modifies **A**, an input to (source of) instruction 2. Therefore instruction 2 must not execute in a given iteration until instruction 1 has executed in that iteration.

Examples of the three possible types of data dependency relations [2] are shown in Figure 1. Under normal circumstances (when only one copy of a variable exists in the machine at any given time) any one of the relations existing implies that a data dependency exists between the two instructions. But, as demonstrated by Tjaden [11] and Wedig [16], if multiple copies of a variable exist, then Relations 2 and 3 (collectively known as *shadow effects*) do not apply. As an example, consider the Relation 2 situation in Figure 1. Say two copies of A exist, then instruction 1 sources from the first copy and instruction 2 writes (sinks) into the second copy; therefore instructions 1 and 2 may execute concurrently. It is desired to have a machine execute such that shadow effects do not inhibit execution, consequently multiple copies of variables should be provided.

```
     Relation 1:      Relation 2:      Relation 3:
     -----------      -----------      -----------
 1.   A = B + 1        C = A * 2        A = B + 1
 2.   C = A * 2        A = B + 1        A = C * 2

  (in all of the examples, the common use of
   the variable A creates the data dependency)
```

**Figure 1:** Examples of the data dependency relations.

[a] This work was supported in part by the Semiconductor Research Corporation under Agreement No. 82-11-007, and in part by A. K. Uht.

[b] Dr. Uht is currently with the Department of Electrical Engineering and Computer Sciences at the University of California, San Diego.

[c] Dr. Wedig is currently an independent consultant. His address is: 580 College Ave., Palo Alto, California 94306.

[d] Procedural instructions; e.g., CALL and RETURN; are special cases of branches, and are included in the models described herein. An overlapped memory-mapped register set is used by the calling mechanism.

### 2.1.2 Procedural Dependencies

Procedural dependencies arise as the result of the presence of branches in the input code. Any type of instruction may be procedurally dependent on one or more branches, but only on branches, and they must be previously-occurring branches, since procedural dependencies are unidirectional (not reflexive) [11, 14]. For example, an instruction which is between a forward branch and its target is procedurally dependent on the forward branch, since its execution is conditional upon how the branch executes. An instruction after the target is not procedurally dependent on the forward branch, since it is guaranteed to execute regardless of how the branch executes, true or false [11]. The type of procedural dependency mentioned above is one of several types, descriptions of which are in [14].

For emphasis, we re-iterate: the union of data and procedural dependencies in code forms the set of semantic dependencies existing in the code, and exists independently of any and all hardware the code might run on.

### 2.1.3 Resource Dependencies

Resource dependencies exist solely due to hardware restrictions of a particular machine's realization. For example, if in a given cycle, three add instructions are eligible for execution, and only two adders are available, then a resource dependency exists. In this work, unless otherwise noted, resource dependencies are assumed not to exist.

### 2.1.4 Related Work

Riseman and Foster [9], and Fisher [4] have shown that much concurrency is extractable if the procedural dependencies can be reduced. Tjaden made the first attempt to reduce procedural dependencies, allowing more concurrency. In [14, 15] procedural dependencies are further reduced, using a *domain* model of control flow. This model, along with the relevant hardware structures appearing in the machine paradigm of [15] (called CONDEL-1), is used in the implementation described herein (called CONDEL-2). (CONDEL stands for CONcurrent Directly Executed Language machine, the emphasis being on CONcurrent.)

The data dependency models used traditionally enforce all three of the data dependency relations enumerated in Section 2.1.1. This has been necessary due to either a lack of duplicate sinks or the use of restrictive algorithms. Less constrained techniques [11, 16] result in either the partial or complete elimination of shadow effects, but suffer from unpleasant implementation features. Specifically, the algorithms for instruction execution are essentially sequential, requiring many steps per cycle, thereby negating any performance gain resulting from concurrency extraction; this is particularly true (many steps needed) for the execution of branches. The prior techniques also only allow one iteration of an instruction to execute per cycle, are potentially costly, and, more importantly, are incompatible with the reduced procedural dependency techniques of CONDEL-1.

Classical machines frequently seek to increase performance via *branch prediction*, in which branches are assumed to execute in one direction before their conditionals have been fully evaluated [6]. This decreases pipeline breaks, improving performance. Another similar technique is to conditionally execute code along both control flow paths from a branch, eliminating the wrong branch when the branch conditional has been fully evaluated. Two problems with these techniques are: rarely can more than one branch prediction be active at any given time, and some state restoration (backtracking) must take place upon a bad prediction. To date, no known branch prediction scheme solving these problems has been implemented, particularly in low-level concurrent machines. In a form of CONDEL-2 a type of branch prediction is allowed which does not require state restoration, and allows multiple branches to execute ahead of time. It is described in Section 3.5.

Other work in the area of hardware low-level concurrency extraction is being done by Torng [13], Patt's group [5], and Cocke [3].

### 2.2 Instruction Streams

There are two possible ways a CPU's execution unit can examine a program's instruction stream during the program's execution, dynamically or statically. The *dynamic instruction stream* is the code as seen in an order determined by the run-time control flow of the code, i.e., as indicated by the contents of a *program counter* (PC) in a traditional machine. The *static instruction stream* is the code as stored in memory, which is normally just the source code order of the program (see Figure 6). Tjaden's and Wedig's [11, 16] schemes both examine the static instruction stream.

Executing the static representation of a program has two major advantages over executing its dynamic representation. If an entire loop fits in CPU storage, then memory accesses due to instruction fetches decrease dramatically. Also, both branches and other types of instructions can execute simultaneously without changing the program's results, reducing the effects of procedural dependencies. The static instruction stream is used in this work.

### 2.3 Problem Definition - Narrowed Version

What is needed is a hardware algorithm for semantic concurrency extraction that exploits a maximal amount of concurrency (not including branch prediction). The model must be high speed, in that the total critical path gate delays must remain low so as not to negate the effect of increased concurrency; the algorithm should also exhibit reasonable cost. It is also desired that some form of branch prediction be developed that does not require backtracking. These requirements specifically indicate a need for efficient representations of instruction execution state and semantic dependencies (partially developed in [11, 15, 16]), and new parallel, high-speed, and affordable techniques to perform the basic concurrency calculations.

### 2.4 Previously Developed Elements, Used in the New Model

The requirements for efficient dependency and state representations are satisfied by modifications of structures proposed by Tjaden and Wedig, as developed by Uht in [14, 15].

The original data dependency matrix, used in CONDEL-1, is upper right triangular, indicating all possible data dependencies between two instructions in a single element. Examples of the new data dependency matrices are shown in Figure 7, with their associated code in Figure 6, indicating the data dependencies between two instructions separately. The matrices are composed of binary elements, and are $n$X$n$ square. A 1 in a matrix indicates a data dependency between the instructions corresponding to the row and column indices, e.g., matrix element $DD^1_{1,2} = 1$ indicates a Relation 1 data dependency between instructions 1 and 2. Upper-right triangular versions of the matrices in Figure 7 exist for procedural dependencies [15]. Note that the static instruction stream model is used. The dependencies are calculated at run-time, as described in [14]; this eliminates extra storage and memory bandwidth that would be necessitated with a compile-time calculation.

Wedig [16] also uses the static instruction stream model. He employs an *Instruction Queue* (see Figure 2) to hold a portion of the static instruction stream. Instructions enter at the bottom and are shifted up, into lower-numbered Instruction Queue rows, as the upper instructions finish executing. Any necessary decoding is performed *relatively statically* [14], one instruction at a time, as an instruction enters the Queue. Each row of the Instruction Queue holds the code data corresponding to instruction *i*, including instruction *i*'s opcode and operand identifiers, as well as a jump destination address if the instruction is a branch. The instructions in the Instruction Queue are accessed in parallel.

Wedig [16] also proposes the *Advanced Execution (AE)* matrix to hold the dynamic instruction state. Each row of the matrix ($AE_{i,*}$) contains the execution state of instruction *i* for a subset of its total number of iterations. The nominal execution order of the instructions corresponding to the Advanced Execution matrix rows is shown in Figure 3. This enforces the basic sequentiality of the serial input code when necessary (when dependencies are present). $AE_{i,j}$ is set either upon an actual execution of instruction *i* in iteration *j*, or upon execution of one or more branches in the Instruction Queue. Setting Advance Execution elements as the result of branch execution is called *virtually executing* the corresponding instructions. A register called the *b-element* stores an integer indicating the total number of iterations that each instruction in the Instruction Queue is to execute (really or virtually). The b-element is incremented when a backward branch executes true (enabling a new iteration for execution), and is decremented when an entire iteration has been executed.

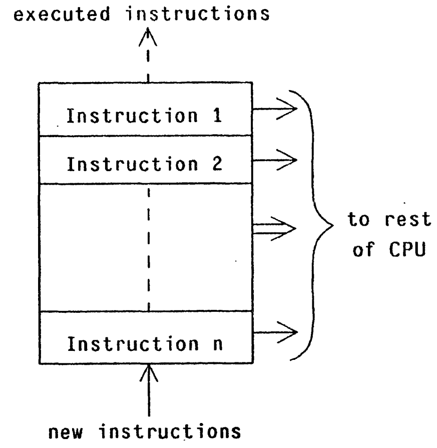


**Figure 2:** Basic Instruction Queue.

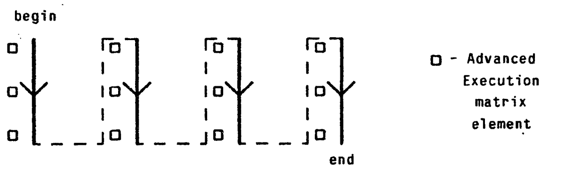


**Figure 3:** Nominal instruction iteration execution order in AE matrix.

An example of code execution with the AE matrix is shown in Figure 4. In the figure, $AE_{1,2}$, $AE_{2,1}$, and $AE_{5,1}$ are set to indicate actual or *real* execution of the corresponding instructions, i.e., an assignment actually occurred or a branch was actually taken. The instructions are all issued in the same cycle since there are no unresolved dependencies between them. Since b = 2, instruction 1 is fully executed. $AE_{3,1}$ and $AE_{4,1}$ are set as a result of the forward branch (instruction 2) executing true (the branch is taken) in iteration 1. Since instructions 3 and 4 are in the domain[e] of the forward branch, they must be kept from really executing, and thus are virtually executed in their first iterations, i.e., their AE elements are set, but they are <u>not</u> issued for execution (no real execution takes place). Therefore, virtual execution of an instruction in a given iteration is the same as disabling the instruction from really executing in the iteration.

```
        Code                          AE matrix   AE matrix
                                      (before 2   (after 2
                  b=2                  executes    executes
                                        true)       true)
                  iteration #:         1 2 3 4     1 2 3 4

1.      A = A + 1                      1 0 0 0     1 1 0 0
2.      IF  A > 0  GOTO  5.            0 0 0 0     1 0 0 0
3.          B = C + D                  0 0 0 0     1 0 0 0
4.          E = F + G                  0 0 0 0     1 0 0 0
5.      W = A                          0 0 0 0     1 0 0 0

    Instructions 3 and 4 are procedurally
       dependent on instruction 1.
```

**Figure 4:** Code execution and the *AE* matrix.

In Wedig's shadow effects reduction model, a matrix of registers called the *Shadow Sink (SSI)* array is used to store multiple copies of instruction sinks, or outputs. The matrix has the same dimensions as the AE matrix (*n*X*m*), and each element of SSI corresponds to the similarly located AE element; in other words, there is one sink register for each AE-represented iteration of each instruction held in the Instruction Queue.

### 2.5 Basic Execution Algorithm

During execution cycles, the dependency information is combined with the dynamic execution state held in the Advanced Execution matrix to generate a list of instructions to be executed in the current cycle. These instructions are issued for execution. This list of executing instructions and the domain information is used to update the Advanced Execution matrix, whence the cycle repeats.

### 2.6 Approach of Solution

Our approach is to develop a new machine model, CONDEL-2, that uses the best features of Tjaden's and Wedig's approaches, the reduced procedural dependency model of CONDEL-1 [15], and new ideas to solve the problem. Specifically, CONDEL-2 employs the static instruction stream model, the basic execution algorithm of [15] (see Section 2.5), and Wedig's Shadow Sink matrix. Modifications of Wedig's AE matrix [16] and the data dependency matrix [14, 15] are also used. New parallel high-speed logic is described that both performs the concurrency calculations, and links instruction outputs (sinks) to other instructions' inputs (sources). Other logic and structures are used to decouple sink storage from instruction execution, both improving performance, and allowing the implementation of a simple form of branch prediction, further improving performance.

## 3 New Solution Description

In this section the major elements of the new model CONDEL-2 are overviewed; details may be found in [14]. In [14] the Relation 1 data dependency in conjunction with a reduced set of procedural dependencies are shown to constitute a minimal set of semantic dependencies for virtually all sequential code. The reduced set of procedural dependencies was established via both exhaustive consideration of control flow types and a recursive proof. This section describes the hardware necessary to implement the minimal semantic dependencies (with restrictive array accesses, described in Section 3.4). The solution description concludes with an example of the operation of the hardware. (The reader may find it helpful to refer to Figures 3 and 6 - 8 as the hardware is introduced.)

### 3.1 Basic Elements of the Solution

The basic problem is to determine which, if any, sink element of the instructions in the IQ should be used as the source element of an instruction in the IQ (it may be the same instruction). In any case the model must allow for independent writing of multiple instruction iterations' sink operands. The SSI matrix of Wedig [16] satisfies this requirement. It is helpful to consider the model of program execution vis-a-vis the AE matrix presented by Wedig [16]; see Figure 3. (For the remainder of the discussion, one source per instruction is assumed[f]. The presentation is easily generalized to accommodate multiple sources.) The directed line shows the nominal or *serial* order of execution of the sequentially biased code in the IQ. A specific iteration of an instruction uses as a source a sink generated previously and residing in either memory or the SSI. Therefore the previous sink is somewhere *serially* previous to the specific iteration, back along the line. The particular SSI word to be used is indicated by both the data dependencies and the execution state of the relevant instructions.

It has been determined [14] that knowing just the Advanced Execution state is insufficient, as the validity of an instruction's sink cannot be determined from the AE state. What is needed is a separation of the *real* from the *virtual* execution state held in the

[e] In the CONDEL models, branch domain indicators are stored in matrices similar to those holding dependencies; see [14, 15].

[f] Adding a source requires adding a like amount of linking logic, as well as requiring an additional data dependency matrix.

AE matrix. Therefore two new matrices are used, the Real Execution (RE) and the Virtual Execution (VE) matrices, described in Section 3.2. Note that AE = RE + VE; this is a *logic* equation.

Only one *serial data dependency* (that of Relation 1) is needed for all instructions, with the exception of array accesses (see Section 3.4). This data dependency occurs when the sink of one instruction's iteration is the same as the source of a serially later instruction's iteration. The data dependency information kept in the DD matrix of CONDEL-1 is separated into multiple square matrices in CONDEL-2, one per possible source-sink pair, to allow for individual access to an instruction's Relation 1 data dependencies (those of an instruction's source on other instructions' sinks). In the new model, an instruction in a given iteration may be data dependent on the same instruction in a previous iteration, e.g., an instruction of the form: I = I + 1.

It is both conceptually and notationally convenient in many cases to look at the AE, RE, and VE matrices (or any nXm matrix) as one-dimensional vectors with length nm, with the elements column ordered as indicated by the line in Figure 3. This is then just the serial ordering of the instructions' iterations. For the remainder of this document, either serial or row and column indexing of an array may be used for the array; the default is the latter; if serial indexing is used, it will be noted.

The Virtual and Real Execution matrices and the separated Data Dependency (DD) matrices are the primary new structures needed to solve the problems. The key element tying them together is the new linking logic, described in Section 3.3.1.

### 3.2 The Hardware Matrices

With the exception of the dependency matrices previously discussed, all of the hardware matrices in CONDEL-2 are similar to the Advanced Execution matrix. There is a one-to-one correspondence between each element in the AE matrix and each element in the other matrices, each element indicating some state corresponding to a particular instruction's iteration. The matrices are of dimension nXm, as is the AE matrix. The new matrices are individually described as follows.

#### 3.2.1 Real Execution (RE) Matrix

Each element of RE is binary. $RE_{i,j} = 1$ (is set to one) iff instruction $IQ_i$ has *really* executed in iteration j. An instruction really executes if, for an assignment statement, an assignment has really occurred; or for a branch, a conditional has really been evaluated and a branch decision made.

#### 3.2.2 Virtual Execution (VE) Matrix

Each element of VE is binary. $VE_{i,j} = 1$ (is set to one) iff instruction $IQ_i$ has *virtually* executed in iteration j. An instruction virtually executes if it is to be disabled (for execution) by a branch's true execution (the branch is taken).

#### 3.2.3 Shadow SInk (SSI) Matrix

Created by Wedig [16], each element of SSI is typically the size of an architectural machine register, i.e., is large enough to hold variables' values. $SSI_{i,j}$ is loaded with the sink value (result) of an assignment instruction $IQ_i$ having executed in iteration j. Variables' values are normally held in SSI at least until they have been copied to memory. Since there are multiple locations (in SSI) to store instruction results (i.e., there are multiple copies of variables), basic shadow effects need not occur.

#### 3.2.4 Instruction Sink Address (ISA) Matrix

Each ISA element is large enough to hold addresses (e.g., main memory addresses) of variables. There is one ISA element for each SSI element. $ISA_{i,j}$ holds the sink address of the sink value held in $SSI_{i,j}$. For non-array write instructions, $ISA_{i,*} = AA_i$, where AA is operand A's address (held in the IQ). For array write instructions, ISA is determined for each iteration at run-time.

The previous model of execution (CONDEL-1) is modified so that now during every cycle each eligible SSI value is written into memory at the location pointed to by the contents of the corresponding ISA element. The determination of eligibility is discussed later.

#### 3.2.5 Advanced STorage (AST) Matrix

The AST matrix is a binary matrix with the same dimensions as the AE matrix. $AST_{i,j}$ is set to 1 if either $VE_{i,j}$ is set to 1 (instruction i receives a virtual execution in iteration j), or $SSI_{i,j}$ has been written to memory. In other words, $AST_{i,j}$ equals 1 iff it may be considered that $SSI_{i,j}$ has been stored; conceptually it is similar to the AE matrix, in that either real or virtual writes cause the matrix elements to be set.

### 3.3 Hardware Logic

#### 3.3.1 Basic Linking and Executable Independence Logic

The fundamental state storage structures necessary to solve the problem have been described. The primary logic that generates the source-to-sink links and calculates executable independence is now outlined[g]. The problem may be re-formulated to a more concrete specification as follows. It is required to have nm sets of less than nm output enabling lines each, one set per source per IQ instruction iteration, each line of which potentially enables (connects) a serially previous sink word to the instruction iteration's source input. These lines are designated

$SEN^u_{t,z}$ (SEN stands for Sink ENable);

where:

- *u* is the serial index to the IQ instruction iteration under consideration for execution;
- *t* indicates the serial SSI element under consideration for linking to an input of *u*, it is always serially previous to u;
- *z* indicates the source of instruction *i* that corresponds to a particular SEN matrix, it is a source index;

($i$ = row($u$); $j$ = col($u$)).

The SEN elements are calculated simply from the RE, VE, and DD state. An instruction iteration u can only link (take a source from) an instruction iteration t if: t is both really executed and data dependent on u, and all instruction iterations serially between u and t are virtually executed (if they are data dependent on u). For each u and z combination, at most one $SEN^u_{t,z}$ is equal to one. If none equal one, then either the source is not in SSI, i.e., it is in memory, or the source has not yet been produced. The rough location (whether or not in memory) of a source is given by the Source From Storage ($SFS_{u,z}$) indicators, u and z defined as above. The SFS values are calculated from DD and VE. A source should be taken from storage if for all serially previous iterations, no valid sink exists in Shadow SInk storage; otherwise, the source should be taken from an appropriate SSI element.

It follows that iteration j of instruction $IQ_i$ (with serial index u) is data executably independent (all data dependencies resolved) if either its source is in memory or one SEN element is set (i.e., a valid sink exists in SSI). Complete Semantic Executable Independence is determined from this and the procedurally executably independent conditions, given in [14].

#### 3.3.2 Decoupling of Sink Storage from Instruction Execution

New logic and structures have been developed to allow memory updates to be decoupled from instruction execution. The write enable lines are called Write Sink Enable, and there is one per Shadow Sink element. The system works as follows. Basically, each $WSE_u$ (Write Sink Enable for serial iteration u) equals one during a cycle iff the corresponding $SSI_u$ is to be written into memory (at location $ISA_u$). The WSE logic is also used to update the contents of the Advanced Storage matrix every cycle; if $WSE_u$ is true, then $AST_u$ is set to indicate that Shadow Sink u has been stored into memory. The combination of the WSE logic and the ISA and AST matrices allow the decoupling of instruction execution from memory updating. This in turn allows the form of branch prediction described in Section 3.5, as well as reducing the

[g] Complete versions of the logic, including the restrictive array accesses, may be found in [14].

negative effects of certain array access sequences. The detailed derivation of the WSE logic may be found in [14].

### 3.4 Less Restrictive Array Accesses

As in CONDEL-1, array accesses are restrictive in CONDEL-2, but not to the same degree. In the implementation of CONDEL-2, data dependency Relation 3 (common sink) type array accesses may be executed concurrently, due to the presence of multiple sink copies (Shadow Sinks). However, since all array reads are made from memory[h], Relation 1 and 2 type array accesses may not execute concurrently. In other words, any two array accesses involving both an array read and an array write to the same array must be sequentialized; otherwise (with only array writes or only array reads taking place) the accesses may proceed concurrently.

### 3.5 Implementation and Discussion of Branch Prediction

Classically, branch prediction techniques have been used to conditionally execute code beyond branches in the dynamic instruction stream ordering. Since such execution is conditional, a certain amount of code-backtracking or state-restoration must be made if the prediction is subsequently found to be wrong. This complicates the hardware of machines using such techniques, and can reduce performance in branch intensive situations. Also recall that with the dynamic instruction stream ordering, and considering the degree of hardware complexity needed with multiple simultaneous branch predictions, usually only one branch is conditionally executed at a given time.

The model described thus far (CONDEL-2) is strictly deterministic, in that no branch prediction is used. A modification to CONDEL-2 is now described that allows a simple form of branch prediction; this form is called *Super Advanced Execution (SAE)*. The basic rule of SAE is as follows:

> *Instructions within an innermost loop assume that the Backward Branch comprising the loop will always execute true.*

This means that inner loops will be presumed to execute an infinite number of iterations (in reality as many as there are in the AE matrix [m]). Note that forward branches within an inner loop may therefore also execute ahead of time, increasing the degree of branch prediction possible. Inner loops are flagged by the hardware at run-time, using a Backward Branch Domain matrix [14]. The way SAE works in the hardware is now described.

Referring to Figure 5, note that b = 3, and therefore normally only those instructions' iterations in columns 1-3 are allowed to execute. Indeed, they must execute for the output of the code to be correct. With SAE these instructions (indicated by X's and T's) execute as before, but in addition those instructions' iterations (indicated by S's) to the right of column 3 (to the right of the b pointer) and within the inner loop are also allowed to execute. This is possible, with slight modifications of the Sink ENable and Semantically Executably Independent logic, by considering the other instructions' iterations (indicated by V's) to be virtually executed[i]; thus, no links are made to these iterations, and the V iterations allow S iterations to link to X or T iterations. The Write Sink Enable (WSE) logic is not modified, however, so that only those sinks whose instructions' iterations are guaranteed to actually execute (i.e., those sinks to the left of and including the column pointed to by the b element) are eligible to be written into memory. This means that instructions' iterations S can execute ahead of time, writing to their corresponding sink in the Shadow Sink matrix, but the sink is not copied into memory at least until the instruction's iteration becomes an X instruction iteration[j]. Therefore no state-restoration or backtracking is needed. See [14] for implementation details of SAE.

The execution algorithm is the same as in the basic CONDEL-2 machine, with the following addendum. When an outer backward branch executes true in a given iteration, it causes all future state (SAE) to be reset. This is necessary to maintain potential dependencies.

Summarizing:

1. No state-restoration or backtracking is necessary for the branch prediction technique of Super Advanced Execution.
2. In this technique, multiple branches, both forward and backward, may execute ahead of time.

```
                AE Matrix - (b = 3)
iteration #  1   2   3   4   5   6
             X   X   X | V   V   V
             X   X   X | V   V   V
             ----------|----------
             X   X   X | S   S   S  <--
             X   X   X | S   S   S    |
             X   X   X | S   S   S    |
             X   X   X | S   S   S  --- backward branch
             ----------|----------
             X   X | T | V   V   V
             X   X | T | V   V   V

             X - regular execution
             S - super advanced execution
             V - considered to be
                   virtually executed
             T - (see footnote in text)
```

Figure 5: Super Advanced Execution and the AE matrix.

### 3.6 Example of CONDEL-2's Instruction Issuing Mechanism

A brief, simple example is now presented illustrating the concepts and logic of CONDEL-2 including Super Advanced Execution. The major logic structures are shown after several cycles of the execution of a small piece of code. The code segment used for the example is shown in Figure 6. The associated dependency matrices are displayed in Figure 7. $DD^1$ holds source 1 data dependencies, $DD^2$ holds source 2 data dependencies, and $DD^3$ holds the Relation 3 data dependencies (only needed for array accesses).

Array "A" has the values:

A(1)=4     A(2)=3     A(3)=2     A(4)=1     A(5)=5

The AE matrix iterations are numbered:

| | | column #: | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| | 1 | 1 | 7 | 13 | 19 | 25 |
| | 2 | 2 | 8 | 14 | 20 | 26 |
| row #: | 3 | 3 | 9 | 15 | 21 | 27 |
| | 4 | 4 | 10 | 16 | 22 | 28 |
| | 5 | 5 | 11 | 17 | 23 | 29 |
| | 6 | 6 | 12 | 18 | 24 | 30 |

The concurrency structures' contents and logic values represent the machine state at the beginning of the cycle, with the following exceptions. The SSI values shown are those occurring after the SEI (Semantically Executably Independent) indicated instructions have executed. The SEI values shown in a cycle are those occurring just prior to the updating of AE, etc., in the same cycle. The SAEVE matrix indicates those iterations assumed to have virtually executed for Super Advanced Execution. Also note that only some of the SEN matrices are shown, due to space limitations.

The steady-state instruction execution rate (five instructions executing per cycle) is reached in cycle 5 (see Figure 8). This is an instance of *saturation*, in which every instruction within a loop

[h] This can be avoided if $O(n^2m^2)$ address comparators are provided to match array sources with array sinks, the addresses of which are not known until execute time; in this case the dependencies with previous array read instructions need not hold. The technique of CONDEL-2 uses much less hardware and is more practical; no comparators are used (for a similar execute-time function).

[i] The instructions in the T region are also considered to be virtually executed by instruction iterations in the S region. This is so that T sinks are not used as inputs to S instruction iterations. Otherwise, T instruction iterations are both viewed, and allowed to execute, as normal (X) instruction iterations.

[j] This can occur only upon the inner loop's Backward Branch actually executing true.

```
I#  Instruction          Sink   Src. 1  Src. 2
--  -----------          ----   ------  ------
1.  I=0                   I       0       -
2.      I=I+1             I       I       1
3.      B=A(I)            B       A       I
4.      C=B*2             C       B       2
5.      D(I)=C            D       I       C
6.      IF I<6 GOTO 2.    -       I       6

   Instructions 2-6 comprise the inner loop.
```

**Figure 6:** Example code segment.

$$\begin{array}{cccccc} \multicolumn{6}{c}{DD^1} \\ 0&1&0&0&1&1 \\ 0&1&0&0&1&1 \\ 0&0&0&1&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{array} \qquad \begin{array}{cccccc} \multicolumn{6}{c}{DD^2} \\ 0&0&1&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{array} \qquad \begin{array}{cccccc} \multicolumn{6}{c}{DD^3} \\ 1&1&0&0&0&0 \\ 1&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{array}$$

(the procedural dependency
matrices contain all 0's)

**Figure 7:** Example dependency matrices.

executes in one iteration every cycle. In this cycle, the first column of the structures will be retired (the storage matrices shifted left and b decremented) since the first column of AE is all 1's. Therefore more state space is made in the storage matrices to continue to allow five instructions to execute per cycle, until the loop terminates. Also, the first SEN matrix indicates that instruction 5 should use the first iteration copy of instruction 2's sink as its source 1. Note that the execution of instruction 2 in iteration 4 is an example of Super Advanced Execution.

## 4 Evaluation Data and its Analysis

### 4.1 Introduction

Measurement data evaluating the new solution follows. The solution is critiqued via both simulations and hardware estimates. In the following sections the benchmarks are briefly described, the performance measurements are given with analyses, and hardware estimates are presented.

### 4.2 Benchmark Descriptions

Two sets of benchmarks were used, a General Purpose set and a Scientific set. The General Purpose benchmarks contain a relatively high degree of control flow complexity; the benchmarks perform a wide variety of tasks and are representative; **Dhrystone** [17] is included. The Scientific benchmarks exhibit a very low degree of control flow complexity, perhaps too low to be realistic. They consist of the 14 standard Lawrence Livermore Loops [8], with the number of loop iterations of each loop reduced by about a factor of 10, to reduce simulation time[k]. All of the benchmarks were hand-coded into an intermediate assembly code; unless noted otherwise, the coder mimicked a typical compiler[l].

b=3

$$\begin{array}{ccccc} \multicolumn{5}{c}{RE} \\ 1&0&0&0&0 \\ 1&1&1&0&0 \\ 1&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&0 \\ 1&1&0&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{VE} \\ 0&1&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{AE} \\ 1&1&1&0&0 \\ 1&1&1&0&0 \\ 1&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&0 \\ 1&1&0&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{SSI} \\ 0&x&x&x&x \\ 1&2&3&4&x \\ 4&3&2&x&x \\ 8&6&x&x&x \\ 8&x&x&x&x \\ x&x&x&x&x \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{AST} \\ 1&1&1&0&0 \\ 1&1&1&0&0 \\ 1&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{WSE} \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&0&0&0 \end{array}$$

$$\begin{array}{ccccc} \multicolumn{5}{c}{SEI} \\ 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 1&0&0&0&0 \\ 0&0&1&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{SAEVE} \\ 0&0&0&1&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{SEN^5_{*,1}} \\ 0&-&-&-&- \\ 1&-&-&-&- \\ 0&-&-&-&- \\ 0&-&-&-&- \\ -&-&-&-&- \\ -&-&-&-&- \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{SEN^5_{*,2}} \\ 0&-&-&-&- \\ 0&-&-&-&- \\ 0&-&-&-&- \\ 1&-&-&-&- \\ -&-&-&-&- \\ -&-&-&-&- \end{array} \quad \begin{array}{ccccc} \multicolumn{5}{c}{SEN^{18}_{*,1}} \\ 0&0&0&-&- \\ 0&0&1&-&- \\ 0&0&0&-&- \\ 0&0&0&-&- \\ 0&0&0&-&- \\ 0&0&-&-&- \end{array}$$

**Figure 8:** Example concurrency structures and logic: cycle 5.

### 4.3 Performance Evaluation - Simulation Results

#### 4.3.1 Introduction

Simulations were run both to verify functionality of the hardware algorithms and to measure their performance, particularly with respect to standard sequential execution. To these ends the execution of a variety of benchmarks was simulated in software assuming both the new hardware algorithms previously described, and, for comparison purposes, a more restrictive algorithm. The simulator mimicked the hardware structures and logic present in the various machine models. All of the simulations produced the correct results, so functionality has been verified. The main performance measurement is the speedup $S$; for a given benchmark, $S$ is defined as the number of cycles necessary to execute the benchmark purely sequentially, divided by the number of cycles necessary to execute the benchmark concurrently. For a set of benchmarks, $S$ is the average of the component benchmarks' speedups, all weighed evenly.

The machine model independent-variable is data concurrency type *dct*. The data dependency and branch prediction models implemented in the simulator are indicated by the *dct* value, defined as:

- 1 - maximally restrictive data dependencies; all data dependency relations shown in Figure 1 hold; note that this model of data dependencies, as implemented in CONDEL-1, precludes the use of Super Advanced Execution (SAE)
- 2 - reduced data dependencies without SAE; only data dependency Relation 1 holds for non-array accesses, and no branch prediction is used
- 3 - reduced data dependencies with SAE; same as 2, but using the branch prediction technique of Section 3.5

In the remainder of this section the simulation results of the machine model of Section 3, CONDEL-2, are presented for both sets of benchmarks. The very reduced procedural dependency model of [15], i.e., procedural concurrency type (*pct*) C, is assumed throughout; the results of the Scientific set are independent of *pct*. For all of the simulations, the Instruction Queue length *n* is fixed at 16 or 32.

#### 4.3.2 General Purpose Set Simulation Results

The simulation results are summarized in the plot of Figure 9. Within the plot, the three curves correspond to the different data

[k] If anything, this makes the simulation results more conservative, since better performance of the models typically occurs as the number of loop iterations increases (reducing the effects of startup execution transients).

[l] The following simple compiler features were assumed: mapping of assignment expressions to low-height binary execution trees, little re-use of registers, and HLL statement translation as described in [14].

concurrency types (*dct* values). The data corresponding to the restrictive data dependency model of CONDEL-1 (*dct* = 1) is included in the plot for comparison purposes. The range of the speedups for individual benchmarks with m = 8 is from 1.59 to 2.77 for *dct* = 2, and from 1.61 to 4.79 for *dct* = 3. In the plot, it is seen that reducing data dependencies improves performance, as does the use of Super Advanced Execution. Increasing Advanced Execution matrix width *m* improves performance, when SAE is used, as it allows more iterations to be active at any given time, and with the branch prediction technique of SAE more iterations are likely to be used. Note that no prior model implements either of the most advanced concurrency types: very reduced procedural dependencies, or reduced data dependencies with Super Advanced Execution.

### 4.3.3 Scientific Set Simulation Results

The Scientific benchmark set simulation results are shown in Figures 10 and 11. Figure 10 includes the results of all of the loops. Figure 11 includes the results for just those loops for which loop capture[m] occurred (Loops 1-6, 11, and 12). The plot of Figure 11 demonstrates the benefits of loop capture. In both plots the performance improvements of the reduced data dependency techniques are seen to be significant, particularly with Super Advanced Execution. Also note that a wider Advanced Execution matrix (larger *m*) gives greater performance in the same situations as with the General Purpose benchmark set results. The individual speedups for each of the loops ranged from 1.66 to 9.55 for *dct* = 2, and a range of 1.66 to 11.69 for *dct* = 3. From the data, it was seen that many of the benchmarks are resistant to performance enhancements. Six performance inhibiting mechanisms have been identified [14]. With the elimination of these mechanisms via relatively simple software preprocessing techniques (extent in a compiler), 9-12 of the loops should execute in saturation (see Section 3.6); this situation currently occurs with only 3 of the loops.

## 4.4 CONDEL-2 Hardware Estimates

### 4.4.1 Hardware Cost

The hardware cost of the reduced data dependency hardware is $O(Kn^2m^2)$, where K is about 7. (We assume unlimited fan-out, fan-in of 10 [except for some wired-OR connections], and 6 gates per storage element.) The main contributors to the cost are the Sink ENable and Write Sink Enable logic. Being conservative (pessimistic), and taking n = 32 and m = 8 (large enough to extract most of the concurrency of the Scientific benchmark set), the cost is about 1 Mgates. If General Purpose code is the primary application of a target machine, then n and m may be reduced (while the machine will still capture most of the concurrency), giving a vastly reduced hardware cost.

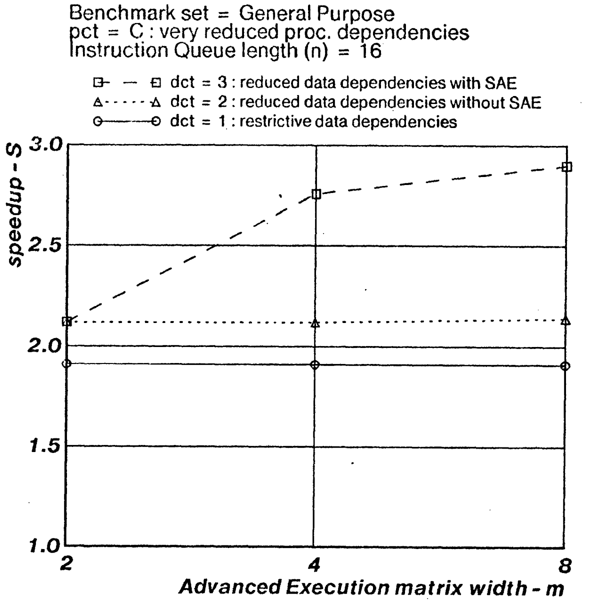


Figure 9: GP performance summary, *pct* = C.

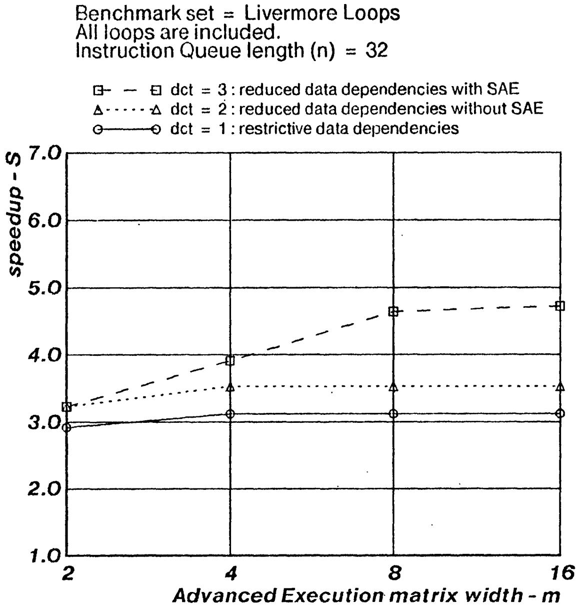


Figure 10: Scientific set performance summary, all loops included.

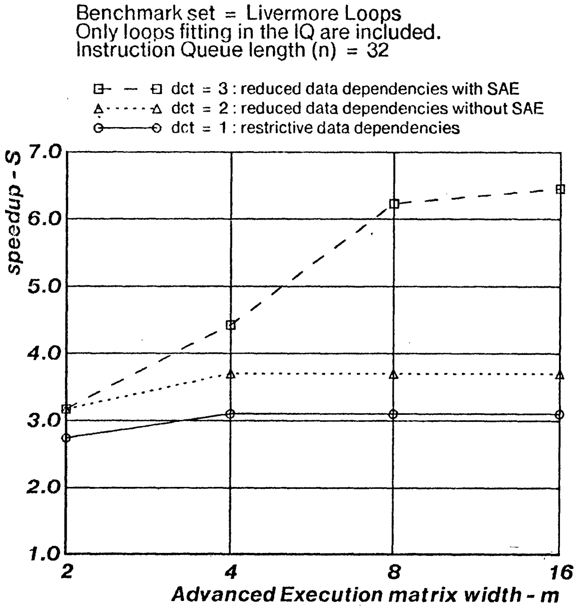


Figure 11: Scientific set performance summary, with loop capture.

The prior techniques have the following costs.

- Tjaden's model: $O(n^2R)$, R is size of reassignable storage
- Wedig's model: $O(n^2m + R\log_2 n)$, R is size of reassignable storage

[m] Loop capture occurs when the loop is completely contained in the Instruction Queue.

The costs of these models may or may not be greater than CONDEL-2, depending on the choice of parameters; however, neither model is as functionally advanced as CONDEL-2.

### 4.4.2 Hardware Delay

The hardware delay is low because CONDEL-2's logic and update algorithms are inherently parallel; limited fan-in and fan-out should not increase the delay excessively, depending on the logic family used and the choices of n and m. This compares very favorably to the prior models of Tjaden and Wedig, in which the algorithms were basically sequential, leading to extremely large delays. In [14, 15] CONDEL-1, a similar kind of machine, is estimated to have a critical path delay of 8 gate-delays; this is the same as a Cray-1, and less than any other machine examined. A preliminary analysis shows that CONDEL-2's critical path delay is about the same as CONDEL-1's, given similar amounts of hardware resources (Processing Elements, etc.).

## 5 Summary

The major concepts presented in this document are functionally sound and provide significant performance gains over standard sequential execution techniques. There are indications that performance may be further improved via software concurrency enhancement.

A significant speedup of sequential imperative code has been demonstrated through the use of low-level concurrency extraction. Reduced data dependencies enhance the concurrency detectable. Although a complete set of restricted resource measurements must yet be made on CONDEL-2, it has been found that the efficiency approaches 1 when execution saturation occurs, where efficiency is the speedup divided by the number of peak resources (Processing Elements) used. CONDEL-2 exhibits improved performance over both sequential machines and the concurrent models of Tjaden and Wedig with reduced hardware delay compared to their models; the hardware cost may or may not be comparable, depending on machine parameters. With the previously mentioned constraints, and disregarding branch prediction, CONDEL-2 extracts all semantic concurrency from an input imperative instruction stream; a parallel logic algorithm is used. A simple branch prediction technique has been implemented which requires no backtracking and allows multiple branches to execute ahead of time.

CONDEL-2, with or without Super Advanced Execution, achieves data-flow execution of typical sequential code (consisting of branches and assignment statements, and assuming restrictive array accesses) with minimal deterministic control-flow constraints.

Planned future work includes:

- increasing the degree of branch prediction implemented in the model;
- making a detailed design study of CONDEL-2, including investigating reductions in its hardware cost;
- investigating compile-time techniques to increase the concurrency extractable (as mentioned in Section 4.3.3);
- examining the performance effects of reduced resources (limited processing elements);
- and performing simulations of more benchmark programs.

## References

1. Anderson, D. W., Sparacio, F. J. and Tomasulo, R. M. "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling." *IBM Journal* (January 1967), 8-24.
2. Bernstein, A. J. "Analysis of Programs for Parallel Processing." *IEEE Transactions on Electronic Computers EC-15* (October 1966), 757-763.
3. Cocke, J. High Performance Scalar Scientific Architectures. Distinguished Lecture given at the Nineteenth Annual Hawaii International Conference on System Sciences, 1986.
4. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. Proceedings of the 10th Annual International Symposium on Computer Architecture, ACM-SIGARCH and the IEEE Computer Society, June, 1983, pp. 140-150.
5. Hwu, W., Melvin, S., Shebanow, M., Chen, C., Wei, J., and Patt, Y. An HPS Implementation of VAX; Initial Design and Analysis. Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, 1986.
6. Lee, J. K. F. and Smith, A. J. "Branch Prediction Strategies and Branch Target Buffer Design." *COMPUTER, IEEE Computer Society 17*, 1 (January 1984), 6-22.
7. Requa, J. E. and McGraw, J. R. "The Piecewise Data Flow Architecture: Architectural Concepts." *IEEE Transactions on Computers C-32*, 5 (May 1983), 425-438.
8. Riganati, J. P. and Schneck, P. B. "Supercomputing." *COMPUTER, IEEE Computer Society 17*, 10 (October 1984), 97-113.
9. Riseman, E. M. and Foster, C. C. "The Inhibition of Potential Parallelism by Conditional Jumps." *IEEE Transactions on Computers* (December 1972), 1405-1411.
10. Thorton, J. E. Parallel Operation in the Control Data 6600. Proceedings of the Fall Joint Computer Conference, AFIPS, 1964, pp. 33-40.
11. Tjaden, G. S. *Representation and Detection of Concurrency Using Ordering Matrices.* Ph.D. Th., The Johns Hopkins University, 1972.
12. Tomasulo, R. M. "An Efficient Algorithm for Expoiting Multiple Arithmetic Units." *IBM Journal* (January 1967), 25-33.
13. Torng, H. C. An Instruction Issuing Mechanism for Performance Enhancement. Tech. Rept. EE-CEG-84-1, School of Electrical Engineering, Cornell University, Ithaca, NY, February, 1984.
14. Uht, A. K. *Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams.* Ph.D. Th., Carnegie-Mellon University, December 1985.
15. Uht, A. K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, 1986.
16. Wedig, R. G. *Detection of Concurrency in Directly Executed Language Instruction Streams.* Ph.D. Th., Stanford University, June 1982.
17. Weicker, R. P. "Dhrystone: A Synthetic Systems Programming Benchmark." *Communications of the ACM 27*, 10 (October 1984), 1013-1030.

# FAULT LOCATION IN $\Omega^r\Omega^{-1}$ NETWORKS

Wei Young
Department of Computer and Information Sciences
University of Alabama
Birmingham, AL 35294

Tse-yun Feng
Department of Electrical Engineering
Pennsylvania State University
University Park, PA 16802

*Abstract* -- A fault detection and location procedure for $\Omega^r\Omega^{-1}$ networks is presented in this paper. Test vectors, with length independent of the size of the network, are first generated; then single fault is detected and various strategies, depending on the type of the fault, are used to locate it. Using other test vectors with length proportional to the size of the network is also discussed.

## 1. Introduction

In this paper, we present a fault diagnostic procedure for rearrangeable $\Omega^r\Omega^{-1}$ networks with $N(=2^n)$ input/output terminals. This procedure also applies to Benes networks which are deducible from $\Omega^r\Omega^{-1}$ networks as discussed in [1]. It is much simpler and faster than the method in [2] which involves complicated computation using the looping algorithm.

We assume individual switching elements are not accessible and use the following notations throughout this paper. An $N=2^n$ input/output $\Omega^r\Omega^{-1}$ network is described by $sE^0sE^1...sE^{n-1}uE^n...uE^{2n-2}$, where s and u are the shuffle and the unshuffle respectively and $(E^i)$'s are switching element stages indexed from 0 to 2n-2. Input/output lines are indexed from 0 to N-1 and $(i)^j=(i_{n-1}i_{n-2}...i_0)^j$ is the ith input/output of the jth stage, if the index of the stage should be emphasized, where $i_{n-1}i_{n-2}...i_0$ is the binary representation of i. Both indexing schemes are viewed from top down in figures. A path in a network, under a certain set of switching element settings, is a route from an input terminal to an output terminal, through intermediate links and connections of switching elements under that set of settings. We also use $\bar{x}$ to denote the complement of x. We shall use the fault model and similar fault classifications in [3]. A brief review is given below.

A fault can occur at either a link or a switching element. A faulty link is one being stuck-at-zero (s-a-0) or stuck-at-one (s-a-1). On the other hand, we have sixteen possible states for a 2x2 cross point switching element, as shown in Table 1. Only states $S_{10}$ and $S_5$ are considered to be valid. Let S be the set of all sixteen possible states in Table 1. We define the cartesian product S×S as the set of functional states. Assuming that the first valid state is $S_{10}$ and the second $S_5$, a functional state $(S_i,S_j)$ of a switching element is interpreted as the switching element responds in states $S_i$ and $S_j$ while it should be in states $S_{10}$ and $S_5$ respectively. In states $S_0$, $S_1$, $S_2$, $S_4$, $S_6$, $S_8$ and $S_9$ we have logically unidentified outputs denoted by "-" and in $S_6$, $S_7$, $S_9$, $S_{11}$, $S_{13}$, $S_{14}$ and $S_{15}$, we have logically erroneous outputs denoted by "$\phi$". The output values of "-" and "$\phi$" depend on the circuit implementation. However, an arbitrary assignment of 0 or 1 to them will not affect our capability of differentiating normal outputs from faulty outputs. It is clear that the cardinality of S×S is 256, only $(S_{10},S_5)$ is considered normal and the remaining 255 states are all considered faulty. Here we only consider the case of single fault, i.e. either a link stuck fault or a switching element fault, but not both.

## 2. Fault Detection

Let $\oplus i=i_{n-1}\oplus i_{n-2}\oplus...\oplus i_0$, where $\oplus$ is the exclusive-or operator. Also let $a=(a_0,a_1,...,a_{N-1})$, where $a_i=\oplus i$ and $0\leq i<N$, and $b=(b_0,b_1,...,b_{N-1})$, where $b_i=1$ if $\oplus i=0$ and 0 if $\oplus i=1$, and $0\leq i<N$, be the two test vectors in our diagnostic procedure. Notice that $\bar{a}=b$ or $\bar{a}_i=b_i$ for all i. It is easy to see that each switching element receives both 0 and 1 as its inputs when the input vector of the network is a or b, if all the switching elements are set to $S_{10}$ or $S_5$ respectively.

A switching element in states $S_{10}$ and $S_5$ is shown in Figure 1. Table 2 and Table 3 show its normal and faulty output patterns when tested in $S_{10}$ and $S_5$ respectively. It is clear from the tables that we need only to apply both (0,1) and (1,0) to each switching element in order to detect a fault. There are two test phases; in phase-1, all switching elements are set to $S_{10}$ and in phase-2, $S_5$. Figure 2 and Figure 3 show the normal inputs and outputs of a network in phase-1 test and phase-2 test with a and b applied when N=8. Notice that each input vector to a switching element is either (1,0) or (0,1) and so is every output vector. It is easy to see that the output vector is the same as the input vector of a network in a phase-1 test. But in a phase-2 test, the output vector is the complement of the input vector because input i is routed to $\bar{i}$ after the middle stage and then to $i^{\bar{0}}$ (i with the $i_0$ bit complemented) at the output side and inputs i and $i^{\bar{0}}$ are complements to each other. This shows that both input and output vectors are very easy to generate.

It is obvious that we only need to apply a and b in one of the two phases to detect a link stuck fault. However, we may have to test both phases to detect a switching element fault. In any case, four tests are enough for detection.

## 3. Fault Location

Since a switching element can produce none, one or two faulty outputs in either phase of the test, there are several combinations. We divide them into three categories, similar to the cases in [3]. (1)One faulty output appears in the four tests. (2)Two faulty outputs appear in either phase-1 or phase-2 test. (3)Two faulty outputs appear in the four tests, one from phase-1 and the other from phase-2.

There are twelve faulty functional states in category 1. Table 4 shows each faulty functional state, the position and pattern of the faulty output, and the phase of test in which a fault occurs. It is clear that every switching element on the path leading to the faulty output could be faulty. A process, similar to the one in [3], using the principle of binary search, can pinpoint the fault in $\lceil\log(2n-1)\rceil$ steps. Then, by supplying the faulty switching element identical inputs, we are able to distinguish the $\phi\phi$ case from the -- case. In summary, for category 1, the location and the functional state of a faulty switching element can be determined within $6+2\lceil\log(2n-1)\rceil$ tests.

For category 2, every state of $S_0$, $S_1$, $S_4$, $S_5$, $S_6$, $S_7$, $S_9$, $S_{13}$, $S_{15}$ in phase-1 and $S_0$, $S_2$, $S_6$, $S_8$, $S_9$, $S_{10}$, $S_{11}$, $S_{14}$, $S_{15}$ in phase-2 produces two faulty outputs associated with two paths. In most cases, except the one in which the faulty switching element is in the middle stage, the two paths have two switching elements in common, as the following theorem shows.

**Lemma 1:** For a switching element in stage j, $0\leq j<n-1$, there is another switching element in stage 2n-2-j, which takes the two outputs of the former as its inputs in either phase-1 test or phase-2 test.

**Proof:** Let i and $i^{\bar{0}}$ be the two output links of a switching element in stage j where $0\leq j<n-1$. In the phase-1 test, they go to $i_j...i_0i_{n-1}...i_{j+1}$ and $i_j...\bar{i}_0i_{n-1}i_{j+1}$ on the output side of the middle stage respectively after n-1-j shuffles. Now n-1-j unshuffles take them to i and $i^0$, which are the two inputs of a switching element,

0190-3918/86/0000/0737 $01.00 © 1986 IEEE

on the input side of stage 2n-2-j. In the phase-2 test, they go to $i_j \ldots i_0 \bar{i}_{n-1} \ldots \bar{i}_{j+1}$ and $i_j \ldots \bar{i}_0 \bar{i}_{n-1} \ldots \bar{i}_{j+1}$ on the output side of the middle stage respectively after n-1-j shuffles and exchanges. Now n-1-j unshuffles and exchanges take them to $i_{n-1} \ldots \bar{i}_{j+1} \ldots i_0$ and $i_{n-1} \ldots \bar{i}_{j+1} \ldots \bar{i}_0$ on the input side of stage 2n-2-j. Hence in either case, the lemma is true.□

**Theorem 2:** In either phase-1 test or phase-2 test, two paths can only meet on at most two switching elements.

**Proof:** If two paths meet on a switching element, whose indices of input/output links are i and $i^{\bar{0}}$, in the middle stage, then they do not meet in any other switching element. Because, in the phase-1 test, the two paths take on the links $i_{n-j-2} \ldots i_0 i_{n-1} \ldots i_{n-j-1}$ and $i_{n-j-2} \ldots \bar{i}_0 i_{n-1} \ldots i_{n-j-1}$ on the output side of the jth stage and on the input side of the (2n-2-j)th stage, where $0 \leq j < n-1$. It is clear that in every stage, except the middle one, the two $i_0$ bits are always different and they are never on the least significant bit. Hence they only meet in the middle stage. Similarly, the conclusion is proved for phase-2 test.

If they meet in a switching element in the first half of the network, then they also meet in the second half as proved in the last lemma. We claim that they do not meet anywhere else. Notice that, in the proof of the above lemma, the $i_0$ bits are always different and not on the least significant position between stages j and 2n-2-j, hence they do not meet there. Beyond stages j and 2n-2-j, the two paths do not meet either, because unshuffles only move the $i_0$ bits away from the least significant bit position. Due to the symmetry of the network, we have the same conclusion for the case that if the two paths meet in the second half.

Therefore, the two paths, if they ever meet, can meet only on at most two switching elements.□

*Example:* Figure 4 shows two faulty paths meeting in the first and the last stage in the phase-2 test.

Now we describe a procedure to differentiate the two possible faulty switching elements in the following theorem, if the two paths have two switching elements in common.

**Theorem 3:** The two switching elements common to the two faulty paths can be differentiated in two tests.

**Proof:** The two faulty paths clearly pass two switching elements in the middle stage. By the last theorem, they must be different. If the fault occurs in the phase-1 test, set the two switching elements in the middle stage to $S_5$ while leaving all others in $S_{10}$ and test the network with a and b. If the faulty switching element is in the second half of the network, the faulty outputs clearly will appear in the original faulty output teminals. If the faulty switching element, with output links i and $i^{\bar{0}}$, is in the first half, say in stage j, where $j \leq n-1$. The two output links go to $i_j \ldots i_0 i_{n-1} \ldots \bar{i}_{j+1}$ and $i_j \ldots \bar{i}_0 i_{n-1} \ldots \bar{i}_{j+1}$ on the output side of the middle stage and also on the output side of the network through the n unshuffles following the middle stage. However, in the original phase-1 test, the two faulty outputs are on $i_j \ldots i_0 i_{n-1} \ldots i_{j+1}$ and $i_j \ldots \bar{i}_0 i_{n-1} \ldots i_{j+1}$ which are clearly different from the two faulty outputs obtained from the above modified phase-1 test. Therefore, we only have to observe the positions of the faulty outputs in the modified test to determine which switching element is really at fault. If the fault is in the phase-2 test, we set the two switching elements in the middle stage to $S_{10}$ leaving all the rest in $S_5$ and test the network with a and b. A similar argument will give us the same conclusion.

Therefore, in any case, we need only two tests with either of the two modified control settings to locate the fault.□

*Example:* Figure 5 shows the two paths coming out from a switching element in the first stage when the settings of the two switching elements in the middle stage are complemented. Comparing to Figure 4, we see that the switching element in the first stage is the same, but the output terminals from the two paths are different.

**Theorem 4:** In category 2, the location and the functional state of a faulty switching element can be determined within ten tests.

**Proof:** Conduct the first four basic tests. Then find the common switching element(s) of the two faulty paths using the last theorem, if necessary, to obtain a single switching element. After the above six tests, the faulty switching element should have been located. If both the faulty outputs are binary vectors, the functional state is obvious from Table 1. If not, we have to conduct the test process discussed before to differentiate $\phi\phi$ from -- and after these extra tests, the functional state should be obvious. Notice that we may need two more tests for a fault in another test phase. Therefore, we need six tests to locate the fault and two more for each test phase to determine the functional state, which is a total of ten tests.□

Category 3 consists of the cartesian product of $\{S_2,S_3,S_8,S_{11},S_{12},S_{14}\}$ and $\{S_1,S_3,S_4,S_7,S_{12},S_{13}\}$. Every member in the first set produces one fault in phase-1 and every member in the second set produces one fault in phase-2. Again we calculate two faulty paths leading to two faulty outputs, but one from each phase. In order to locate the faulty switching element, we need the following theorem.

**Theorem 5:** The two faulty paths can have at most one link and two switching elements in common in category 3.

**Proof:** Consider a faulty switching element in category 3. It is clear that if it produces one fault in the phase-1 test and the other in the phase-2 test but both on one of its output links, then the switching element, having this faulty link as one of its inputs, in the next stage, if any, is also on both faulty paths. If the faulty switching element produces faults on different output links in different phases, the two faulty paths must join together on one of its input links. Therefore, the switching element, having this link as one of its outputs, in the previous stage, if any, is also on both faulty paths. Accordingly, two faulty paths pass at least one common link and may go through one, two or more common switching elements. Depending on the position of the link, we can prove, in each case, at most one common link and hence two common switching elements may appear as follows:

(1)The link is in the first shuffle, say at the position i of the input side of 0th stage. It goes to i in the phase-1 test and to $i^{\bar{0}}$ in the phase-2 test. Then n-1 shuffles (left circular shifts) and switching stages take them to $i_0 i_{n-1} \ldots i_1$ and $\bar{i}_0 \bar{i}_{n-1} \ldots \bar{i}_1$ respectively on the output side of stage n-1 while for a stage j between 0 and n-1, the two indices should be $i_{n-j-1} \ldots i_0 i_{n-1} \ldots i_{n-j}$ and $i_{n-j-1} \ldots \bar{i}_0 \bar{i}_{n-1} \ldots \bar{i}_{n-j}$ respectively on the output side of that stage. We can see that the two paths do not intersect in the first half of the network after the first shuffle, because the $i_0$ bits are always different. Continuing traversing in the second half, we have n-1 unshuffles and switching stages which take the two paths to $i_{k-n+1} \ldots i_0 i_{n-1} \ldots i_{k-n+2}$ and $i_{k-n+1} \ldots \bar{i}_1 \bar{i}_0 \bar{i}_{n-1} \ldots \bar{i}_{k-n+2}$ respectively at the output side of stage k, where $n-1 < k < 2n-2$, and to $i_{n-1} i_{n-2} \ldots i_0$ and $i_{n-1} \ldots \bar{i}_1 i_0$ respectively on the output side of stage 2n-2. Then followed by the last unshuffle, they go to $i_0 i_{n-1} \ldots i_1$ and $i_0 i_{n-1} \ldots \bar{i}_1$ respectively. In any stage, the $i_1$ bits are always different. Therefore, the two paths can join only at one link.

(2)The link is in the last unshuffle. Since an $\Omega^r\Omega^{-1}$ network is symmetrical with respect to the middle stage, we obtain another $\Omega^r\Omega^{-1}$ network if we view the former in the reversed direction, that is interchanging the roles inputs and outputs. Therefore, this case is the same as (1), if considered in reversed direction. Consequently, the claim is also true in this case.

(3)The link is in the first half of the network but not in the first shuffle, say at input position i of the jth stage where $0 < j < n$. The two paths coming out from the jth stage are on i and $i^{\bar{0}}$ respectively. Shuffles and switching stages take them to $i_{n-k+j-1} \ldots i_0 i_{n-1} \ldots i_{n-k+j}$ and $i_{n-k+j-1} \ldots \bar{i}_0 \bar{i}_{n-1} \ldots \bar{i}_{n-k+j}$ on the output side of stage k, $j \leq k < n$. They differ from each other at least on the $i_0$ bit. Now unshuffles and switching stages route them to different positions in each stage of the second half, since the $i_{n-k+j}$ bits of the two paths are always different from each other. Hence the two paths do not join on the right side of the faulty link. For the left side, i goes to output position $i_0i_{n-1}...i_1$ of the (j-1)th stage, on the input side of the same stage, the two paths take on positions $i_0i_{n-1}...i_1$ and $i_0i_{n-1}...\bar{i}_1$ respectively. Then we have j-1 shuffles and j-2 switching stages. The two paths do not join in this part either, because the $i_1$ bits are different from each other. Therefore, the two paths coincide only on the faulty link.

(4)The link is in the second half of the network but not in the last unshuffle. Due to the symmetry of $\Omega^r\Omega^{-1}$ networks, this case viewed reversely is the same as (3). Hence the claim is also true here.□

*Example:* Figure 6 shows two faulty paths in category 3. As we can see, there are two switching elements and one link in common.

Excluding the one switching element case, we are guaranteed to locate two possible faulty switching elements by the above theorem. This category can further be divided into six cases defined in Table 5 and Table 6 as in [3]. Case F and link stuck fault are not distinguishable as suggested in the proof of the above theorem. Processes similar to those in [3] are used to derive the following result: the location and functional state of a faulty switching element can be determined within eight tests for case A, ten for cases B and C and twelve for cases D and E.

## 4. An Alternative Procedure

If we allow more bits for test vectors, some speedup may be achieved. Instead having the previous a nd b, we shall use n+1 bits for each input/output. Numbers between 0 and 2N-1 are used when their binary representations (n+1 bits) being mentioned for the purpose of convenience. Let $e=(e_0,e_1,...,e_{N-1})$, where $e_i=i+1$ for $0\leq i<N$, be our test vector, that is $e_i$ will be fed into input terminal i in a test. Similarly we have two phases, again, called phase-1 test and phase-2 test. In phase-1, all switching elements are set to $S_{10}$ except those, in the middle stage, whose $(\oplus i)$'s of the indicies of their upper input/output links are equal to 1, being set to $S_5$. In phase-2, the settings are all complements to those of phase-1. In other words, all switching elements are set to $S_5$ except those, in the middle stage, with $(\oplus i)$'s of the indices of their upper input/output links equal to 1, being set to $S_{10}$. Figure 7 and Figure 8 show the normal inputs and outputs in each phase of test when N=8. In either phase, the test vector is e. Outputs can easily be calculated as follows. Let $\ominus i=i_{n-1}\oplus i_{n-2}\oplus...\oplus i_1$, i.e. the exclusive-or of all bits except the 0th bit. In phase-1, the input terminal i goes to link i on the input side of the middle stage. If $\ominus i=0$, it goes to i on the output side and goes also to the output terminal i. If $\ominus i=1$, then it goes to $i^{\bar{0}}$ on the output side and goes to the output terminal $i^{\bar{0}}$. Therefore, on the output terminal side, position i has $e_i=i+1$, the number at input i, if $\ominus i=0$ and has $e_{(i^{\bar{0}})}=i^{\bar{0}}+1$, the number at the neighbor of input i, if $\ominus i=1$. However, in phase-2, it is more complicated. Input terminal i goes to $(\bar{i}_{n-1}...\bar{i}_1i_0)=\bar{i}^0$ (this means that every bit is complemented except the 0th bit) on the input side of the middle stage. The value of $\ominus\bar{i}^0$ depends on n and $\ominus i$. We have four cases: (1) n even and $\ominus i=0$, then $\ominus\bar{i}^0=1$. (2) n even and $\ominus i=1$, then $\ominus\bar{i}^0=0$. (3) n odd and $\ominus i=0$, then $\ominus\bar{i}^0=0$. (4) n odd and $\ominus i=1$, then $\ominus\bar{i}^0=1$. Hence, on the output side of the middle stage, i goes to $\bar{i}^0$ in cases (1) and (4) and to $\bar{i}$ in cases (2) and (3). Finally on the output terminal side, i goes to i in cases (1) and (4) and to $i^{\bar{0}}$ in cases (2) and (3). Therefore, output terminal i has $e_i=i+1$ in cases (1) and (4) and has $e_{(i^{\bar{0}})}=i^{\bar{0}}+1$ in cases (2) and (3). Thus both the control settings and the outputs of the network in each phase can be calculated efficiently.

It is obvious that each switching element in the tests receives two different numbers as inputs under normal conditions. Any switching element fault or link stuck fault will show eventually at the output side of the network. Therefore, in this procedure, we need only to perform two basic tests, phase-1 with e and phase-2 with e, in order to detect any single fault. However, we should bear in mind that there is a factor n+1 left out in this count. Some notations are modified, for example, $\phi\phi$ is replaced by a number between 0 and 2N-1, -- by -, 00 by 0 and 11 by 2N-1. A basic theorem without proof is given below.

**Theorem 6:** Two paths (in either phase) have at most one switching element in common.

*Example:* Figure 9 shows two paths in the phase-1 test of the procedure. As we can see, they meet at exactly one switching element.

Based on the above theorem, similar diagnostic procedure can be developed. In many cases, the latter is faster due to more information from more bits and special settings for switching elements.

## 5. Conclusion

A fault diagnostic procedure for rearrangeable $\Omega^r\Omega^{-1}$ netwroks has been presented. The number of tests in most cases is a constant, which makes this procedure very efficient compared to the one in [2]. Faster procedures are expected if more information is used to conduct tests. However, the number of tests may very with the size of the network. Therefore, a mix of the above may achieve better efficiency.

## REFERENCE

[1] Lee, K.Y., "On the Rearrangeability of a 2logN-1 Stage Permutation Network", *Proc. 1981 Int'l Conf. Parallel Processing:* 221-228, 1981.

[2] Opferman, D.C. and Tsao-wu, N.T., "On a Class of Rearrangeable Switching Networks", *Bell Sys. Tech. J.* 50(5):1579-1618, May-June, 1971.

[3] Feng, T.-Y. and Wu, C.-L., "Fault Diagnosis for a Class of Multistage Interconnection Netwroks", *IEEE Trans. Comput.* C-30(10):743-758, October, 1981.

[4] Wu, C.-L. and Feng, T.-Y., "The Reverse-Exchange Interconnection Network", *IEEE Trans. Comput.* C-29(9):801-811, September, 1980.

[5] Wu,C.-L. and Feng, T.-Y., "On a Class of Multistage Interconnection Networks", *IEEE Trans. Comput.* C-29(8):694-702, August, 1980.

[6] Argawal, D.P., "Testing and Fault Tolerance of Multistage Interconnection Networks", *Computer* 15(4):41-53, April, 1982.

[7] Lawrie, D.H., "Access and Alignment of Data in an Array Processor", *IEEE Trans. Comput.* C-24(12):1145-1155, December, 1975.

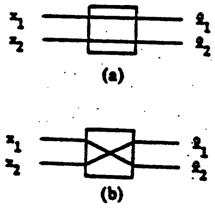


Figure 1: (a)$S_{10}$, (b)$S_5$.

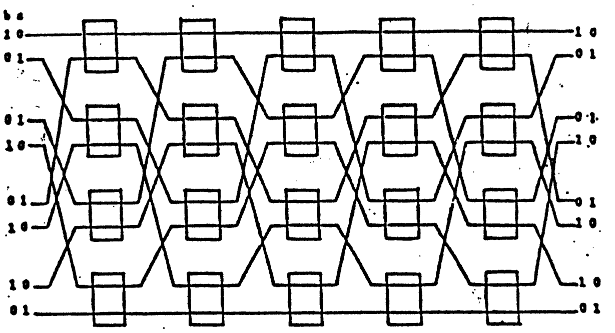


Figure 2: Phase-1 Test

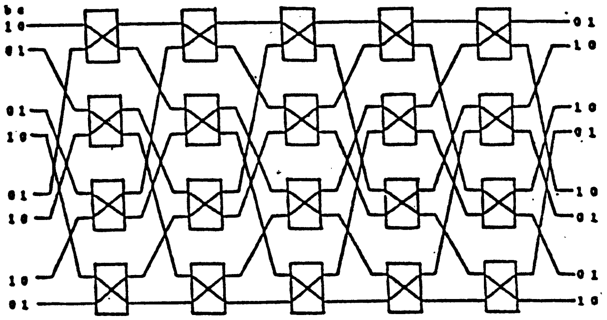


Figure 3: Phase-2 Test

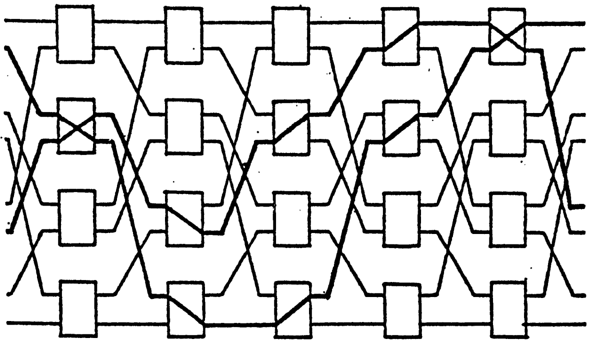


Figure 4: Two Faulty Paths in Category 2

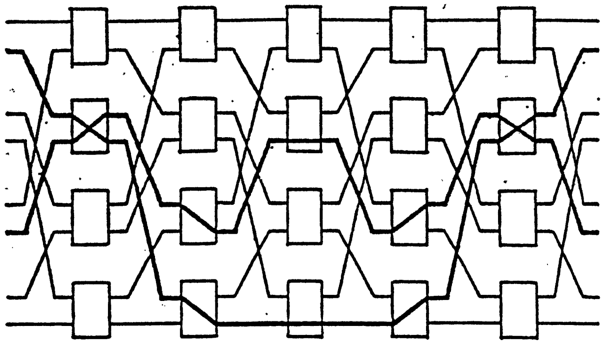


Figure 5: Differentiating Two Switching Elements

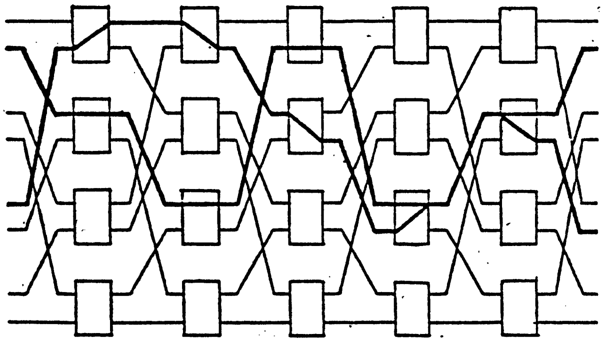


Figure 6: Two Faulty Paths in Category 3

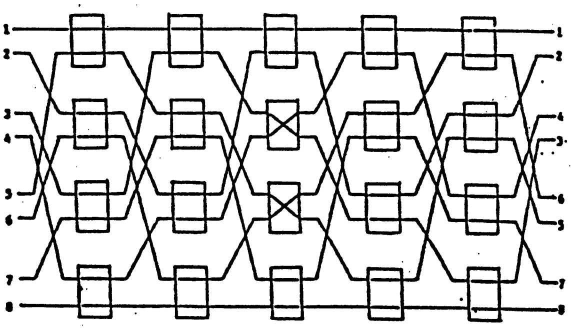


Figure 7: Phase-1 Test

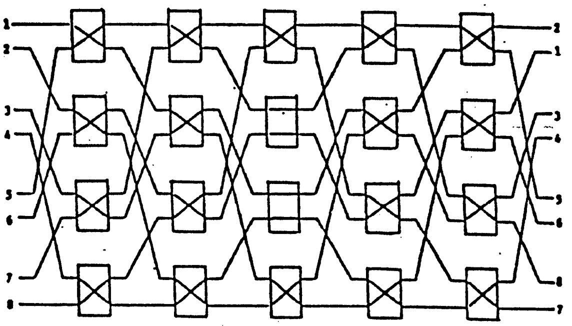


Figure 8: Phase-2 Test

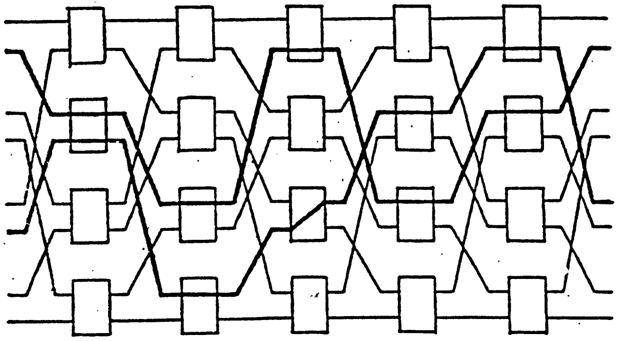


Figure 9: Two Paths in phase-1 test

Table 1: 16 State of a Switching Element.

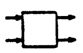
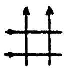
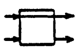
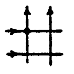
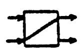
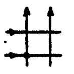
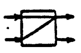
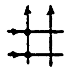
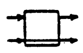
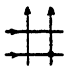
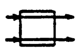
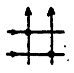
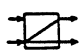
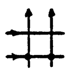
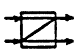
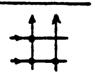
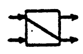
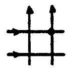
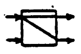
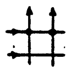
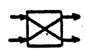
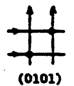
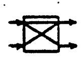
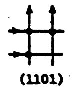
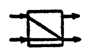
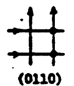
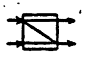
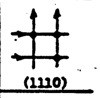
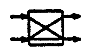
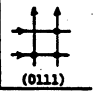
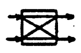
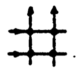
| State Name | Switching Element Symbol | Crosspoint Switching Matrix Symbol | State Name | Switching Element Symbol | Crosspoint Switching Matrix Symbol |
|---|---|---|---|---|---|
| $S_0$ |  |  (0000) | $S_8$ |  |  (1000) |
| $S_1$ |  |  (0001) | $S_9$ |  |  (1001) |
| $S_2$ |  |  (0010) | $S_{10}$ |  |  (1010) |
| $S_3$ |  |  (0011) | $S_{11}$ |  |  (1011) |
| $S_4$ |  |  (0100) | $S_{12}$ |  |  (1100) |
| $S_5$ |  |  (0101) | $S_{13}$ |  |  (1101) |
| $S_6$ |  |  (0110) | $S_{14}$ |  |  (1110) |
| $S_7$ |  |  (0111) | $S_{15}$ |  |  (1111) |

Table 2: Faults, Test Inputs and Outputs in $S_{10}$.

| Fault | | Test | | Output | | | |
|---|---|---|---|---|---|---|---|
| | | | | Normal | | Faulty | |
| | | $x_1$ | $x_2$ | $\hat{x}_1$ | $\hat{x}_2$ | $\hat{x}_1$ | $\hat{x}_2$ |
| Part I: Link Stuck Fault | $x_1^0, \bar{x}_1^0$ | 1<br>1 | 0<br>1 | 1<br>1 | 0<br>1 | 0<br>0 | 0<br>1 |
| | $x_1^1, \bar{x}_1^1$ | 0<br>0 | 0<br>1 | 0<br>0 | 0<br>1 | 1<br>1 | 0<br>1 |
| | $x_2^0, \bar{x}_2^0$ | 0<br>1 | 1<br>1 | 0<br>1 | 1<br>1 | 0<br>1 | 0<br>0 |
| | $x_2^1, \bar{x}_2^1$ | 0<br>1 | 0<br>0 | 0<br>1 | 0<br>0 | 0<br>1 | 1<br>1 |
| Part II: Switching Element Fault | $S_{10}$-$S_0$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | -<br>-<br>-<br>- | -<br>-<br>-<br>- |
| | $S_{10}$-$S_1$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 1<br>0<br>0<br>1 | -<br>-<br>-<br>- |
| | $S_{10}$-$S_2$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | -<br>-<br>-<br>- | 1<br>0<br>0<br>1 |
| | $S_{10}$-$S_3$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | 1<br>0 | 1<br>0 |
| | $S_{10}$-$S_4$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | -<br>-<br>-<br>- | 0<br>1<br>0<br>1 |
| | $S_{10}$-$S_5$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | 1<br>0 | 0<br>1 |
| | $S_{10}$-$S_6$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | -<br>-<br>-<br>- | ϕ<br>ϕ<br>0<br>1 |
| | $S_{10}$-$S_7$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | 1<br>0 | ϕ<br>ϕ |
| | $S_{10}$-$S_8$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | -<br>-<br>-<br>- |
| | $S_{10}$-$S_9$ | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | 0<br>1<br>0<br>1 | 1<br>0<br>0<br>1 | ϕ<br>ϕ<br>0<br>1 | -<br>-<br>-<br>- |
| | $S_{10}$-$S_{11}$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | ϕ<br>ϕ | 1<br>0 |
| | $S_{10}$-$S_{12}$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | 0<br>1 | 0<br>1 |
| | $S_{10}$-$S_{13}$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | ϕ<br>ϕ | 0<br>1 |
| | $S_{10}$-$S_{14}$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | 0<br>1 | ϕ<br>ϕ |
| | $S_{10}$-$S_{15}$ | 0<br>1 | 1<br>0 | 0<br>1 | 1<br>0 | ϕ<br>ϕ | ϕ<br>ϕ |

Table 3: Faults, Test Inputs and Outputs in $S_5$.

| Fault | | Test | | Output | | | |
|---|---|---|---|---|---|---|---|
| | | | | Normal | | Faulty | |
| | | $x_1$ | $x_2$ | $\hat{x}_1$ | $\hat{x}_2$ | $\hat{x}_1$ | $\hat{x}_2$ |
| Part I: Link Stuck Fault | $x_1^0, \hat{x}_2^0$ | 1 | 0 | 0 | 1 | 0 | 0 |
| | | 1 | 1 | 1 | 1 | 1 | 0 |
| | $x_1^1, \hat{x}_2^1$ | 0 | 0 | 0 | 0 | 0 | 1 |
| | | 0 | 1 | 1 | 0 | 1 | 1 |
| | $x_2^0, \hat{x}_1^0$ | 0 | 1 | 1 | 0 | 0 | 0 |
| | | 1 | 1 | 1 | 1 | 0 | 1 |
| | $x_2^1, \hat{x}_1^1$ | 0 | 0 | 0 | 0 | 1 | 0 |
| | | 1 | 0 | 0 | 1 | 1 | 1 |
| Part II: Switching Element Fault | $S_5$-$S_0$ | 0 | 1 | 1 | 0 | - | - |
| | | 1 | 0 | 0 | 1 | - | - |
| | | 0 | 0 | 0 | 0 | - | - |
| | | 1 | 1 | 1 | 1 | - | - |
| | $S_5$-$S_1$ | 0 | 1 | 1 | 0 | 1 | - |
| | | 1 | 0 | 0 | 1 | 0 | - |
| | | 0 | 0 | 0 | 0 | 0 | - |
| | | 1 | 1 | 1 | 1 | 1 | - |
| | $S_5$-$S_2$ | 0 | 1 | 1 | 0 | - | 1 |
| | | 1 | 0 | 0 | 1 | - | 0 |
| | | 0 | 0 | 0 | 0 | - | 0 |
| | | 1 | 1 | 1 | 1 | - | 1 |
| | $S_5$-$S_3$ | 0 | 1 | 1 | 0 | 1 | 1 |
| | | 1 | 0 | 0 | 1 | 0 | 0 |
| | $S_5$-$S_4$ | 0 | 1 | 1 | 0 | - | 0 |
| | | 1 | 0 | 0 | 1 | - | 1 |
| | | 0 | 0 | 0 | 0 | - | 0 |
| | | 1 | 1 | 1 | 1 | - | 1 |
| | $S_5$-$S_6$ | 0 | 1 | 1 | 0 | - | ϕ |
| | | 1 | 0 | 0 | 1 | - | ϕ |
| | | 0 | 0 | 0 | 0 | - | 0 |
| | | 1 | 1 | 1 | 1 | - | 1 |
| | $S_5$-$S_7$ | 0 | 1 | 1 | 0 | 1 | ϕ |
| | | 1 | 0 | 0 | 1 | 0 | ϕ |
| | $S_5$-$S_8$ | 0 | 1 | 1 | 0 | 0 | - |
| | | 1 | 0 | 0 | 1 | 1 | - |
| | | 0 | 0 | 0 | 0 | 0 | - |
| | | 1 | 1 | 1 | 1 | 1 | - |
| | $S_5$-$S_9$ | 0 | 1 | 1 | 0 | ϕ | - |
| | | 1 | 0 | 0 | 1 | ϕ | - |
| | | 0 | 0 | 0 | 0 | 0 | - |
| | | 1 | 1 | 1 | 1 | 1 | - |
| | $S_5$-$S_{10}$ | 0 | 1 | 1 | 0 | 0 | 1 |
| | | 1 | 0 | 0 | 1 | 1 | 0 |
| | $S_5$-$S_{11}$ | 0 | 1 | 1 | 0 | ϕ | 1 |
| | | 1 | 0 | 0 | 1 | ϕ | 0 |
| | $S_5$-$S_{12}$ | 0 | 1 | 1 | 0 | 0 | 0 |
| | | 1 | 0 | 0 | 1 | 1 | 1 |
| | $S_5$-$S_{13}$ | 0 | 1 | 1 | 0 | ϕ | 0 |
| | | 1 | 0 | 0 | 1 | ϕ | 1 |
| | $S_5$-$S_{14}$ | 0 | 1 | 1 | 0 | 0 | ϕ |
| | | 1 | 0 | 0 | 1 | 1 | ϕ |
| | $S_5$-$S_{15}$ | 0 | 1 | 1 | 0 | ϕ | ϕ |
| | | 1 | 0 | 0 | 1 | ϕ | ϕ |

Table 4: Category 1

| Faults of Category 1 | Upper (U) or Lower (L) Link by Which the Faulty Switching Element Sends the Fault | Test Phase at Which Fault Appears. | Faulty Output —: (00 or 11)<br>ϕϕ: (00 or 11)<br>Binary Vector (10 or 01) |
|---|---|---|---|
| $(S_2,S_5)$ | U | 1 | — |
| $(S_3,S_5)$ | U | 1 | Binary Vector |
| $(S_8,S_5)$ | L | 1 | — |
| $(S_{11},S_5)$ | U | 1 | ϕϕ |
| $(S_{12},S_5)$ | L | 1 | Binary Vector |
| $(S_{14},S_5)$ | L | 1 | ϕϕ |
| $(S_{10},S_1)$ | L | 2 | — |
| $(S_{10},S_3)$ | L | 2 | Binary Vector |
| $(S_{10},S_4)$ | U | 2 | — |
| $(S_{10},S_7)$ | L | 2 | ϕϕ |
| $(S_{10},S_{12})$ | U | 2 | Binary Vector |
| $(S_{10},S_{13})$ | U | 2 | ϕϕ |

Table 5: Six Cases in Category 3

| | Faulty Outputs | |
|---|---|---|
| Case | 1 | 2 |
| A | 01 or 10<br>Binary Vector | 01 or 10<br>Binary Vector |
| B | 01 or 10<br>Binary Vector | ϕ ϕ |
| C | 01 or 10<br>Binary Vector | - - |
| D | ϕ ϕ | - - |
| E | ϕ ϕ | ϕ ϕ |
| F | - - | - - |

Table 6: Functional States in Category 3

| Valid State $S_{10}$ / Valid State $S_5$ | $S_{12}$ | $S_3$ | $S_2$ | $S_8$ | $S_{11}$ | $S_{14}$ |
|---|---|---|---|---|---|---|
| $S_{12}$ | A | A | C | C | B | B |
| $S_3$ | A | A | C | C | B | B |
| $S_4$ | C | C | F | F | D | D |
| $S_1$ | C | C | F | F | D | D |
| $S_{13}$ | B | B | D | D | E | E |
| $S_7$ | B | B | D | D | E | E |

# IMPACT OF CLUSTER NETWORK FAILURE ON THE PERFORMANCE OF CLUSTER-BASED SUPERSYSTEMS

Imad E. O. Mahgoub and Dharma P. Agrawal

Department of Electrical and Computer Engineering
Box 7911
North Carolina State University
Raleigh, NC 27695

## Abstract

This paper investigates the effect of Cluster Interconnection Network (CINs) failure on the performance of a Cluster-based Supersystem [1]. Three different cases of memory request distribution have been treated. The first case assumes uniform distribution while the second employs the concept of favorite memory. Further generalization of the second case is made in the third case by adding a private memory and associating one with each processor. System performance degradation due to the failure of a single Cluster IN and due to the failure of multiple Cluster INs has been evaluated analytically in terms of available Bandwidth.

## I Introduction

It is a well known fact that the switching speed of logic has almost attained its physical limit and only incremental speed improvement can be expected in future. Multiprocessor/Multicomputer based supersystems [2-8] offer a great potential for substantial performance enha9 offer a great potential for substantial is how to design an appropriate Interconnection Network to cater to communication between various functional modules in the system [10]. Several interconnection schemes have been proposed [11] and there are relative advantages and disadvantages of each one of them. A single shared bus scheme is the simplest and economical while cross-bar switches represent another end of the spectrum. The bus scheme suffers from the contention problem while the complexity and the cost of a crossbar is proportional to its total number of cross-points. As a compromise between the two, several multistage networks [10-12] and multibus systems [13-14] have been proposed which could provide a reasonable performance at an acceptable cost.

In [1] we described and analyzed Cluster-based Supersystems which have a hierarchy that reduces the network complexity. Conflict-free access within each cluster is secured by relatively smaller crossbar switches. To maintain the hierarchy and the regularity of the structure, clusters are connected by a network similar to the one used for interconnecting processors within each cluster. These systems take full advantage of program locality and they have been shown to perform very close to the crossbar-based systems for higher rate of favorite requests. However, the analysis carried out in [1] assumed that the system operates in a fault-free mode which may not be realistic . In reallity, cluster-based supersystems are prone to faults that could at least degrade their performance.

In this paper we have dealt with the situation where one or more cluster INs could fail. Basic assumptions and other essential background material is included in section II. Performance degradation due to the loss of one cluster IN has been calculated in section III. In section IV, the performance degradation is determined in the event of total loss of all cluster INs. Concluding remarks have been included in section V.

## II Background

In the analysis of the cluster-based supersystems [1], we assume the following model to exist:

(i) The multiprocessor system is synchronous with N processors and N memory modules , in which the processors issue their requests at the beginning of a memory cycle.

(ii) Memory cycle time is the same as the processor cycle time and is constant.

(iii) The requests generated in a cycle are random and are independent of one another.

(iv) The requests issued in successive cycles are independent of the requests issued in the previous cycle.

(v) The requests which are not accepted are discarded.

(vi) Propagation delay of the interconnection network is neglected.

(vii) $\psi$ is the probability with which a processor generates a request in a cycle.

The fourth assumption is unrealistic because a discarded request will indeed be resubmitted in the next cycle. However, this assumption simplifies the analysis and produces negligible discrepancies from the actual results [17].

It is also assumed that $N = n.k$ [1]. This enables us to have k - clusters with each cluster consisting of n - processors. Crossbar switches of size n x n are used for each cluster ( cluster IN ) while a kxk global crossbar switch is used to interconnect all k - clusters together (Fig.2). All the processors forming an individual cluster , have to pass through an associated single bus to gain access to the global IN.

The system's performance is analyzed on the basis of the analysis of crossbar [15,14] and shared-bus schemes , both of which have been included here just for the completeness of the text.

0190-3918/86/0000/0743 $01.00 © 1986 IEEE

### Analysis of a crossbar

Consider an NxN crossbar , and let $p_c$ be the probability that a request is delivered to an input of the crossbar. The probability that this request is directed to a particular output of the N outputs is $p_c$/N. The probability that this particular output is not requested by any of the N inputs is $(1 - p_{c/N})^N$. The probability that this particular output is requested by at least one input

$$= 1 - (1 - p_{c/N})^N \tag{1}$$

This is the well known formula for the crossbar [15].

### Shared bus analysis

We adopt the model for the shared bus Introduced in [16] which views the shared bus as a cascade of an Nx1 crossbar and a 1xN crossbar. Let $p_0$ be the probability that the bus is requested by a processor out of the N processors tied to the bus , then the probability that none of the N processors request the bus is $(1 - p_0)^N$. The probability at least one processor will request the bus is $1 - (1 - p_0)^N$.

Hence the rate of request at the bus

$$= 1 - (1 - p_0)^N \tag{2}$$

Rate of request at a particular output line out of the N output lines tied to the bus

$$= (1 - (1 - p_0)^N)/N \tag{3}$$

It may be noted that if there is only one output line tied to the bus then eq.(3) reduces to eq.(2).

### Fault-free crossbar-based and cluster-based Multiprocessors

The fault-free analysis of the crossbar-based and the cluster-based multiprocessors has been carried out in [1] for three different cases (based on the distribution of requests among the memory modules) namely: Equally likely , favorite memories , and favorite with private memories. To avoid the duplication, analysis for the fault-free system will not be included here. Some data is presented in the form of Tables (Tables I, II, III) and graphs (Figs. 5, 6, 7) so that a comparison between performance of the fault-free and the faulty systems could be done.

It may be noted that minor modifications in analysis given in [1] is needed to allow for the restriction imposed in this paper, namely, a processor is internally connected only to its private memory. This change is deemed necessary for a fair comparison between the fault-free and faulty cases.

### Faulty cluster-based multiprocessors

The analysis performed here, deals only with the failure of the cluster interconnection networks. There are k nxn CINs while there is only one kxk global IN and the global network is most important from the point of view of keeping all the clusters connected together. Hence, it is desirable to have a reliable global IN by designing it with appropriate care, or alternatively to duplicate the global IN and keep the second global IN as a standby spare. For our analysis, the global IN is assumed to be fault-tolerant and failures are assumed to occur only in the CINs.

In the event of failure of a cluster IN requests from the processors to the memory modules in that cluster are directed through the global IN. Although the system is still operational , its performance will be degraded. This degradation will be the least when a single cluster IN fails , and the most when the cluster INs in all the clusters fail. Bearing in mind that the performance of the system for any number of failed cluster INs will lie between these two limits , we are only going to concern ourselves with the analysis of these two extreme cases.

## III Failure of a single cluster IN

Failure of a single cluster IN results in a situation where two groups of memory modules are formed and each group has a different rate of request (Fig.3). The first group consists of those memory modules which are part of the cluster with failed IN , while the remaining memory modules residing in the remaining healthy clusters form the second group.

Let

$p_{ext} \overset{\Delta}{=}$ the probability that a processor issues a request to a memory module external to its own cluster.

$p_G \overset{\Delta}{=}$ the rate at which an input of the global IN is requested.

$p_C \overset{\Delta}{=}$ the rate at which an input of the cluster IN is requested.

$p_{fi} \overset{\Delta}{=}$ the rate of request at a memory module due to requests from favorite processors directed through the cluster IN.

$p_{fo} \overset{\Delta}{=}$ the rate of request at a memory module due to requests from favorite processors directed through the global IN (in the event of failure of the cluster IN).

$p_{nf} \overset{\Delta}{=}$ the rate of requests at a memory module due to requests from nonfavorite processors through the global IN.

### 3.1 Equally likely case

This is the case where the requests generated by a processor are uniformly distributed among the memory modules.

(a) First group analysis ( group1 )

Let $p_{ext1}$ be the probability that a request generated by a processor of group1 will be directed through the global IN, hence

$$p_{ext1} = \psi \tag{4}$$

( since no requests can be made through the cluster IN )

Let $p_{G1}$ be the rate at which the input of the global IN (accessible by processors of group1) is requested, hence

$$p_{G1} = 1 - (1 - p_{ext1})^n \tag{5}$$

Let $p_1^{(1)}$ be the rate of request at a memory module of group1 due to requests issued by processors in group1.

$$p_1^{(1)} = p_{G1/N} \tag{6}$$

Let $p_2^{(1)}$ be the rate of request at a memory module of group1 due to requests issued by processors in the second group.

$$p_2^{(1)} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\ \ 1}]/n \tag{7}$$

Hence the total rate of request at a memory module of group1

$$\equiv p_1 = p_1^{(1)} + p_2^{(1)} - p_1^{(1)} p_2^{(1)} \tag{8}$$

(b) Second group analysis ( group2 )

Let $p_{ext2}$ be the probability that a processor of group2 will issue a request to a memory module external to its own cluster, hence

$$p_{ext2} = (k - 1)\psi/k \tag{9}$$

Let $p_{G2}$ be the rate at which an input of the global IN (accessible by processors of group2) is requested, hence

$$p_{G2} = 1 - (1 - p_{ext2})^{n} \tag{10}$$

Let $p_2^{(2)}$ be The rate of request at a memory module of group2 due to requests issued by the processors of group2 but not belonging to the same cluster.

$$p_2^{(2)} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\quad 2}]/n \tag{11}$$

Let $p_1^{(2)}$ be The rate of request at a memory module of group2 due to requests issued by processors of group1.

$$p_1^{(2)} = p_{G1}/N \tag{12}$$

$$p_C = \psi/k \tag{13}$$

Let $p_{i2}^{(2)}$ be The rate of request at a memory module of group2 due to requests issued by processors of group2 and in the same cluster.

$$p_{i2}^{(2)} = 1 - (1 - p_C/n)^{n} \tag{14}$$

Hence the total rate of request at a memory module of group2

$$\equiv p_2 = p_{i2}^{(2)} + p_1^{(2)} + p_2^{(2)} - p_{i2}^{(2)}p_1^{(2)} - p_{i2}^{(2)}p_2^{(2)}$$
$$- p_1^{(2)}p_2^{(2)} + p_{i2}^{(2)}p_1^{(2)}p_2^{(2)} \tag{15}$$

The Bandwidth

$$\equiv BW = p_1.n + p_2.(N - n) \tag{16}$$

Table IV gives the degraded performance of the cluster-based multicomputer system for the equally likely case.

### 3.2 Favorite memories case

In this case processors (favorite) communicate more often with memory modules (favorite) in the same cluster as the processors. Let s be the probability that a processor addresses a particular favorite memory given that it has generated a request.

(a) First group analysis

$$p_{ext1} = \psi \tag{17}$$

$$p_{G1} = 1 - (1 - p_{ext1})^{n} \tag{18}$$

$$p_{fo} = sp_{G1} \tag{19}$$

Let $p_{nf2}^{(1)}$ be The rate of request at a memory module of group1 due to requests issued by nonfavorite processors.

$$p_{nf2}^{(1)} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\quad 1}]/n \tag{20}$$

Hence the rate of request at a memory module of group1

$$\equiv p_1 = p_{fo} + p_{nf2}^{(1)} - p_{fo}p_{nf2}^{(1)} \tag{21}$$

(b) Second group analysis

$$p_{ext2} = (1 - ns)\psi \tag{22}$$

$$p_{G2} = 1 - (1 - p_{ext2})^{n} \tag{23}$$

Let $p_{nf2}^{(2)}$ be the Rate of request at a memory module of group2 due to requests issued by the non favorite processors in group2.

$$p_{nf2}^{(2)} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\quad 2}]/n \tag{24}$$

Let $p_{nf1}^{(2)}$ be the Rate of request at a memory module of group2 due to requests issued by the nonfavorite processors in group1.

$$p_{nf1}^{(2)} = [(1 - ns)\frac{p_{G1}}{k-1}]/n \tag{25}$$

$$p_C = ns\psi \tag{26}$$

$$p_{fi} = 1 - (1 - p_C/n)^{n} \tag{27}$$

Hence the total rate of request at a memory module of group2

$$\equiv p_2 = p_{fi} + p_{nf2}^{(2)} + p_{nf1}^{(2)} - p_{fi}p_{nf2}^{(2)} - p_{fi}p_{nf1}^{(2)}$$
$$- p_{nf1}^{(2)}p_{nf2}^{(2)} + p_{fi}p_{nf2}^{(2)}p_{nf1}^{(2)} \tag{28}$$

$$BW = p_1.n + p_2.(N - n) \tag{29}$$

The results for this case are given in Table V.

### 3.3 Favorite with private memories case

A private memory is added to each processor and each processor accesses its private memory through an internal bus connection.

Let m be the probability that a processor requests its private memory given that it has generated a request.

(a) First group analysis

$$p_{ext1} = (1 - m)\psi \tag{30}$$

$$p_{G1} = 1 - (1 - p_{ext1})^{n} \tag{31}$$

$$p_{fo} = sp_{G1} \tag{32}$$

The rate of request at a memory module of group1 due to requests issued by nonfavorite processors

$$\equiv p_{nf2}^{(1)} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\quad 1}]/n \tag{33}$$

Hence the total rate of request at a memory module of group1

$$\equiv p_1 = p_{fo} + p_{nf2}^{(1)} - p_{fo}p_{nf2}^{(1)} \tag{34}$$

(b) Second group analysis

$$p_{ext2} = (1 - ns - m)\psi \tag{35}$$

$$p_{G2} = 1 - (1 - p_{ext2})^{n} \tag{36}$$

The rate of request at a memory module of group2 due to requests issued by the non favorite processors in group2

$$\equiv p^{(2)}_{nf2} = [1 - (1 - \frac{p_{G2}}{k-1})^{k\ \ 2}]/n \tag{37}$$

The Rate of request at a memory module of group2 due to requests issued by the nonfavorite processors in group1

$$\equiv p^{(2)}_{nf1} = [(1 - ns)\frac{p_{G1}}{k-1}]/n \tag{38}$$

$$p_C = ns\psi \tag{39}$$

$$p_{fi} = 1 - (1 - p_C/n)^n \tag{40}$$

Hence the total rate of request at a memory module of group2

$$\equiv p_2 = p_{fi} + p^{(2)}_{nf2} + p^{(2)}_{nf1} - p_{fi}p^{(2)}_{nf2} - p_{fi}p^{(2)}_{nf1} - p^{(2)}_{nf1}p^{(2)}_{nf2} + p_{fi}p^{(2)}_{nf2}p^{(2)}_{nf1} \tag{41}$$

$$BW = p_1.n + p_2.(N - n) \tag{42}$$

Table VI indicates the performance degradation for this case.

## IV Failure of the INs in all the clusters

In this situation, all requests generated by processors will be delivered to memory modules through the global IN (Fig.4).

Hence,

$$p_{ext} = \psi \tag{43}$$

$$p_G = 1 - (1 - p_{ext})^n \tag{44}$$

### 4.1 Equally likely case

The total rate of request at a memory module

$$\equiv p = [1 - (1 - p_G/k)^k]/n \tag{45}$$

$$BW = pN. \tag{46}$$

Results for this case are given in Table VII.

### 4.2 Favorite memories case

$$p_{fo} = p_G s \tag{47}$$

$$p_{nf} = [1 - (1 - (1 - sn)\frac{p_G}{k-1})^{k\ \ 1}]/n \tag{48}$$

The total rate of request

$$\equiv p = p_{fo} + p_{nf} - p_{fo}p_{nf} \tag{49}$$

$$BW = pN. \tag{50}$$

Table VIII indicates the degradation of performance for this case.

### 4.3 Favorite with private memories case

Again, with the exception of requests to the private memories, all requests are delivered through the global IN.

$$p_{ext} = (1 - m)\psi \tag{51}$$

$$p_G = 1 - (1 - p_{ext})^n \tag{52}$$

$$p_{fo} = p_G s \tag{53}$$

$$p_{nf} = (1 - (1 - (1 - sn)\frac{p_G}{k-1})^{k\ \ 1})/n \tag{54}$$

The total rate of request

$$\equiv p = p_{fo} + p_{nf} - p_{fo}p_{nf} \tag{55}$$

$$BW = pN. \tag{57}$$

The degraded performance for this case is given in Table IX.

## V Conclusions

Performance degradation of cluster-based supersystems due to failure of cluster INS has been analytically determined. Two limiting situations of only one cluster IN failure and all the cluster INs failures have been considered. The analysis has been carried out for three different cases : Equally likely, Favorite memories, and Favorite with Private memories. It has been concluded that the Favorite memories scheme is still superior over the other two schemes (Figs. 5, 6, 7), in spite of steep performance degradation - for the same values of the system parameters.

## Acknowledgement

The work is part of the BHIVE project and was supported in part by the Army Research Office under grants No. DAAG29-85-K-0236.

## References

[1] D.P. Agrawal and Imad E.O. Mahgoub, "Performance Analysis of Cluster-based Supersystems," First Int'l. Conf. on Supercomputing Systems , Dec. 1985, pp. 593-602.

[2] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar-A Large Scale Multiprocessor," 1983 Parallel Processing Conf. Proc. , pp. 524-529.

[3] D.E. Shaw, "SIMD and MSIMD Variants of the Non-Von Supercomputers," Proc. COMPCON , Spring, 1984, pp. 360-363.

[4] K.E. Batcher, "Design of a Massively Parallel Processor," IEEE Trans. on Computers , Vol. C-29, No. 9, Sept. 1980, pp. 1-9.

[5] H.J. Siegel et al., "PASM: A Partitionable SIMD/MIMD system for Image Processing and Pattern Recognition," IEEE Trans. on Computers , Vol. C-30, No. 10, Oct. 1982, pp. 952-962.

[6] C.R. Vick, S.I. Kartashev, and S.P. Kartashev, "Adaptable Architecture for Supersystems," IEEE Computer , Vol. 13, Nov. 1980, pp. 17-35.

[7] L.S. Haynes, R.L. Lau, D.P. Siewiorek and D. Mizell, "A Survey of Highly Parallel Computing," IEEE Computer , Vol. 15, No. 1, Jan. 1982, pp. 9-24.

[8] D.P. Agrawal and W.E. Alexander, "B-HIVE: A Heterogeneous, Interconnected, Versatile, and Expandable Multicomputer System," IEEE-CS Computer Arch. Tech. Comm. , Sept. 1984, pp. 19-25.

[9] A Gottlieb et al., "The NYU Ultracomputer -Designing an MIMD shared Memory Parallel Computer," IEEE Trans. on Computers , Feb. 1983, pp. 175-189.

[10] T.Y. Feng, "A survey of INterconnection Networks," IEEE Computer , Vol. 14, Dec. 1981, pp. 12-27.

[11] C.L. Wu and T.Y. Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. on Computers , Vol. C-29, No. 8, Aug. 1980, pp. 694-702.

[12] L.D. Wittie, "Communication Structure for Large Networks of Micro-computers," IEEE Trans. on Computers , Vol. C-30, No. 4, April 1981 , pp. 264-273.

[13] L.N. Bhuyan and D.P. Agrawal, "Generalized Hypercube and Hyperbus structures for a Computer Network," IEEE Trans. on Computers , Vol. C-33, No. 4, April 1984, pp. 323-333.

[14] T.N. Mudge, J.P. Hayes, G.D. Buzzard, and D.C. Winser, "Analysis of Multiple Bus Interconnection Networks," Proc. 1984 Int'l. Conf. on Parallel Processing , Aug. 21-24, 1984, pp. 228-232.

[15] L.N. Bhuyan and C.W. Lee, "An Interference Analysis of Interconnection Networks," Proc. 1983 Parallel Processing Conf. , pp. 2-9.

[16] L.N. Bhuyan, "A Combinatorial Analysis of Multibus Multiprocessors," Proc. 1984 Parallel Processing Conf. , Aug. 21-24, 1984, pp 225-227.

[17] D.P. Bhandarkar, "Analysis of Memory Interference in Multiprocessors," IEEE Trans. on Computers , Vol. C24, No. 9, Sept. 1975, pp 897-908.

**Table I. Performance of the Fault-free Cluster-based Multiprocessor System ($\psi = 1$).**

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 13.95 | 9.36 | 8.47 |
| 48 | 20.25 | 12.90 | 10.35 |
| 64 | 26.54 | 16.44 | 12.23 |
| 80 | 32.84 | 19.98 | 14.11 |
| 96 | 39.14 | 23.52 | 15.99 |
| 112 | 45.43 | 27.06 | 17.87 |
| 128 | 51.73 | 30.60 | 19.75 |

**Table II. Performance of the Fault-free Cluster-based Multiprocessor System with Favorite Memories ($\psi = 1$).**

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 19.97 | 20.40 | 21.31 |
| 48 | 29.55 | 30.04 | 31.06 |
| 64 | 39.09 | 39.63 | 40.75 |
| 80 | 48.60 | 49.18 | 50.34 |
| 96 | 58.10 | 58.70 | 60.00 |
| 112 | 67.60 | 68.22 | 69.58 |
| 128 | 77.10 | 77.73 | 79.15 |

**Table III. Performance of the Fault-free Cluster-based Multiprocessor System with private and Favorite Memories ($\psi = 1$).**

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 7.17 | 7.40 | 7.87 |
| 48 | 10.24 | 10.60 | 11.34 |
| 64 | 13.21 | 13.67 | 14.66 |
| 80 | 16.15 | 16.69 | 17.87 |
| 96 | 19.06 | 19.65 | 21.ol |
| 112 | 21.97 | 22.59 | 24.09 |
| 128 | 24.87 | 25.52 | 27.12 |

**Table IV. Degraded Performance of the Cluster-based Multiprocessor System Due to Failure of a Single Cluster IN ($\psi = 1$).**

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 8.14 | 8.15 | 8.48 |
| 48 | 11.30 | 10.83 | 10.17 |
| 64 | 14.45 | 13.49 | 11.84 |
| 80 | 17.60 | 16.15 | 13.50 |
| 96 | 20.75 | 18.81 | 15.16 |
| 112 | 23.90 | 21.47 | 16.81 |
| 128 | 27.05 | 24.13 | 18.46 |

**Table V. Degraded Performance of the Cluster-based Multiprocessor System with Favorite Memories Due to Failure of a Single Cluster IN ($\psi = 1$).**

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 11.37 | 16.51 | 19.72 |
| 48 | 16.24 | 23.83 | 28.32 |
| 64 | 21.04 | 31.07 | 36.85 |
| 80 | 25.81 | 38.26 | 45.34 |
| 96 | 30.57 | 45.43 | 53.78 |
| 112 | 35.32 | 52.59 | 62.19 |
| 128 | 40.07 | 59.73 | 70.58 |

Table VI. Degraded Performance of the Cluster-based Multiprocessor System with Private and Favorite Memories Due to Failure of a Single Cluster IN (ψ = 1).

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 4.57 | 6.48 | 7.58 |
| 48 | 6.13 | 8.98 | 10.77 |
| 64 | 7.63 | 11.34 | 13.76 |
| 80 | 9.10 | 13.63 | 16.63 |
| 96 | 10.56 | 15.88 | 19.41 |
| 112 | 12.02 | 18.10 | 22.14 |
| 128 | 13.47 | 20.30 | 24.82 |

Table VII. Degraded Performance of the Cluster-based Multiprocessor System Due to Failure of All Cluster INs (ψ = 1).

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 1.50 | 2.73 | 5.25 |
| 48 | 1.50 | 2.73 | 5.25 |
| 64 | 1.50 | 2.73 | 5.25 |
| 80 | 1.50 | 2.73 | 5.25 |
| 96 | 1.50 | 2.73 | 5.25 |
| 112 | 1.50 | 2.73 | 5.25 |
| 128 | 1.50 | 2.73 | 5.25 |

Table VIII. Degraded Performance of the Cluster-based Multiprocessor System with Favorite Memories Due to Failure of All Cluster INs (ψ = 1).

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 1.99 | 3.94 | 7.79 |
| 48 | 1.99 | 3.96 | 7.85 |
| 64 | 1.99 | 3.97 | 7.88 |
| 80 | 2.00 | 3.97 | 7.90 |
| 96 | 2.00 | 3.97 | 7.91 |
| 112 | 2.00 | 3.97 | 7.92 |
| 128 | 2.00 | 3.98 | 7.92 |

Table IX. Degraded Performance of the Cluster-based Multiprocessor System with Private and Favorite Memories Due to Failure of All Cluster INs (ψ = 1).

| N | BW | | |
|---|---|---|---|
| | k = 2 | k = 4 | k = 8 |
| 32 | 1.97 | 3.02 | 4.84 |
| 48 | 1.97 | 3.15 | 5.49 |
| 64 | 1.99 | 3.18 | 5.80 |
| 80 | 1.99 | 3.20 | 5.96 |
| 96 | 1.99 | 3.20 | 6.04 |
| 112 | 1.99 | 3.21 | 6.08 |
| 128 | 2.00 | 3.21 | 6.11 |

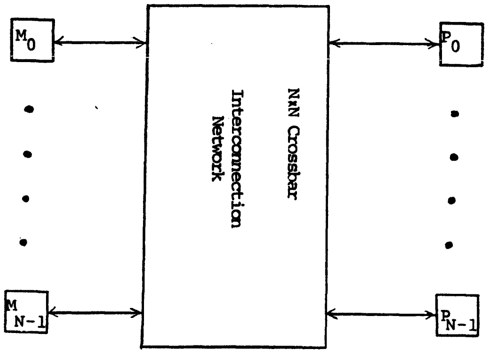


Fig. 1. Crossbar-Based Multiprocessor System

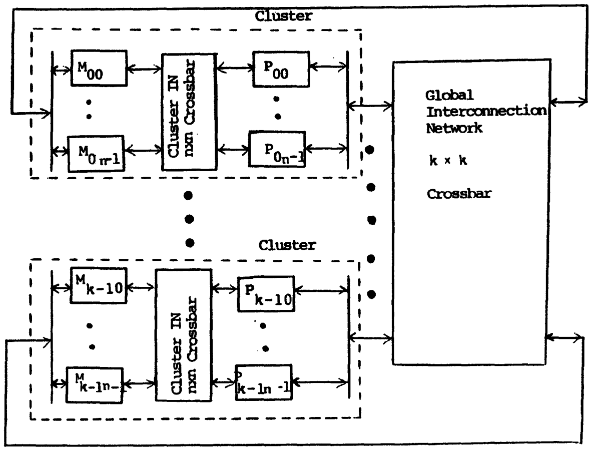


Fig.2. Cluster-Based Multiprocessor System

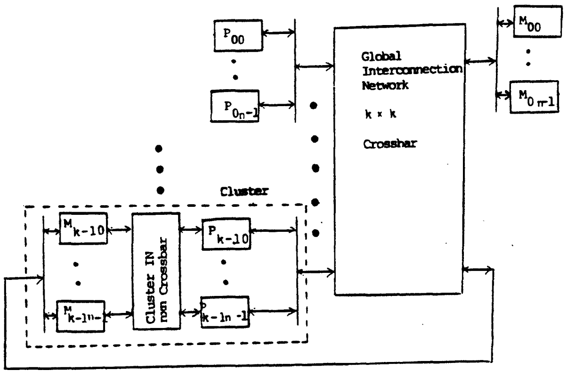


Fig.3 Cluster-based Multiprocessor System with a Failure Cluster IN

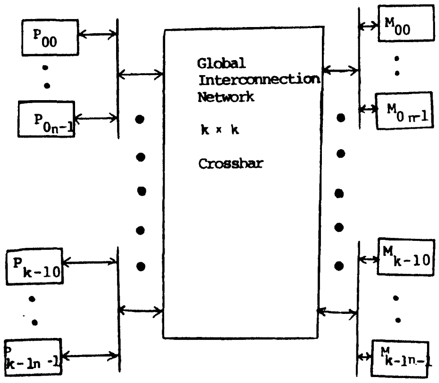


Fig.4 the Reconfiguration of Cluster-based Multiprocessor System Following the Failure of All Cluster INs

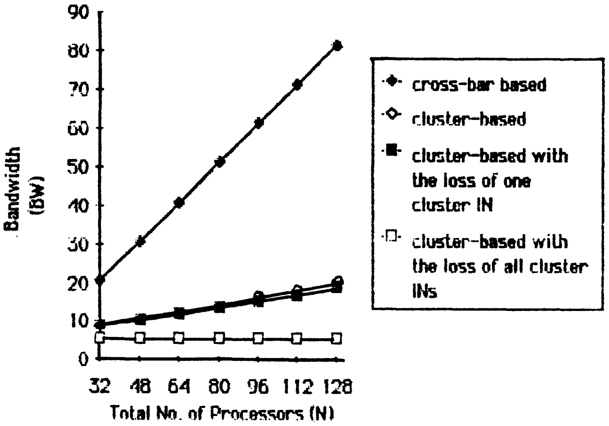


Fig. 5 System Performance for the Equally likely Case

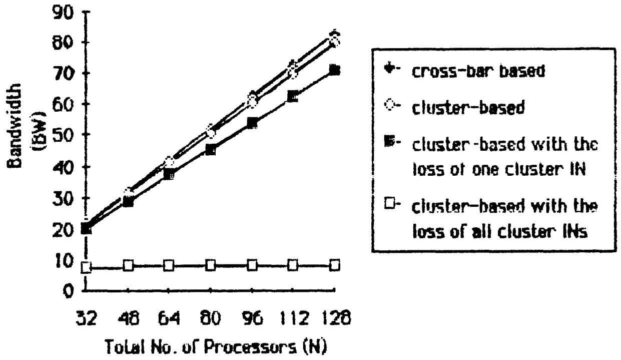


Fig.6 System Performance for the Favorite Memories case

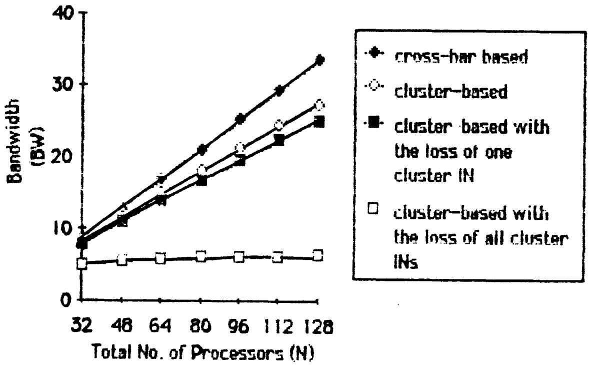


Fig.7 System Performance for the Favorite and Private Memories case

# RESOURCE SHARING INTERCONNECTION NETWORKS IN MULTIPROCESSORS

*Jie-Yong Juang and Benjamin W. Wah*[*]

## ABSTRACT

In this paper circuit-switched interconnection networks for resource sharing in multiprocessors, named *resource sharing interconnection networks*, are studied. Resource scheduling in systems with such an interconnection network entails the efficient search of a mapping from requesting processors to free resources such that circuit blockages in the network are minimized and resources are maximally used. The optimal mapping is obtained by transforming the scheduling problems into various network-flow problems for which existing algorithms can be applied. A distributed architecture to realize a maximum-flow algorithm using token propagations is also described. The proposed method is applicable to any general network configuration modeled as a digraph in which the requesting processors and free resources can be partitioned into two disjoint subsets.

## 1. INTRODUCTION

In this paper we investigate the problem on the sharing of computing resources in multiprocessors and the distributed scheduling of shared resources in a circuit-switched interconnection network.

A *resource* is a processing element to carry out a designated function. Examples include a general purpose processor, a special functional unit, a VLSI systolic array, an input/output device, and a communication channel. A resource is accessible by any processor via an interconnection network. A *request* generated by a processor can be directed to any one of a pool of free resources that are capable of executing the designated task. An interconnection network is an essential element of these systems as it interconnects processors and resources. Its function is to route requests initiated from one point to another point connected on the network. The network topology is dynamic, and the links can be reconfigured by setting the network's active switching elements. The notable characteristic of these networks is that they operate with address mapping. That is, a request is initiated with a specific destination or a set of destinations, and routing is done by examining the address bits. Routing of requests is usually done in parallel. As classified by Feng [8], these networks include the single or multistage networks and the crossbar switch. Examples are the banyan, indirect binary n-cube, cube, perfect shuffle, flip, Omega, data manipulator, augmented data manipulator, delta, baseline, Benes, and Clos. Examples of systems designed with interconnection networks are Trac, Staran, C.mmp, Illiac IV, Pluribus, Numerical Aerodynamic Simulation Facility (NASF), the Ballistic Missile Defense testbed, MPP, and Connection Machine. The performance of resource sharing systems under address mapping has been studied by Rathi, Tripathi and Lipovski [21], Fung and Torng [10], and Marsan, et al [17].

Wah proposed a network with distributed scheduling intelligence, called *resource sharing interconnection network* (*RSIN*) [23, 22]. Instead of using an address-mapping scheme, which requires a centralized scheduler to seek and give the address of a free resource to a request before it enters the network, the request is sent into the network without any destination tags. It is the responsibility of the network to route the maximum number of requests to the free resources. In this way the scheduling intelligence is distributed in the network. Distributed resource scheduling avoids the bottleneck of a centralized scheduler [21]. The objective of a good scheduling scheme is to avoid network blockages and to maximize resource utilization, which requires an efficient algorithm at each switching node to collect the minimum amount of status information.

The PUMPS architecture (see Figure 1) for image analysis and pictorial database management [3] is a typical example of resource sharing multiprocessors in which VLSI systolic arrays, each realizing an image processing function, are organized into a pool of resources. Most dataflow architectures can also be considered as resource sharing systems. The processing units are the pool of resources, and a RSIN connects them to memory cells. In a resource sharing system with load balancing, processors are considered as resources, thus requests generated are queued at the processors as well as the resources. An imbalance of workload at the processors will be balanced by a load balancing scheme.

The design of a RSIN with optimal resource scheduling is studied in this paper. The results are derived with respect to multistage interconnection networks, called *multistage resource sharing interconnection networks* (*MRSIN*), and are applicable to any general network configuration modeled as a digraph in which the requesting processors and free resources are partitioned into two disjoint subsets. Central to the design of such an

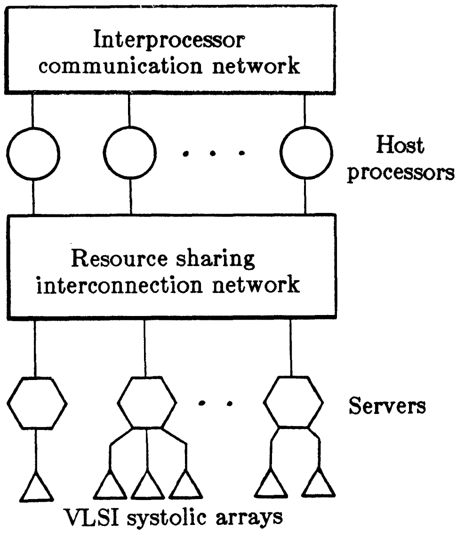

Interprocessor communication network
Host processors
Resource sharing interconnection network
Servers
VLSI systolic arrays

Figure 1. PUMPS: An example of a multiprocessor with shared systolic arrays.

[*] This research was supported by National Science Foundation Grant DMC 85-19649.

J. Y. Juang is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. B. W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

1986 International Conference on Parallel Processing.

0190-3918/86/0000/0750 $01.00 © 1986 IEEE

interconnection network is the development of an efficient distributed algorithm to disseminate status information through the complex interconnection structure. The algorithm to be presented is simple, efficient, and independent of the interconnection topology.

## 2. RESOURCE SHARING INTERCONNECTION NETWORKS

The assumptions of the RSIN used in this study are summaried as follows.

(a) Circuit switching is assumed rather than packet switching for the following reasons. First, packet switching is used in conventional networks with address mapping because it allows a network path to be shared by more than one request concurrently. Reducing the packet delay by balancing loads among alternate paths is less critical in a RSIN because a request can always search for another available resource if a path is blocked. Moreover, the overhead of rerouting a packet is higher than that of rerouting a resource request [16]. Second, owing to the resource characteristics, a task cannot be processed until it is completely received. The extra delay in breaking a task into multiple packets may decrease the utilization of resources, and hence increase the response time of the system.

(b) One or more types of resources may exist in the system. A RSIN connecting only one type of resources is called a *homogeneous* RSIN, while a RSIN connecting multiple types of resources is a *heterogeneous* one.

(c) A priority level may be associated with a request to show the urgency of the request. A preference value may be associated with a resource to show the desirability of being used for service. The costs of allocation are inversely related to the priorities and preferences.

(d) Each request needs one resource only.

(e) A processor can transmit one task at a time to the resources. Other tasks arriving during the task transmission time are queued. The circuit between a processor and a resource can be released once the request has been transmitted. The processor can continue to make other requests, while the resource will be busy until the task is completed.

We have not investigated the problem on the selection of the number of resources in each type and their placements in the output ports. This problem has been studied by Briggs et al., who have considered the problem of choosing the number of resources in each type in which one resource is connected to each output port and one resource is requested each time [2]. We have not considered the case in which more than one resource or multiple types of resources are requested by one request. Here, the scheduling algorithm is dependent on the number of resources in each type, the way that resources are distributed to the output ports, and the network characteristics. Further, deadlocks may occur, and distributed resolution of deadlock may have a high overhead.

The goal of the scheduling algorithm is to find a request-resource mapping such that the total cost is minimized. In the special case in which all requests are of equal priorities and all resources have equal preferences, the scheduling problem becomes the mapping of the maximum number of requests to the free resources.

The maximal request-resource mapping may be hampered by blockages in the system. In a conventional address-mapped interconnection network, blockages may be caused by conflicts in either the same resource being requested by more than one request or a network link requested by two circuits. In a RSIN, a resource conflict can be resolved by rerouting all but one request to other free resources. However, this may not always lead to better resource utilization because the allocation of one request to a resource may block one or more other requests from accessing free resources. A scheduling algorithm that schedules requests according to the state of the network and resources is, therefore, essential. As an example, consider an 8-by-8 Omega network** in Figure 2a with switch-boxes that can be individually set to either a straight or an exchange connection. Processors $p_1$, $p_3$, $p_5$, $p_7$ and $p_8$ are requesting one resource each, and resources $r_1$, $r_3$, $r_5$, $r_7$ and $r_8$ are available. The circuits between $p_2$ and $r_6$ and $p_4$ and $r_4$ have been established previously. $p_6$ is not making request, and $r_2$ is busy. All free resources will be allocated if one of the following request-resource mappings is used: $\{(p_1, r_3), (p_3, r_5), (p_5, r_7), (p_7, r_1), (p_8, r_8)\}$ or $\{(p_1, r_3), (p_3, r_8), (p_5, r_7), (p_7, r_1), (p_8, r_5)\}$. But if the mapping $\{(p_1, r_1), (p_3, r_5), (p_5, r_3), (p_7, r_7), (p_8, r_8)\}$ is used, a maximum of four out of five resources can be allocated, since the path from $p_8$ to $r_8$ is blocked. Simulation results showed that the average blocking probability can be as low as 2% for a MRSIN embedded in an 8-by-8 cube network [22, 12]. If a heuristic routing algorithm is used, the average blocking probability increases to 20%. Further degradation occurs if the network is not completely free.

## 3. OPTIMAL RESOURCE SCHEDULING IN MRSIN

To bind a request to a resource in the system, a RSIN determines a mapping from pending requests to free resources, and provides connections to as many request-resource pairs as possible. In this section, methods to optimize request-resource mappings are discussed. Exhaustive methods that examine all possible ordered mappings have exponential complexity. In a homogeneous MRSIN, suppose x processors are making requests, y

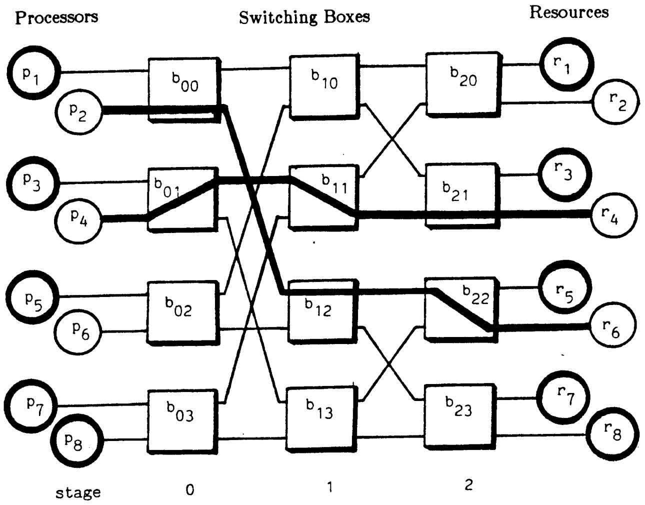


Figure 2a. A MRSIN embedded in an 8-by-8 Omega network (dark paths in the network show circuits that are already occupied; processors $p_1$, $p_3$, $p_5$, $p_7$ and $p_8$ are making requests; resources $r_1$, $r_3$, $r_5$, $r_7$ and $r_8$ are available).

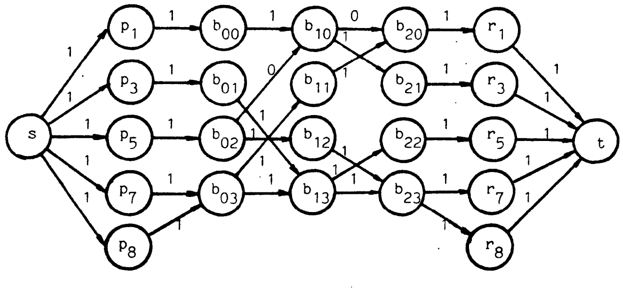


Figure 2b. The flow network transformed from the MRSIN in Figure 2a using Transformation 1 (the number associated with each arc is the amount of flow assigned to it by the maximum-flow algorithm; all arcs have unit capacity).

resources are available, and the network is completely free. The scheduler has to try a maximum of $C_y^x \bullet y!$ (for $x \geqslant y$) or $C_x^y \bullet x!$ (for $y \geqslant x$) mappings to find the best one, where $C_i^j$ is the number of combinations of choosing i objects out of j objects [22, 12]. Suboptimal heuristics can be used but is only practical when x and y are small.

In this section we transform the optimal request-resource mapping problem into various network-flow problems for which many efficient algorithms exists [11, 13]. The basic concepts of flow networks are briefly reviewed first.

### 3.1. Flow Networks

A flow network is usually represented by a digraph in which each arc is associated with a capacity and possibly a cost. Let $D = (V, E)$ be a digraph with two distinct nodes s (*source*) and t (*sink*). A capacity function, c(e), is defined on every arc of the graph, where c(e) is a non-negative real number for all $e \in E$. A flow function $f$ assigns a real number $f(e)$ to arc e such that the following conditions hold.

(1) *Capacity limitation*:

$$0 \leqslant f(e) \leqslant c(e) \quad \text{for every arc } e \in E \tag{1}$$

(2) *Flow conservation*: Let $\alpha(v)$ (resp. $\beta(v)$) be the set of incoming (resp. outgoing) arcs of vertex v. For every $v \in V$,

$$\sum_{e \in \alpha(v)} f(e) - \sum_{e \in \beta(v)} f(e) = \begin{cases} -F & v = s \\ F & v = t \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The capacity constraint restricts the amount of flow that can be assigned to a link, and flow conservation implies that an intermediate node in the network does not absorb or create flows.

A *legal flow* is a flow assignment that satisfies the capacity and flow-conservation constraints. In a network flow problem, it is necessary to find the legal flow that optimizes a given objective function. For example, in the maximum-flow problem, it is necessary to find the maximum amount of flow that can be advanced from source to sink in a flow network $G(V, E, s, t, c)$ under the capacity and flow-conservation constraints. The problem can be formulated as a linear program.

*Maximum-Flow Problem*

Maximize F
subject to:
(1) Flow conservation (Eq. (1));
(2) Capacity limitation (Eq. (2)).

Many other examples of network flow problems, including the minimum-cost flow and the trans-shipment problems, can be found in the literature [11].

An *s-t path* is a directed path from s to t. Insertion of a dummy node in a path will increase the path length but will not affect the flow assignment. Increasing the length of s-t paths in this way such that all s-t paths are of equal length is called *s-t path equalization* in this paper. All s-t paths are assumed to be equalized when the network is loop-free.

### 3.2. Optimal Resource Mapping in Homogeneous MRSIN

A switch-box in a MRSIN is a crossbar switch without broadcast connections. The following theorem shows that the setting of a non-broadcasting switch is equivalent to a legal integral flow assignment in a flow network of unit capacity. Note that an integral flow is a flow assignment in which the amount of flow assigned to each link is of integral value.

**Theorem 1*****: For any MRSIN, there exists a flow network for which a legal integral flow is equivalent to a valid request-resource mapping.

To use existing algorithms to solve a flow problem, a MRSIN has to be transformed into a flow network such that the optimization of request-resource mappings is equivalent to the optimization of the corresponding objective function in the flow network. To this end, additional nodes may be introduced, the capacity of a link may be greater than one, and a cost may be associated with a link.

The following transformation produces a flow network such that the optimal request-resource mapping can be derived from its maximum flow.

**Transformation 1**: Generate a flow network $G(V, E, s, t, c)$ from a homogeneous MRSIN.

(T1) Create three node-sets **P**, **X** and **R** for processors, switch-boxes and resources, respectively. Introduce two additional nodes: source s and sink t. Let

$$V' = \{s, t\} \cup P \cup X \cup R$$

(T2) Add an arc from the source to every node associated with a processor. Denote this set of arcs by **S**.

$$S = \{ (s,v) \mid v \in P \}$$

Add an arc between every node associated with a resource and the sink. This set of arcs is called **T**.

$$T = \{ (v,t) \mid v \in R \}$$

For each link in the MRSIN that connects two switch-boxes, or a processor to a switch-box, or a switch-box to a resource, add an arc between the corresponding nodes in the flow graph. Denote this set of arcs by **B**.

$$B = \{ (v,w) \mid v \in P \cup X, w \in X \cup R \}$$

Define $E' = S \cup T \cup B$

(T3) Assign link capacities according to the following function.

$$\underset{e \in B}{c(e)} = \begin{cases} 0 & \text{associated link is occupied} \\ & \quad \text{or non-existent in the MRSIN} \\ 1 & \text{associated link is free} \end{cases}$$

$$\underset{e \in S}{c(e)} = \begin{cases} 0 & \text{associated processor does not generate request} \\ 1 & \text{associated processor generates request} \end{cases}$$

$$\underset{e \in T}{c(e)} = \begin{cases} 0 & \text{associated resource is unavailable} \\ 1 & \text{associated resource is available} \end{cases}$$

(T4) Obtain arc-set **E** by removing those arcs with zero capacity.

$$E = E' - \{e \mid e \in E', c(e) = 0\}$$

Obtain node-set **V** by deleting those nodes that are not reachable from s.

$$V = V' - \{v \mid \alpha(v) \cup \beta(v) = \emptyset, v \in V'\}$$

Applying the above transformation to the MRSIN in Figure 2a results in the flow network in Figure 2b. The following theorem shows that Transformation 1 can be used to find the optimal request-resource mapping.

**Theorem 2**: In a homogeneous MRSIN, the number of resources allocated by a mapping is equal to the amount of flow that can be advanced from the source to the sink in the flow network obtained by Transformation 1.

From Theorem 2 and a known result that the maximum flow of a network with integral capacity is integral [11], we conclude that the optimal mapping can be derived from the maximum flow in the transformed flow network.

Many algorithms have been developed to obtain the maximum flow in a flow network. The algorithm by Ford and Fulkerson [9] is a primal-dual algorithm in which the flow value is increased by iteratively searching for *flow augmenting paths* until the minimum cut-set of the network is saturated. At this

** The input ports are numbered in a way different from Lawrie's Omega network [15] because all resources are homogeneous, and the permutation of requesting processors will not affect the resource utilization. Broadcast connection is not needed in the switch-boxes since each request needs one resource.

*** All theorems are stated without proof due to space limitation.

point, no more flow can be advanced since the minimum cut-set is the bottleneck. A flow augmenting path is an s-t path through which additional flow can be advanced from the source to the sink. When arc e on the s-t path points in the direction as the s-t path, additional flow may be advanced through e if the current flow assigned to e is less than c(e). In contrast, if arc e points in the opposite direction, then additional flow may be pushed through the s-t path by canceling its current flow. Advancing flow through an augmenting path in this way will always increase the total amount of flow, and the flow-conservation and capacity limitations will not be violated. For example, in Figure 3, an original flow $f$ is assigned along the path s-a-d-t. Then path s-c-d-a-b-t is a flow augmenting path. Advancing one unit of flow through this augmenting path results in a new flow assignment $f$ '. Two units of flow are pushed through two separate paths s-a-b-t and s-c-d-t according to this assignment.

In the MRSIN, advancing flow through an augmenting path is equivalent to a *resource re-allocation*, i.e., a permutation of the possible request-resource mappings. Consider the MRSIN in Figure 4, which is the counterpart of the flow network in Figure 3.[****] The original flow $f$ is equivalent to the request-resource mapping $\{(p_a, r_d), (p_c, r_b)\}$. The allocation of resource $r_b$ to request $p_c$ is blocked according to this mapping. The existence of the flow augmenting path s-c-d-a-b-t shows that this blockage can be removed. Advancing flow through this augmenting path results in a new mapping $\{(p_a, r_b), (p_c, r_d)\}$ and the allocation of both resources. As another example, applying the maximum-flow algorithm to the flow network in Figure 2b, the flow assignment as shown in the figure is obtained, and the request-resource mapping $\{(p_1, r_3), (p_3, r_5), (p_5, r_8), (p_7, r_1), (p_8, r_7)\}$ is derived. Note that the optimal mapping may not be unique, but suboptimal mappings are eliminated.

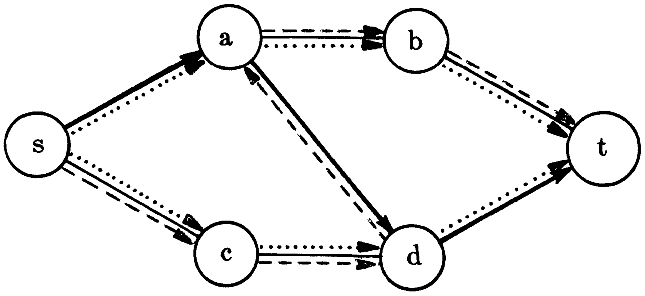


Figure 3. An illustration of advancing flow through a flow augmenting path. (All arcs have unit capacity. The initial flow is assigned to path s-a-d-t. The flow augmenting path s-c-d-a-b-t is indicated as dashed lines. The final flow assignment is obtained after advancing a unit of flow through the flow augmenting path and is indicated as dotted lines.)

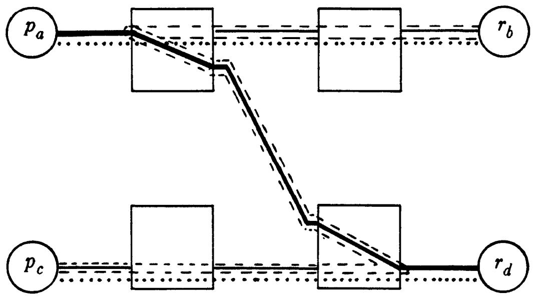


Figure 4. Resource re-allocation corresponding to flow augmentation in Figure 3. (Initially, one resource is allocated—dark lines. The flow augmenting path is indicated as dotted lines. Two resources are allocated after re-allocation—dotted lines.)

Finding a flow augmenting path from the source to the sink in a flow network is the central idea in most maximum-flow algorithms. The improvement lies in the efficient search of the flow augmenting paths [5, 4]. For example, in Dinic's algorithm, the shortest augmenting path is always advanced first with the aid of an auxiliary layered network, and hence the computational complexity is bounded by $O(|E|^3)$ for general networks. In our case, the links have unit capacity, and the time complexity is reduced to $O(|V|^{2/3} \cdot |E|)$ [11].

### 3.3. Homogeneous MRSIN with Request Priority and Resource Preference

In a homogeneous MRSIN with request priority and resource preference, each request is associated with a priority level, and each resource is assigned a preference value. Many application-dependent attributes, such as workload, execution speed, utilization and capability, can be encoded into request priorities and resource preferences. The objective of resource scheduling here is to maximize the number of resources allocated, while allowing requests of higher priority to be allocated and resources of higher preference to be chosen. However, it is not necessary for requests and resources to be allocated in order of their priorities and preferences. The allocation of a resource to a request may be blocked by requests of higher priority, and the resource may be allocated to a request of lower priority. A similar argument applies to resources.

With respect to a flow network, the request priority can be considered as the cost of carrying a flow through the path associated with this request. The resource preference can be considered similarly. As a result, the request-resource mapping problem in this class of MRSIN can be transformed into finding a flow assignment to minimize the total cost of flows in the network.

Consider a flow network $\mathbf{G}(\mathbf{V}, \mathbf{E}, s, t, c, w)$ in which w(e), the cost per unit flow, is associated with arc $e \in \mathbf{E}$. In the minimum-cost flow problem, a legal s-t flow assignment is sought that allows a given amount of flow F to be circulated from source to sink with the minimum cost. The objective is to determine the set of least expensive s-t paths through which the fixed flow F can be advanced. The constraints in this problem are the same as those in the maximum-flow problem. The problem may be defined in a linear programming formulation.

*Minimum-Cost Flow Problem*

Minimize $\sum_{e \in \mathbf{E}} w(e) f(e)$

subject to:

(1) Flow conservation (Eq. (1));
(2) Capacity limitation (Eq. (2)).

In allocating resources, the objective is to find a corresponding flow network whose optimal flow leads to an optimal request-resource mapping. The main idea behind the transformation is to embed priority and preference information into the objective function by proper cost assignments on links. The amount of flow to be circulated can be considered as the number of requests pending for allocation. However, this amount may exceed the capacity of the flow network or the number of available resources, and additional paths have to be introduced to prevent overflow. A possible transformation is given as follows.

**Transformation 2**: Generate a flow network $\mathbf{G}(\mathbf{V}, \mathbf{E}, s, t, c, w)$ from a homogeneous MRSIN with request priorities and resource preferences.

(T1) Create node-sets **P**, **X** and **R** for processors, switch-boxes and resources, respectively, and introduce special nodes: source, s, sink, t, and a *bypass node*, u. Let

$$\mathbf{V}' = \{s, t, u\} \cup \mathbf{P} \cup \mathbf{X} \cup \mathbf{R}$$

[****] The switch-boxes are combined with the processors or resources in the flow network in Figure 3, but will not affect the discussion.

(T2) Create arc-sets **S**, **T** and **B** as in Step (T2) of Transformation 1. Add an arc from the node associated with a processor to the bypass node, and connect the bypass node to the sink. This set of arcs is denoted as **L**.

$$\mathbf{L} = \{(v, u) \mid v \in \mathbf{P}\} \cup \{(u, t)\}$$

Define $\mathbf{E}' = \mathbf{S} \cup \mathbf{T} \cup \mathbf{B} \cup \mathbf{L}$

(T3) Define capacity function c as in Step (T3) of Transformation 1. In addition, define

$$\underset{e\in\mathbf{L}}{c(e)} = \begin{cases} 1 & e\neq(u,t) \\ \alpha(u) & e=(u,t) \end{cases}$$

(T4) Define cost function w that represents the cost of advancing one unit of flow through a link as follows.

$$w(e) = \begin{cases} 0 & \text{for } e\in\mathbf{B} \\ \max(y_{max}+1, q_{max}+1) & \text{for } e\in\mathbf{L} \\ y_{max}-y_p & \text{for } e\in\mathbf{S},\ p\in\mathbf{P} \\ q_{max}-q_w & \text{for } e\in\mathbf{T},\ w\in\mathbf{R} \end{cases}$$

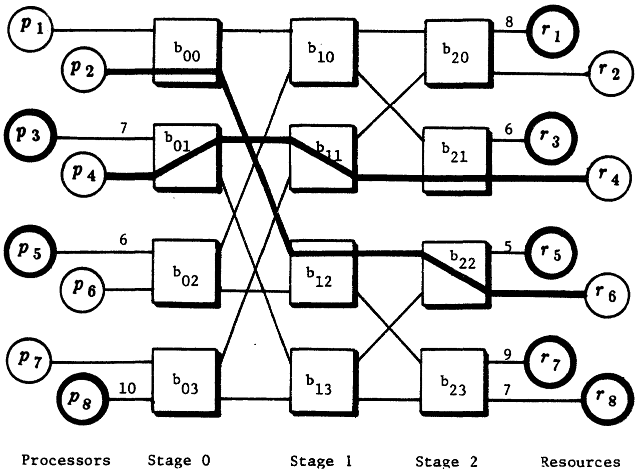


Figure 5a. A MRSIN with request priority and resource preference (highest priority is 10; highest preference is 10; dark paths in the network are already occupied; processors $p_3$, $p_5$ and $p_8$ are making requests; resources $r_1$, $r_3$, $r_5$, $r_7$ and $r_8$ are available).

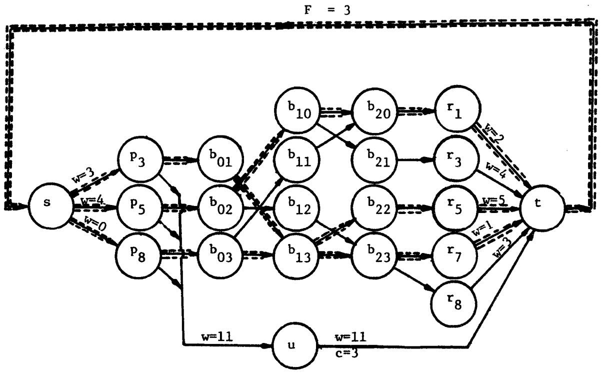


Figure 5b. The flow network transformed from the MRSIN in Figure 5a using Transformation 2 (non-zero flows assigned by the out-of-kilter algorithm are shown as dashed lines in the figure; return flow from t to s is added by the out-of-kilter algorithm; all arcs have unit capacity; cost of arc is zero except where indicated).

where $y_{max}$ is the highest priority level, $y_p$ is the priority of request from processor p, $q_{max}$ is the highest preference level, and $q_w$ is the preference of resource w. Note that any cost function that is inversely related to priorities and preferences can be used.

(T5) Create arc-set **E** and node-set **V** as in Step (T4) of Transformation 1.

(T6) Set the total flow F to the number of requests.

As an example, in the MRSIN in Figure 5a, each request is attributed a priority level, and an available resource is given a preference value. The preference and priority levels range from 1 to 10. A minimum-cost flow network obtained from Transformation 2 is shown in Figure 5b. The following theorem shows the correctness of Transformation 2.

**Theorem 3:** The optimal request-resource mapping on a homogeneous MRSIN with request priority and resource preference can be derived from the minimum-cost integral flow of the flow network, which is obtained by Transformation 2.

Edmonds and Karp have developed a scaled out-of-kilter algorithm to obtain the minimum-cost flow of a general flow network in polynominal time [5]. For a flow network of 0-1 capacity, the time complexity is bounded by $O(|V|\cdot|E|^2)$. Furthermore, in the minimum-cost flow assignment obtained, the flow assigned to a link is integral if the links have integral capacities. As an example, applying the minimum-cost flow algorithm on the flow network in Figure 5b results in the request-resource mapping $\{(p_3, r_5), (p_5, r_1), (p_8, r_7)\}$. The selected paths are shown as dashed lines in Figure 5b. Note that the minimum-cost flow obtained may not be unique, and alternate minimum-cost flows are possible. However, alternative mappings will not improve the cost of allocation.

### 3.4. Optimal Resource Scheduling in Heterogeneous MRSIN

A heterogeneous MRSIN consists of multiple types of resources, and a processor may generate a request of a given type of resource. Such a MRSIN is equivalent to a flow network carrying different types of commodities. A multicommodity flow network has multiple source-sink pairs, each of which is associated with one type of commodity. A flow coming out of a source of a given commodity can only be absorbed by the sink of the same type of commodity. Flows of different commodities may share a link as long as the total flow does not exceed the capacity of the link.

For a flow network with k types of commodities, there are k source-sink pairs, $(s^i, t^i)$, for i=1 to k. Let $F^i$ be the total flow of the i'th commodity and $f^i(e)$ be the flow of the i'th commodity on edge e. The search for the maximum flow can be formulated as follows [1].

*Multicommodity Maximum-Flow Problem*

Maximize $\sum_{i=1}^{k} F^i$

subject to:

(1) Flow conservation: For i = 1, ...., k

$$\sum_{e\in\alpha(v)} f^i(e) - \sum_{e\in\beta(v)} f^i(e) = \begin{cases} -F^i & v = s^i \\ F^i & v = t^i \\ 0 & \text{otherwise} \end{cases}$$

(2) Capacity limitation:

$$0 \leqslant \sum_{i=1}^{k} f^i(e) \leqslant c(e) \quad \text{for all } e\in\mathbf{E}$$

A multicommodity flow network may be visualized as the superposition of k single-commodity flow networks. Each layer in the superposition represents a single-commodity flow. To obtain the optimal request-resource mapping in a heterogeneous MRSIN without priority and preference, a transformation similar to Transformation 1 can be applied to obtain a layer for each type of resources, and the layers are superposed to form a multicommodity flow network.

The optimal mapping for a heterogeneous MRSIN with request priorities and resource preferences can be obtained by transforming the problem into the multicommodity minimum-cost flow problem. Let $w^i(e)$ be the cost per unit flow for the i'th commodity on edge e. The problem can be formulated as follows.

*Multicommodity Minimum-Cost Flow Problem*

$$\text{Minimize } \sum_{i=1}^{k} \sum_{e\in E} w^i(e)\, f^i(e)$$

subject to:

(1) Flow conservation: For i = 1, ..., k

$$\sum_{e\in\alpha(v)} f^i(e) - \sum_{e\in\beta(v)} f^i(e) = \begin{cases} -F^i & v = s^i \\ F^i & v = t^i \\ 0 & \text{otherwise} \end{cases}$$

(2) Capacity limitation:

$$0 \leq \sum_{i=1}^{k} f^i(e) \leq c(e) \quad \text{for all } e\in E$$

The equivalent flow network consists of k source-sink pairs and k bypass nodes, where k is the number of types of requested resources. As for requests without priority, the flow network can be regarded as the superposition of k single-commodity flow networks, and Transformation 2 can be applied to each of them.

The problem of finding the maximum integral flow in a multicommodity flow network of general topology has been shown to be NP-hard. Fortunately, circuit switched loop-free interconnection networks have transformations that belong to a restricted class of multicommodity flow networks in which the optimal flow values are always integral [6]. For this class of flow networks, the integral multicommodity optimal flows can be obtained efficiently by the *Simplex Method*, which has been shown empirically to be a linear-time algorithm [18].

## 4. ARCHITECTURE OF MRSIN TO SUPPORT OPTIMAL SCHEDULING

Two architectures to carry out the optimal resource scheduling algorithms have been studied. In the first approach, a dedicated monitor is responsible for resource scheduling (see Figure 6). It maintains the status of the interconnection network and resources. The monitor enters a scheduling cycle when there are pending requests. Requests received during a scheduling cycle will not be processed until the next cycle. In a scheduling cycle, a flow network is generated, and the optimal request-resource mapping is derived. Then the monitor sends an acknowledgement to each requesting processor that has been allocated a resource, notifies resources that are allocated, and establishes paths in the network. The implementation is sequential, and the overhead is measured by the number of instructions executed in the algorithm.

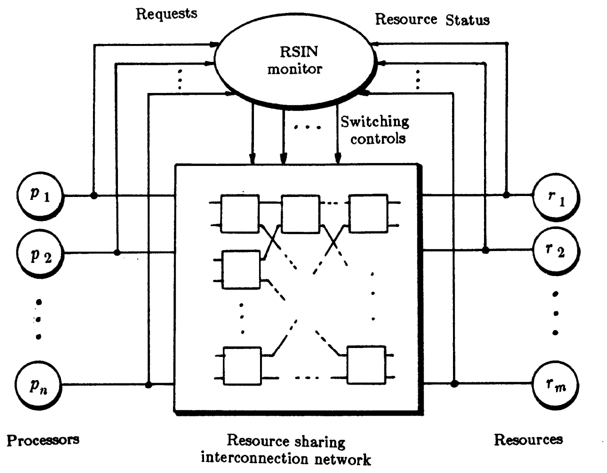


Figure 6. A monitor architecture to carry out optimal resource scheduling in a RSIN.

A distributed architecture, on the other hand, distributes the scheduling intelligence in the switch-boxes of the interconnection network. Optimal scheduling is achieved through cooperations among processes in the switch-boxes. No transformation to a network-flow problem is necessary because the network-flow algorithm is carried out in a distributed fashion in the switch-boxes. The complexity of the process in each switch-box is central to the design of the distributed architecture. Our previous study shows that the maximum-flow algorithm for homogeneous MRSIN without priority and preference can be efficiently implemented in a distributed fashion [14]. For systems with heterogeneous resources or with priorities and preferences, there is no improvement over a monitor except for reasons such as fault tolerance and modularity.

In the following sections, we describe a distributed realization of Dinic's maximum-flow algorithm to obtain the optimal request-resource mapping.

### 4.1. Dinic's Maximum-Flow Algorithm

Dinic's algorithm is based on the flow augmentation method described in Section 3.2. It improves over Ford and Fulkerson's algorithm by advancing flow through the shortest augmenting path, which can be found from a *layered network* derived from the original flow graph. A procedure summarizing Dinic's algorithm is as follows.

```
procedure dinic (V, E);
/* The algorithm has two alternating phases. The algorithm
   alternates between these two phases until no more flow can
   be augmented. */
(1) Initialization: Flow network with an initial flow assignment.
(2) Layered Network Construction Phase:
      (2a) Include s in the first layer;
      (2b) Construct the next layer;
      (2c) If layer is empty, then go to Step (4);
      (2d) If t is not in this layer, then go to Step (2b);
(3) Maximal-Flow Search Phase:
      /* Determine an increment to the flow assignment by finding
      the maximal flow in the layered network. */
      (3a) Search for s-t paths;
      (3b) If an s-t path does not exist, then go to Step (2a);
      (3c) Advance flow through this path; Go to Step (3a);
(4) Stop: Maximum flow assignment has been obtained.
```

In the layered network, nodes of the original flow network are organized into stages. The first stage consists of the source node(s) of the network, and the remaining stages are constructed iteratively. A stage consists of nodes that are not included in the previous stages and have either an unsaturated arc or an arc with non-zero flow originating from nodes in the previous stage. These two types of arcs, called *useful links*, are transformed to arcs in the layered network. Depending on the direction of the associated useful link, its capacity in the layered network can be either the remaining capacity or its current flow. As a result, nodes in a layered network are arranged into disjoint subsets, $V_0, \ldots, V_\ell$, such that no arc points from $V_j$ to $V_i$ ($i \leq j$).

A legal flow in a layered network is said to be maximal if every (s,t)-directed path in the layered network is saturated. Note that a maximal flow in a layered network is not necessarily the maximum flow because flows in opposite directions in the same arc are not considered. Moreover, computing the maximal flow is easier than computing the maximum flow. In Dinic's algorithm, the maximal flow is obtained by a depth-first search.

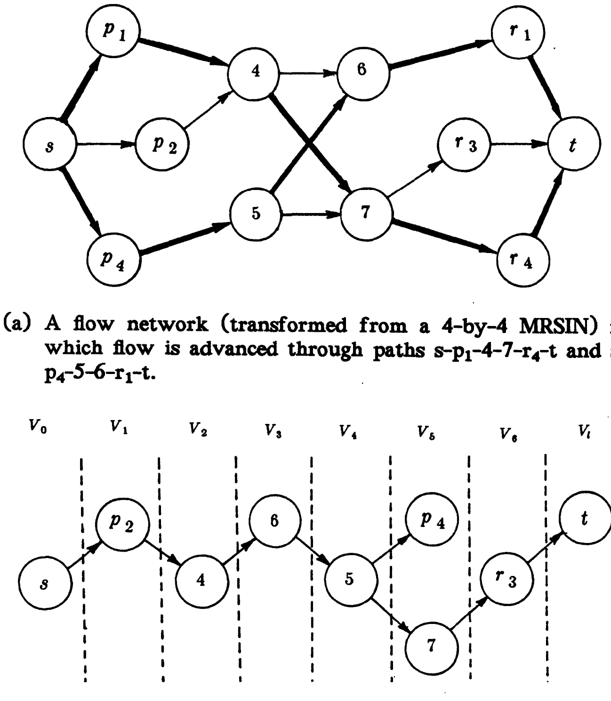
Since the amount of flow that can be advanced through an arc in the layered network is the net increase of flow to the associated arc in the original network, the maximal flow obtained in the layered network is a net increment to the existing flow. Further, the maximum flow of a flow network is finite. Hence, the maximum flow can be obtained in a finite number of iterations in constructing the layered network.

An example illustrating the construction of a layered network is shown in Figure 7. Figure 7a is a flow network associated with a MRSIN in which three processors, $p_1$, $p_2$ and $p_4$, are making requests and three resources, $r_1$, $r_3$ and $r_4$, are available. The flow assignment shown by darkened arcs in Figure 7a results in a mapping such that $p_1$ is mapped to $r_4$ and $p_4$ is mapped to $r_1$. The request generated by $p_2$ is blocked. Figure 7b is a layered network constructed from the flow network in Figure 7a. The layered network shows that there is a flow augmenting path from $p_2$ to $r_3$. This path includes the arc leading from node 6 to node 5, which is associated with the arc leading from node 5 to node 6 in the original network (see Figure 7a). This flow augmenting path shows that all three resources can be allocated if $p_4$ is re-allocated to $r_3$ and $p_2$ is re-allocated to $r_1$.

### 4.2. A Distributed Architecture for Homogeneous MRSIN without priority

A distributed MRSIN embedded in an 8-by-8 Omega network is shown in Figure 8. The scheduling intelligence is distributed in the switch-boxes (NS). A processor is connected to the network through a request server (RQ), and a resource is monitored by a resource server (RS). A common status bus connects these components together. Autonomous process in each component communicates with other processes by passing tokens via direct links, and are synchronized by exchanging status via the status bus.

A scheduling cycle begins when there are pending requests and ready resources. A request generated in the middle of a scheduling cycle has to wait until the next cycle. The procedure indicating the flow of phase transitions in a scheduling cycle is shown as follows.



(a) A flow network (transformed from a 4-by-4 MRSIN) in which flow is advanced through paths s-$p_1$-4-7-$r_4$-t and s-$p_4$-5-6-$r_1$-t.

(b) The layered network derived from the flow network in (a). The (s,t)-path s-$p_2$-4-6-5-7-$r_3$-t is equivalent to an augmenting path in the original flow network.

Figure 7. An illustration of a layered-network construction (all arcs have unit capacity).

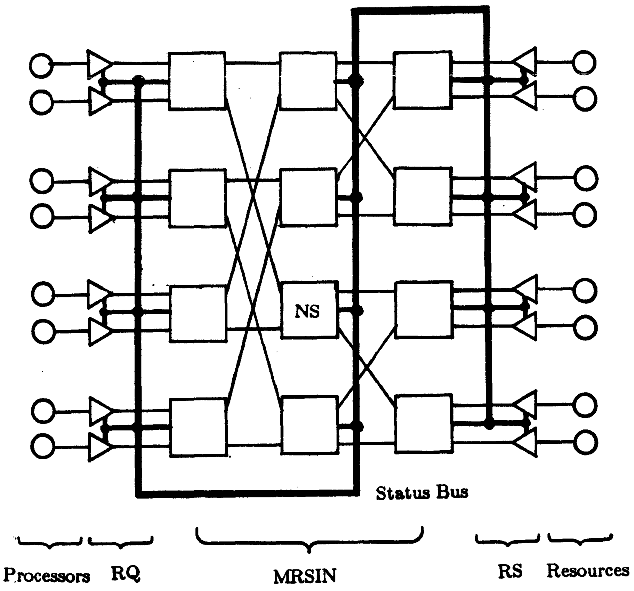


Figure 8. A distributed MRSIN embedded in an 8-by-8 Omega network.

```
procedure mrsin (P, X, R);
/* During a scheduling cycle, the network alternates between
   two phases until all request tokens are blocked. */
(1) Initialization: MRSIN with pending requests and ready
    resources;
(2) Request-Propagation Phase:
    /* In this phase, each requesting RQ sends a request token to
    the network, which will eventually arrive at the resources if
    there are free resources to allocate. */
    (2a) Requesting RQs send tokens;
    (2b) Propagate tokens to next NSs;
    (2c) If all request tokens are blocked, then go to Step (4);
    (2d) If request tokens are not received by the RSs associated
         with free resources, then go to Step (2b);
(3) Resource-Acknowledgment Phase:
    /* The RSs associated with free resources and receiving
    request tokens will send resource tokens to the network to
    acknowledge the acceptance of requests. When a resource
    token is received by a requesting RQ, the RQ and RS will
    form a matched pair, and the path connecting them is
    registered. The network returns to the request-propagation
    phase when all resource tokens are blocked. */
    (3a) Propagate resource tokens to RQs;
    (3b) If all resource tokens are blocked, then go to Step (2a);
    (3c) Register path; Go to Step (3a);
(4) Stop: Optimal request-resource mapping has been obtained.
```

Note that the flow of this procedure is exactly the same as the control flow of Dinic's algorithm.

To carry out Dinic's algorithm by token propagations, a switching element has to follow several token-propagation rules. If a link is free, a switch-box will deliver any request token received from the processor side to the resource side. These directions are reversed if the link is already registered. If there are more than one path to send the request token, the token is duplicated and sent along all paths. A resource token is expected from where a request token was delivered. However, no duplication of resource token is done, and one path is chosen at random. It can be verified that these token-propagation rules correctly implement the layered-network construction process and the search for the maximal flow.

To avoid chaos in token propagations, processes residing in different switch-boxes have to be synchronized on phase transitions. This synchronization can be achieved by broadcasting the status of each process on the status bus. The status of the network can be one of four independent possibilities: request-pending, resource-ready, request-token-propagation, and resource-token-propagation, and can be represented as a Boolean vector (RP, SR, RTP, STP). Each process will synchronize the broadcast of its status on the bus with other processes. The result is a wired-OR of the statuses broadcast by all processes. The occurrence of events that cause a phase transition can be detected by observing the status bus. For example, the transition from a request-propagation phase to a resource-acknowledgement phase is known to every process when each observes a change of the status vector from (1,1,1,0) to either (1,1,1,1) or (1,1,0,1).

We have shown that every process involved can be carried out by a simple sequential machine and is realizable in logic circuits [14]. With this design, there are two factors that contribute to a significant speedup as compared to a monitor architecture: (a) the augmenting paths are searched in parallel, and (b) the time complexity is measured in gate delays instead of instruction execution cycles. As a result, the scheduling algorithm will run at least 100 times faster than a software implementation of the network flow algorithm.

## 5. CONCLUSIONS

A RSIN is suitable to support resource sharing in multiprocessors. Optimal request-resource mapping in RSIN is obtained by maximizing the number of communication paths that interconnect pairs of processors and resources. In this paper, we have transformed various request-resource mapping problems into network flow problems for which efficient algorithms exist. Table 1 is a summary of the results we have obtained. The proposed method is independent of the interconnection structure and is applicable to all configurations in which the requesting processors and free resources are partitioned into two disjoint subsets. In particular, the method is applicable to networks with multiple paths between source-destination pairs, such as the data manipulator [7], the augmented data manipulator [19], and the Gamma network [20]. The resource utilization, however, will depend on the network configuration, the resources available, the permutation of the various types of resources, and the permutation of the requesting processors.

| *Scheduling Discipline* | | *Homogeneous Resources General Topology* | | *Heterogeneous Resources* | |
|---|---|---|---|---|---|
| | | *No Priority & Preference* | *Priority & Preference* | *Loop-free Topology* | *General Topology* |
| Optimal Scheduling | Equivalent Flow Problem | Max-Flow | Min-Cost Circulation | Real Multi-Commodity | Integer Multi-Commodity |
| | Algorithm | Ford-Fulkerson, Dinic | Out-of-Kilter | Linear Programming | NP-hard |
| Distributed Algorithm | | Yes | NA | NA | NA |
| Implementation | | Distributed Architecture | Monitor Architecture | | |
| | | Synchronized by Broadcasting | Software | | |
| | | Communicate by Token Propagation | | | |

Table 1. Summary of optimal resource scheduling schemes for resource sharing interconnection networks.

## REFERENCES

[1] A. A. Assad, "Multicommodity Network Flows—A Survey," *Networks*, vol. 8, pp. 37-91, 1978.

[2] F. A. Briggs, M. Dubios, and K. Hwang, "Throughput Analysis and Configuration Design of a Shared-Resource Multiprocessor System: PUMPS," *Proc. 8th Annual Symposium on Computer Architecture*, pp. 67-79, 1981.

[3] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS Architecture for Pattern Analysis and Image Database Management," *Trans. on Computers*, vol. C-31, no. 10, pp. 969-983., IEEE, Oct. 1982.

[4] E. A. Dinic, "Algorithm for Solution of a Problem of Maximal Flow in a Network with Power Estimation," *Soviet Math. Dokl.*, vol. 11, pp. 1277-1280, 1970.

[5] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *J. of the ACM*, vol. 19, no. 2, pp. 248-264, April 1972.

[6] J. R. Evans and J. J. Jarvis, "Network Topology and Integral Multicommodity Flow Problems," *Networks*, vol. 18, pp. 107-119, 1978.

[7] T. Y. Feng, "Data Manipulating Functions in Parallel Processors and Their Implications," *Trans. on Computers*, vol. C-23, no. 3, pp. 309-318, IEEE, March 1974.

[8] T. Y. Feng, "A Survey of Interconnection Networks," *Computer*, pp. 12-27, IEEE, Dec. 1981.

[9] L. R. Ford and D. R. Fulkerson, *Flow in Networks*, Princeton University Press, Princeton, NJ, 1962.

[10] F. Fung and H. Torng, "On the Analysis of Memory Conflicts and Bus Contentions in a Multiple-Microprocessor System," *Trans. on Computers*, vol. C-28, no. 1, pp. 28-37, IEEE, Jan. 1979.

[11] B. Golden, M. Ball, and L. Bodin, "Current and Future Research Directions in Network Optimization," *Computers and Operations Research*, vol. 8, pp. 71-81, 1981.

[12] A. Hicks, *Resource Scheduling on Interconnection Networks*, M.S. Thesis, Purdue University, West Lafayette, IN, Aug. 1982.

[13] J. Y. Juang and B. W. Wah, "Optimal Scheduling Algorithms for Multistage Resource Sharing Interconnection Networks," *Proc. 8'th Int'l Computer Software and Applications Conf.*, pp. 217-225, IEEE, Nov. 1984.

[14] J. Y. Juang, *Resource Allocation in Computer Networks*, Ph.D. Thesis, Purdue University, West Lafayette, IN, Aug. 1985.

[15] D. Lawrie, "Access and Alignment of Data in an Array Processor," *Trans. on Computers*, vol. C-24, no. 12, pp. 215-255, IEEE, Dec. 1975.

[16] M. Lee and C.-L. Wu, "Performance Analysis of Circuit Switching Baseline Interconnection Networks," *Proc. of the 11th Annual Int'l Symp. on Computer Architecture*, pp. 82-90, 1984.

[17] M. A. Marsan and M. Gerla, "Markov Models for Multiple Bus Multiprocessor Systems," *Trans. on Computers*, vol. C-31, no. 3, pp. 239-248, IEEE, March 1982.

[18] E. H. McCall, "Performance Results of the Simplex Algorithm for a Set of Real-Word Linear Programming Models," *Comm. of the ACM*, vol. 25, no. 3, pp. 207-213, March 1982.

[19] R. J. McMillen and H. J. Siegel, "Routing Schemes for the Augmented Data Manipulator Network in an MIMD System," *Trans. on Computers*, vol. C-31, no. 12, pp. 1202-1214, IEEE, Dec. 1982.

[20] D. S. Parker and C. S. Raghavendra, "The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths," *Proc 9th Annual Symposium on Computer Architecture*, pp. 73-80, 1982.

[21] B. D. Rathi, A. R. Tripathi, and G. J. Lipovski, "Hardwired Resource Allocators for Reconfigurable Architectures," *Proc. Int'l Conference on Parallel Processing*, pp. 109-117, IEEE, Aug. 1980.

[22] B. W. Wah and A. Hicks, "Distributed Scheduling of Resources on Interconnection Networks," *Proc. National Computer Conference*, pp. 697-709, AFIPS Press, 1982.

[23] B. W. Wah, "A Comparative Study of Distributed Resource Sharing on Multiprocessors," *Trans. on Computers*, vol. C-33, no. 8, pp. 700-711, IEEE, Aug. 1984.

# APPLICATION OF DATA DRIVEN NETWORKS TO SPARSE MATRIX MULTIPLICATION *)

Rami Melhem **)

Department of Mathematics
The University of Pittsburgh
Pittsburgh, PA. 15260.

## ABSTRACT

We consider linear, data-driven, arrays in which each cell is augmented with the capability of recognizing and skipping operations that involve zero operands. This type of networks is shown to be efficient for highly sparse matrices, despite the potential data conflict that may result due to irregular zero distributions. Two different approaches, which take advantage of the sparsity of the matrix, are considered; The first is aimed at the reduction of the execution time, and the second is aimed at the reduction of the number of cells in the network.

## 1. INTRODUCTION

Many techniques have been suggested for the efficient solution of sparse linear systems through clever, but highly irregular, storage and manipulation of the non-zero coefficients in the system [1]. These techniques, however, are not suitable for parallel processing, which requires, in general, a rather regular pattern of computation. In this paper, we consider a fundamental operation in iterative solution schemes for linear systems. Namely, the multiplication of a matrix by a vector.

The computation involved in the matrix/vector multiplication is quite regular, and thus appropriate for regular VLSI networks. That is systolic [2] and data-driven [3] arrays. Obviously, the benefits of this type of networks may become visible only when the size of the matrix is large. However, large systems that appear in practice are usually sparse, and hence seem inefficient for solution on regular VLSI networks due to the potential loss of resources in operations which involve zero operands.

In order to avoid the waste of resources caused by trivial operations, and, in the same time, retain the advantages of fast specialized cells and efficient local communications, we suggest to add to each cell in the network the capability of recognizing and skipping trivial operations. With data-driven synchronization, this results in a shorter average cycle for each cell, which may reduce the execution time of the entire network.

Two different approaches for the multiplication of sparse matrices by vectors are introduced. In the first approach, the same number of cells which would be needed if the matrix were dense is used, but a considerable speed-up is obtained by skipping trivial operations. In the second approach, the non-zero elements of the matrix are grouped in few stripes which are almost parallel to the diagonal of the matrix, and a specific cell is assigned to perform the operations associated with the elements of a particular stripe. This approach is aimed at the reduction of the number of cells without slowing down the speed of the computation.

## 2. COMPUTATIONAL CELLS WITH DATA DEPENDENT OPERATIONS

Consider a data driven version of the systolic network given in [2] for the multiplication of an $n \times n$ banded matrix $A$ by a vector $x$. For simplicity, we assume that the number of upper diagonals of $A$ is equal to the number of lower diagonals, namely $B_h$, and hence, the band-width of $A$ is $B = 2B_h + 1$. The network, called from now on $MV_1$, is shown in Fig. 1.

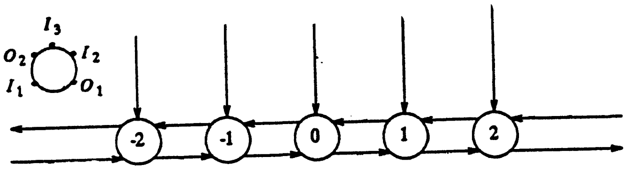


Fig. 1 - The network $MV_1$ with $B_h = 2$.

Every cell in $MV_1$ has three input ports and two output ports, namely $I_1, I_2, I_3, O_1$ and $O_2$, respectively. Its operation may be described by the following cycle which is repeated indefinitely (Here, $[I]$ denotes the content of port $I$, and $O \leftarrow \alpha$ means that $\alpha$ is written on port $O$).

*CYCLE 1:*

```
(1) Wait until data is available on $I_1, I_2$ and $I_3$ ;
      $\alpha = [I_1]$ ; $\beta = [I_2]$ ; $\gamma = [I_3]$
(2) $\rho = \alpha + \beta * \gamma$
(3) Wait until previous data on $O_1$ and $O_2$ is consumed ;
      $O_2 \leftarrow \beta$ ; $O_1 \leftarrow \rho$.
```

Each internal communication link directed from cell $q$ to cell $k$ may be regarded as a queue. Only cell $q$ may write on this queue and only cell $k$ may read (and delete) its front element. The maximum capacity of this queue will be denoted by $d$. In addition to internal links, the network has external links for communication with a host system. More specifically, the host supplies the elements of the $k^{th}$ diagonal of $A$, $-B_h \leqslant k \leqslant B_h$, on port $I_3$ of cell $k$, the elements of the vector x on port $I_2$ of cell $B_h$, and the elements of the result vector $y$ (initialized to zero) on port $I_1$ of cell $-B_h$. With this it is easy to see that the elements of the product vector $y = Ax$ are produced on port $O_1$ of cell $B_h$.

Now assuming that $\zeta\%$ of the elements in the band of $A$ are zeroes, then it is clear that $\zeta\%$ of the resources in $MV_1$ are wasted in the execution of trivial operations in step 2 of CYCLE 1. In order to reduce this waste, we may attempt to skip the multiply/add operation whenever $[I_3] = 0$. More specifically, consider a network $MV_2$, which is identical to $MV_1$ except that each cell executes the following cycle instead of CYCLE 1.

*CYCLE 2:*

```
(1) As in step 1 of CYCLE 1
(2) If ($\gamma \equiv 0$) Then 2.1) $\rho = \alpha$
                     Else 2.2) $\rho = \alpha + \beta * \gamma$
(3) As in step 3 of CYCLE 1
```

An execution of CYCLE 2 which goes through step 2.1 is called a trivial execution of the cycle, otherwise the execution is called non-trivial. Trivial executions of CYCLE 2 may, or may not, be shorter than non-trivial executions, depending on

*) This work is in part supported under ONR contract N00014-85-K-0339.

**) On leave from the Department of Computer Science, Purdue University.

0190-3918/86/0000/0758 $01.00 © 1986 IEEE

the waiting time spent in steps 1 and 3. Hence, the total execution time $T_2$ of $MV_2$ depends primarily on the effect of data conflict on the execution of individual cells.

One way of obtaining an upper bound on the execution time of data driven networks of the type discussed above is to force some hypothetical synchronization on the computation, such that execution alternates between two phases. Namely a communication phase and a processing phase. We call the resulting computation a pseudo systolic computation and we call a communication phase followed by a processing phase a global cycle. Given that the additional synchronization may only slow down execution, then it is clear that the execution time of a data driven computation is bounded by the execution time of the corresponding pseudo systolic computation.

For example, a pseudo systolic version of $MV_1$, called $MV_{1,p}$, may be obtained by replacing step 2 in CYCLE 1 with

```
(2) Wait for a synchronization signal SYNC
    $\rho = \alpha + \beta * \gamma$
```

Where we assume that all the cells in $MV_{1,p}$ are connected to a hypothetical controller that issues the signal SYNC after it detects the termination of a communication phase. That is when all the cells are waiting in step 2. With this it is easy to see that the execution time $T_1$ of $MV_1$ is bounded by the execution time $T_{1,p}$ of its pseudo systolic version, namely

$$T_1 \leqslant T_{1,p} = 2\,(B_h + n)\,(\tau_c + \tau_m)$$

where $\tau_m$ is the time for a floating point multiply/add operation (step 2) and $\tau_c$ is the time for the transmission of one data item between two cells.

Tracing the execution of $MV_2$ is more complicated than $MV_1$ because of possible trivial executions of CYCLE 2. A pseudo systolic version of $MV_2$, namely $MV_{2,p}$, may be obtained if CYCLE 2 is replaced by the following cycle:

```
CYCLE 2P: /* Pseudo Systolic Version of CYCLE 2 */
(1) Wait until data is available on $I_1$, $I_2$ and $I_3$ ;
    $\alpha = [I_1]$ ; $\beta = [I_2]$ ; $\gamma = [I_3]$
(2) If ($\gamma \equiv 0$) then
    2.1) wait until previous data on $O_1$ and $O_2$ are consumed
    2.2) $O_1 \leftarrow \alpha$ ; $O_2 \leftarrow \beta$
    2.3) Go To step 1
(3) Wait for a synchronization signal SYNC
(4) $\rho = \alpha + \beta * \gamma$
(5) Wait until previous data on $O_1$ and $O_2$ are consumed ;
    $O_1 \leftarrow \rho$ ; $O_2 \leftarrow \beta$.
```

In other words, trivial operations are first skipped until a non-trivial operand is found in $[I_3]$, then the multiplication is performed. As in $MV_{1,p}$ , we assume that all the cells are connected to a controller that issues the signal SYNC in such a way that execution alternates between communication and processing phases. During the communication phase, the data is moving in the network until each cell is either blocked due to lack of data (step 1, 2.1 or 5), or is blocked in step 3 (with $[I_3] \neq 0$). When this state is reached, the controller issues SYNC and all the cells which are blocked at step 3 executes the multiplication (step 4), simultaneously, while the other cells remain idle until the end of the global cycle.

In order to isolate the effect of internal data conflict from any delay caused by slow communication with the host, we assume that external inputs on $I_3$ of cells $B_{-h}, \ldots, B_h$ as well as on $I_1$ of cell $-B_h$ and $I_2$ of cell $B_h$ are available when needed. We also assume that $MV_{2,p}$ terminates execution, for a specific input matrix $A$ , in $N_2$ global cycles. With this, we may define the function $\alpha$:$[-B_h, B_h] \times [1, N_2] \rightarrow A$ such that $\alpha(k,t)$ is the element of $A$ that appears on port $I_3$ of cell $k$ at the beginning of the processing phase of the $t^{th}$ global cycle.

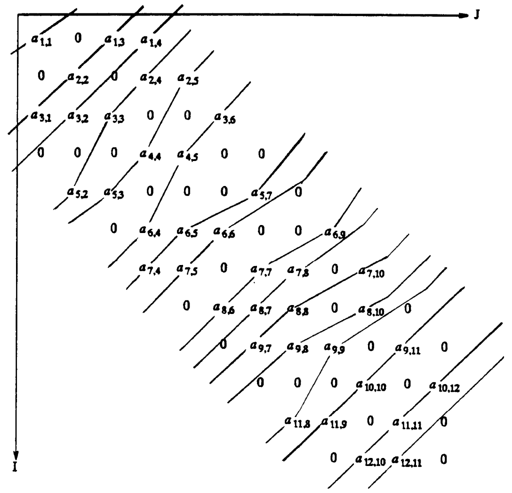
Although a data item is always available on $I_3$ of any cell during a specific processing phase, only those cells which receive the corresponding elements of the vectors $y$ and $x$ on $I_1$ and $I_2$, respectively, perform a multiply/add operation, while the other cells remain idle. Let $M_t$ be the subset of cells which are not idle during the $t^{th}$ processing phase and define the $t^{th}$ computation front as the set of elements of $A$ which are operated upon during this phase. More precisely,

$$CF_t = \{\alpha(k,t) \mid k \in M_t\}$$

Note that the members of $CF_t$ , for any $t$ , are non zero elements of $A$ .

The succession of computation fronts represents the progress in the execution of the pseudo systolic computation. More specifically, given a certain matrix, we may connect the elements of each front by a piece-wise linear curve and thus obtain a visual picture that describes the propagation of the computation. For example, we show in Fig. 2 the computation fronts corresponding to the specific given matrix pattern. Note that the concept of computation fronts is the same as that suggested in [6]. However, by allowing irregular fronts, we are able to model computations that depend on the value of the input as well as its availability.

In [5], we give a method for the automatic construction of computation fronts for pseudo systolic computations. Clearly, the number of such fronts is equal to the number of global cycles needed to complete the computation.



J
I
$a_{1,1}$ 0 $a_{1,3}$ $a_{1,4}$
0 $a_{2,2}$ 0 $a_{2,4}$ $a_{2,5}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ 0 0 $a_{3,6}$
0 0 0 $a_{4,4}$ $a_{4,5}$ 0 0
$a_{5,2}$ $a_{5,3}$ 0 0 0 $a_{5,7}$ 0
0 $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ 0 0 $a_{6,9}$
$a_{7,4}$ $a_{7,5}$ 0 $a_{7,7}$ $a_{7,8}$ 0 $a_{7,10}$
0 $a_{8,6}$ $a_{8,7}$ $a_{8,8}$ 0 $a_{8,10}$ 0
0 $a_{9,7}$ $a_{9,8}$ $a_{9,9}$ 0 $a_{9,11}$ 0
0 0 0 $a_{10,10}$ 0 $a_{10,12}$
$a_{11,8}$ $a_{11,9}$ 0 $a_{11,11}$ 0
0 $a_{12,10}$ $a_{12,11}$ 0

Fig. 2 - The Computation Fronts for $MV_{2,p}$

## 3. A NETWORK FOR MATRICES WITH NON-ZERO DIAGONAL ELEMENTS

Most of the matrices resulting in practical applications have non-zero diagonal elements, and it may be shown [5] that $MV_2$ is not suitable for this type of matrices because it does not allow computation fronts to be parallel to the diagonal of the matrix. In this section, we introduce a network, $MV_3$, which allows the fronts to be parallel to the diagonal.

For simplicity, we introduce $MV_3$ for dense, banded, matrices. It is composed of B cells (see Fig 3). Each cell has two input ports and two output ports, and is equipped with a counter "Ct" and an accumulator "Acc". The cycle of a cell may be described as follows:

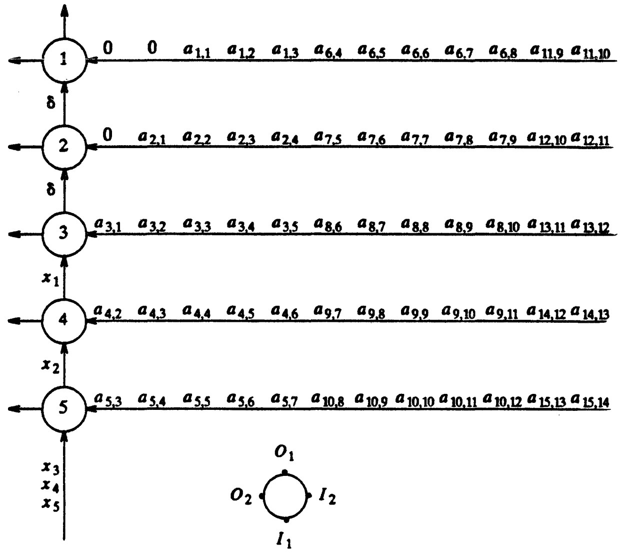

1
0 0 $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ $a_{6,7}$ $a_{6,8}$ $a_{11,9}$ $a_{11,10}$
δ
2
0 $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{7,5}$ $a_{7,6}$ $a_{7,7}$ $a_{7,8}$ $a_{7,9}$ $a_{12,10}$ $a_{12,11}$
δ
3
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{8,6}$ $a_{8,7}$ $a_{8,8}$ $a_{8,9}$ $a_{8,10}$ $a_{13,11}$ $a_{13,12}$
$x_1$
4
$a_{4,2}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{4,6}$ $a_{9,7}$ $a_{9,8}$ $a_{9,9}$ $a_{9,10}$ $a_{9,11}$ $a_{14,12}$ $a_{14,13}$
$x_2$
5
$a_{5,3}$ $a_{5,4}$ $a_{5,5}$ $a_{5,6}$ $a_{5,7}$ $a_{10,8}$ $a_{10,9}$ $a_{10,10}$ $a_{10,11}$ $a_{10,12}$ $a_{15,13}$ $a_{15,14}$
$x_3$ $x_4$ $x_5$
$O_1$
$O_2$
$I_2$
$I_1$

Fig. 3 - $MV_{3,p}$ after two cycles ($B = 5$)

```
CYCLE 3: /* Initially, $Ct = B_h + 1$. This allows data
           to fill-in during the first $B_h$ cycles */
(1) Wait until data is available on $I_1$ and $I_2$ ;
    $\alpha = [I_1]$ ; $\beta = [I_2]$
(2) $\rho = [Acc] + \alpha * \beta$
(3) Wait until previous data on $O_1$ and $O_2$ are consumed
(4) If $Ct = B$ THEN $O_1 \leftarrow \alpha$ ; $O_2 \leftarrow \rho$ ; $Acc \leftarrow 0$ ; $Ct = 1$
                 ELSE $O_1 \leftarrow \alpha$ ; $Acc \leftarrow \rho$ ; $Ct = Ct + 1$
```

The elements of the vector $x$ are applied to port $I_1$ of cell $B$, and the elements in the rows $k$, $k+B$, $k+2B$,..., of the matrix $A$ are concatenated and applied to port $I_2$ of cell $k$ (see Fig 3). Given this input, and assuming pseudo systolic synchronization, it may be easily seen that the elements $y_1,...,y_B$ of the result vector are produced on ports $O_2$ of cells $1,...,B$, respectively, after $B_h+B$ global cycles. The elements $y_{B+1},...,y_{2B}$ are produced on the same ports at the end of cycle $B_h+2B$, and in general, $y_{rB+1},...,y_{rB+B}$, $r=0,1,...$ are produced at the end of cycle $B_h+(r+1)B$. In other words,

$$T_{3,p} = (B_h + \beta B)(\tau_m + \tau_c)$$

where, $\beta=((n-1)\div B)+1$. Similar to the case of $MV_1$, the execution time of $MV_3$ may be reduced for sparse matrices, if step 2 in CYCLE 3 is replaced by a conditional statement that skips trivial operations. We call the resulting network $MV_4$. The construction of the computation fronts for the pseudo systolic version of $MV_4$ provides a means for the estimation of its speed-up over $MV_3$, obtained by skipping trivial operations.

## 4. A NETWORK WITH REDUCED NUMBER OF CELLS

Numerical Techniques for the solution of partial differential equations are major sources of large sparse matrices. The matrices resulting from these techniques have a band with $B = O(\sqrt{n})$, and hence the number of cells needed in $MV_2$ and $MV_4$ grows with the size of the matrix.

In order to obtain a network with a number of cells independent of $n$, we use the property in these matrices that the number of elements in each row is bounded by a constant which is independent of $n$. This property permits the inclusion of all the non-zero elements of the matrix into a stripe structure that contains few stripes which are, almost, parallel to the diagonal. A specific cell is then assigned to perform the operations associated with each stripe. First, we define a stripe structure of a matrix.

*Definition* : A $Q$-stripe $S$ of an $n \times n$ matrix $A$ is a set of positions $S = \{(i, \sigma(i)) \mid i = 1,...,n\}$ where $\sigma$ is a non-decreasing function. The prefix $Q$ (stands for "Quasi") is used to distinguish Q-stripes from a more restrictive definition of stripes that is given in [4]. However, only $Q$-stripes are referred to in this paper, and hence the prefix Q will be omitted.

*Definition:* A stripe structure of $A$ is a sequence of stripes $S_1=\{(i, \sigma_1(i))\}$, ..., $S_\pi=\{(i, \sigma_\pi(i))\}$ such that

$$\sigma_k(i) < \sigma_{k+1}(i) \quad i=1,...,n, \quad k=1,...,\pi-1$$

and $S_1 U...U\ S_\pi$ contains all the positions of nonzero elements in $A$. That is $a_{i,j} \neq 0$ only if $(i,j) \epsilon\ S_1 U...U\ S_\pi$. For example, we show in Fig. 4 the stripe structure of a specific matrix. Note that some zero elements are included in each stripe because, by definition, a stripe should contain a position for each row.

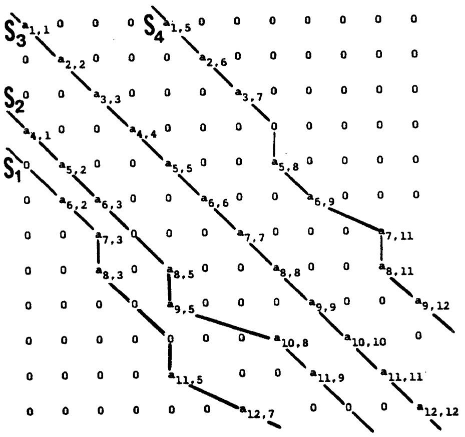

$S_3$ $a_{1,1}$ 0 0 0 $S_4$ $a_{1,5}$ 0 0 0 0 0 0 0
0 $a_{2,2}$ 0 0 0 $a_{2,6}$ 0 0 0 0 0 0
$S_2$ 0 0 $a_{3,3}$ 0 0 0 $a_{3,7}$ 0 0 0 0 0
$a_{4,1}$ 0 0 $a_{4,4}$ 0 0 0 0 0 0 0 0
$S_1$ 0 $a_{5,2}$ 0 0 $a_{5,5}$ 0 0 $a_{5,8}$ 0 0 0 0
0 $a_{6,2}$ $a_{6,3}$ 0 0 $a_{6,6}$ 0 0 $a_{6,9}$ 0 0 0
0 0 $a_{7,3}$ 0 0 0 $a_{7,7}$ 0 0 0 $a_{7,11}$
0 0 $a_{8,3}$ 0 $a_{8,5}$ 0 0 $a_{8,8}$ 0 0 $a_{8,11}$
0 0 0 0 $a_{9,5}$ 0 0 0 $a_{9,9}$ 0 0 $a_{9,12}$
0 0 0 0 0 0 0 $a_{10,8}$ $a_{10,10}$ 0
0 0 0 0 $a_{11,5}$ 0 0 $a_{11,9}$ $a_{11,11}$
0 0 0 0 0 0 $a_{12,7}$ 0 0 0 $a_{12,12}$

Fig 4 - A stripe structure

Given a stripe structure of a matrix A, we may use the network $MV_5$ shown in Fig. 5 for the multiplication of $A$ by a vector. The elements of the vector $x$ are fed to port $I_1$ of cell $\pi$ and the elements of the result vector $y$, initialized to zero, are fed to port $I_2$ of cell 1. Successive elements in a particular stripe $S_k$ are fed to port $I_3$ of cell $k$, and along with each element $a_{i,\sigma_k(i)} \epsilon\ S_k$ supplied on $I_3$, the value of $\sigma_k(i)$ is supplied on port $I_4$.

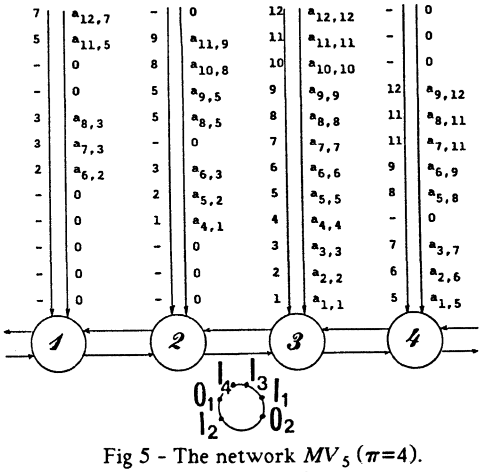

7 $a_{12,7}$ - 0 12 $a_{12,12}$ - 0
5 $a_{11,5}$ 9 $a_{11,9}$ 11 $a_{11,11}$ - 0
- 0 8 $a_{10,8}$ 10 $a_{10,10}$ - 0
- 0 5 $a_{9,5}$ 9 $a_{9,9}$ 12 $a_{9,12}$
3 $a_{8,3}$ 5 $a_{8,5}$ 8 $a_{8,8}$ 11 $a_{8,11}$
3 $a_{7,3}$ - 0 7 $a_{7,7}$ 11 $a_{7,11}$
2 $a_{6,2}$ 3 $a_{6,3}$ 6 $a_{6,6}$ 9 $a_{6,9}$
- 0 2 $a_{5,2}$ 5 $a_{5,5}$ 8 $a_{5,8}$
- 0 1 $a_{4,1}$ 4 $a_{4,4}$ - 0
- 0 - 0 3 $a_{3,3}$ 7 $a_{3,7}$
- 0 - 0 2 $a_{2,2}$ 6 $a_{2,6}$
- 0 - 0 1 $a_{1,1}$ 5 $a_{1,5}$
1
2
3
4
$I_4$ $I_3$
$O_1$ $I_1$
$I_2$ $O_2$

Fig 5 - The network $MV_5$ ($\pi$=4).

In order to provide the flexibility needed to deal with sparse structures, we assume that each cell contains a counter $CX$ that keeps track of the index of the data received on $I_2$.

Noting that data on $I_3$ and $I_4$ are always available, the operation of each cell may be described by

```
CYCLE 5 : /* Initially $CX = 0$ */
(1) Wait until data is available on $I_2$ ;
    $\alpha = [I_3]$ ; $j = [I_4]$ ; $\eta = [I_2]$
(2) While (CX < j) Do
        Wait until data is available on $I_1$ ;
        $\xi = [I_1]$ ; $CX = CX+1$ ; $O_1 \leftarrow \xi$
(3) $\eta = \eta + \alpha * \xi$
(4) $O_2 \leftarrow \eta$
```

More descriptively, after cell $k$ receives $a_{i,\sigma_k(i)}$, it continues to transmit the components of $x$ from $I_1$ to $O_1$ until it finds $x_{\sigma_k(i)}$. At this time the inner product is computed (step 3) and the result is written out. It is easy to see that the elements of the result vector $y = Ax$ are produced on $O_2$ of cell $\pi$. However, the network may operate correctly only if $\sigma_k(i) < \sigma_{k+1}(i)$ and $\sigma_k(i) \leqslant \sigma_k(i+1)$, which are satisfied by the definition of a stripe structure. Note that we assumed, implicitly, that the queues on the communication links which carry the x-data stream in $MV_5$ do not overflow. In [4], the maximum size, $d_x$ of these queues is found to be equal to the maximum separation between the stipes of the matrix.

A pseudo systolic version $MV_{5,p}$ of $MV_5$ may be obtained by inserting a "wait for SYNC" in step 3. The conditions for two elements of A to be in the same computation front may be easily derived [4] and used for the systematic construction of the fronts.

## 5. AN EXAMPLE

In order to compare the different networks presented in this paper, we consider the matrix that is generated from the discretization of a second order partial differential equation on the grid shown in Fig 6. For this matrix, $n = 270$ and $B = 39$, but only 16% of the elements within the band are non-zeroes. The diagonal elements of the matrix are non-zeroes, and seven non-overlapping stripes may cover all the non-zero elements.

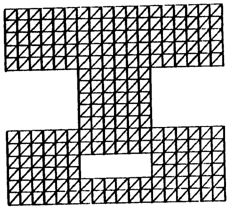


Fig 6 - A finite element grid

In Table 1, we compare the execution time of the different networks for this specific matrix. The number of global cycles is estimated by the construction of the computation fronts. The execution time of each global cycle, however, depends on the execution time of its communication and processing phases. The time of a processing phase is, roughly, $\tau_m$. However, the time of a communication phase depends on the communication activities during that phase. Although we did not discuss a way for the estimation of the communication activities, we give in Table 1 an estimate for these activities obtained by an analysis of the data profiles which are associated with the computation fronts. The details of this type of analysis may be found in [4] and [5].

For $MV_{4,p}$ and queue size d=1, execution terminates in 60 global cycles instead of 292 for $MV_{3,p}$. That is, a speed up of 4.867 is obtained (the increase in the communication time is small). Noting that only 16% of the operations in $MV_{3,p}$ involve non-zero operands, and thus that only one out of $\frac{100}{16}$=6.36 operations is useful, it is clear that, by skipping trivial operations in $MV_{4,p}$, we can retrieve $\frac{4.867}{6.36} \approx 76\%$ of the inefficiency in $MV_{3,p}$. Of course, internal data conflict accounts for the unretrieved 24%.

| | number of cells | size of comm. queues | number of global cycles | execution time |
|---|---|---|---|---|
| $MV_1$ | 39 | d=1 | 559 | $559\tau_m + 559\tau_c$ |
| $MV_2$ | 39 | d=1 | 270 | $270\tau_m + 305\tau_c$ |
| $MV_3$ | 39 | d=1 | 292 | $292\tau_m + 292\tau_c$ |
| $MV_4$ | 39 | d=1 | 60 | $60\tau_m + 425\tau_c$ |
| $MV_4$ | 39 | d=3 | 54 | $54\tau_m + 492\tau_c$ |
| $MV_5$ | 7 | $d_x$=12<br>$d_y$=1 | 279 | $279\tau_m + 305\tau_c$ |

Table 1 - Performance of the different networks

The result for $MV_5$ is obtained assuming that each y-stream communication link may buffer only one element, and that each x-stream communication link may buffer at least 12 data items. Clearly, the merit of $MV_5$ is not due to the speed up it achieves as much as it is for its small number of cells.

## 6. CONCLUSION

Pseudo systolic computations and irregular computation fronts are two tools that are used effectively in the analysis of the networks introduced in this paper. The alternative for this type of analysis is the simulation of the computations. Such simulation, however, requires explicit assumptions about the parameters $\tau_m$ and $\tau_c$, which are strongly dependent on technology and architectural details.

Our primary interest is in sparse systems that result from the solution of partial differential equations, which is one of the major sources of sparse matrices. For this reason, the networks suggested in Sections 3 and 4 take advantage of some specific properties in these matrices. However, the concept that we introduce is quite general. Namely, the inclusion of the non-zero elements of a sparse matrix in a pattern which is regular enough to allow for the efficient manipulation of the matrix on VLSI networks. The results reported in Section 5, and other similar results [4,5], support our argument.

## REFERENCES

[1] A. George and J. Liu, "Computer Solutions of Large Sparse Positive Definite Systems," Prentice-Hall series in Computational Mathematics (1981).

[2] H. T. Kung and C. E. Leiserson, "Systolic Arrays for VLSI," in Introduction to VLSI Systems, Ed. by C. Mead and L. Conway, Addison-Wesley, Reading, Mass (1980).

[3] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer and D. B. Rao, "Wavefront Array Processor: Language, Architecture and Applications," IEEE Trans. on Computers, C-31, (1982).

[4] R. Melhem, "Parallel Solution of Linear Systems with Sparse Striped Matrices; Parts 1 and 2," Tech. Reports ICMA-86-91/92, The University of Pittsburgh.

[5] R. Melhem, "A Study of Data Interlock in Computational Networks for Sparse Matrix Multiplication," Tech. Report TR-CS-505, Purdue University. (To appear in IEEE Trans. on Computers).

[6] U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in VLSI Systems and Computations. Ed. by H. T. Kung, B. Sproull and G. Steele, Computer Science Press (1981).

# ON THE SYSTOLIC DETECTION OF SHORTEST ROUTES

Uwe Schwiegelshohn and Lothar Thiele
Technical University Munich
Institute of Network and Circuit Theory
D-8000 Munich 2, Arcisstr.21, GERMANY

**Abstract** -- This paper presents a parallel algorithm for the solution of the single-pair, single-source and all-pairs shortest path problems. The first time, all essential properties of the shortest paths can be computed on a pure systolic parallel processor in $O(n)$ time and on $O(n^2)$ area. If $O(n^3)$ area is spended, all single-pair shortest paths can be stored within the same time bound.

## I. Introduction

In this paper we are concerned with the well known shortest path problem. The single-pair, single-source and all-pairs problems entail finding shortest paths between two specified vertices, from one vertex to all others and between all pairs of vertices, respectively.

We concentrate on the systolic model of computation. In order to conform to the basic VLSI restrictions the parallel computer consists of single processing elements which have some local memory and are able to compute a simple set of instructions. Global communication and a severe system overhead are avoided as no shared memory is allowed.

In order to solve the all-pairs shortest path problem on a parallel computer the use of the Ford-Bellman-Moore, the matrix multiplication and the Floyd-Warshall algorithms have been proposed [1-6]. Lakhani [3] uses the Ford-Bellman-Moore algorithm to solve the all-pairs problem in $O(n^2)$ time. However, the complex architecture requires a complicated communication scheme and area consuming interconnections between processing cells. In [1,2,4,5,6] it is shown that the **distances** of the all-pairs shortest paths can be computed in $O(n)$ time and on $O(n^2)$ area if a parallel version of the Floyd-Warshall algorithm (**FWA**) is used.

But, in almost any application of the shortest path (**SP**) problem the **detection of the optimal route** itself is indispensible.

The new approach uses the same square arrangement of n×n identical processing cells as [1,2,4] or [5,6]. The following extensions are essential:

- In comparison to [7] within the same kind of loops of the FWA the predecessor and depth of each vertex w.r.t any SP is determined.
- In order to deal with the trees of SPs a conversion to an edge-based description is performed. Sorting operations are used to compute the desired path functions.

The first time, the following quantities can be computed on a systolic processor in $O(n)$ time and on $O(n^2)$ area: distance matrix, selection of a shortest route with minimal depth, predecessors and depth of any vertex w.r.t any SP, single-source trees of SPs as ordered lists of edges, adjacency matrix of a tree of SPs, single-pair SP as a list of edges, all-pairs SPs as $n^2$ lists of edges (if $O(n^3)$ area is spended).

## II. The Extended Floyd-Warshall Algorithm

Let us first clarify the notation used in the subsequent paper. We assume that each edge $(i,j)\varepsilon E$ has an associated integer valued length $l(i,j)$. A path from vertex r to t can be regarded as a tupel $P^k=(i_0,i_1,\ldots,i_k)$ of vertices $i_j$ with $k\geq 1$, $i_0=r$, $i_k=t$ and $(i_{j-1},i_j)\varepsilon E$ for all $1\geq j\geq k$. The path $P^k$ has the depth k which means that it consists of k edges and k+1 vertices. The length l of a path is the summation of the lengths of its edges. In order to clarify the extended FWA the following definitions can be used:

$\Pi_m(r,t) = \{\, P^k | k\geq 1, i_0=r, i_k=t, i_j\leq m \text{ for all } 1\leq j<k \,\}$
$\hat{=}$ set of paths at stage m from vertex r to t where all intermediate vertices are $\leq$ m

$d_m(r,t) = \text{Min}\{\, l(P^k) \mid P^k\varepsilon\Pi_m(r,t) \,\}$
$\hat{=}$ shortest distance at stage m from vertex r to t

$\Delta_m(r,t) = \{\, P^k \mid P^k\varepsilon\Pi_m(r,t) \text{ and } l(P^k)=d_m(r,t) \,\}$
$\hat{=}$ set of shortest paths from vertex r to t at stage m

$s_m(r,t) = \text{Min}\{\, k \mid P^k\varepsilon\Delta_m(r,t) \,\}$
$\hat{=}$ minimum depth at stage m from vertex r to t

$\Sigma_m(r,t) = \{\, P^k \mid P^k\varepsilon\Delta_m(r,t) \text{ and } k=s_m(r,t) \,\}$
$\hat{=}$ set of shortest paths at stage m from r to t with minimum depth

Assuming that one path $P^k\varepsilon\Sigma_m(r,t)$ has been selected it is possible to define the following values:

$p_m(r,t) = i_{k-1}$
$\hat{=}$ predecessor of t in the selected path from vertex r to t

In extension of e.g.[7] the algorithm described in Fig.1 has the following properties:

- Concurrent determination of the m-stage distance, predecessor and depth matrices
- Concurrent selection of the all-pairs SPs at stage m with minimum number of edges.

It will be seen in the next section that these improvements enable the computation of the shortest route between two specified vertices. Now, we can formulate the subsequent Lemma:

**Lemma:** If no negative cycle exists then $\Sigma_m(m,j)=\Sigma_{m-1}(m,j)$ and $\Sigma_m(j,m)=\Sigma_{m-1}(j,m)$ for each j.

**Proof:** We restrict the proof on $\Sigma_m(m,j)=\Sigma_{m-1}(m,j)$. Let us assume that $P^k\varepsilon\Sigma_m(m,j)\backslash\Sigma_{m-1}(m,j)$. Therefore $P^k$ could be partitioned into a cycle through m of depth b and $P^a=(m,\ldots,j)\varepsilon\Pi_{m-1}(m,j)$ where the length

0190-3918/86/0000/0762 $01.00 © 1986 IEEE

of the cycle is $l \geq 0$ and $b>0$. It would follow that $k=a+b>a \geq s_{m-1}(m,j)$ and $l(P^k)=l+l(P^a) \geq l(P^a) \geq d_{m-1}(m,j)$ which contradicts the assumption. □

Consequently it is possible to dispense with the modification of the elements $\cdot(m,j)$ and $\cdot(j,m)$ at stage m of the algorithm of Fig.1.

**Theorem:** At stage m the modified FWA detects a path $P^k$ from vertex r to t with $P^k \varepsilon \Sigma_m(r,t)$.

**Proof:** We assume by an inductive hypothesis that the theorem holds at stage m-1. The path $P^k$ is the resulting path from r to t of the extended FWA. We distinguish between two cases a) and b):

a) $\Sigma_{m-1}(r,t) \cap \Sigma_m(r,t) \neq \emptyset$
  $\rightarrow s_{m-1}(r,t)=s_m(r,t)$ and $d_{m-1}(r,t)=d_m(r,t)$
  $\rightarrow P^k \varepsilon \Sigma_{m-1}(r,t) \subseteq \Sigma_m(r,t)$

b) $\Sigma_{m-1}(r,t) \cap \Sigma_m(r,t)=\emptyset$
  $\rightarrow d_{m-1}(r,t)>d_m(r,t)$ or
  $( d_{m-1}(r,t)=d_m(r,t)$ and $s_{m-1}(r,t)<s_m(r,t) )$
  $\rightarrow P^k$ consists of paths $P^a \varepsilon \Sigma_{m-1}(r,m)$ and
  $P^b \varepsilon \Sigma_{m-1}(m,t)$ selected of FWA at stage m-1

  We assume that there exists $P^\kappa \varepsilon \Sigma_m(r,t)$ which consists of $P^\alpha \varepsilon \Pi_{m-1}(r,m)$ and $P^\beta \varepsilon \Pi_{m-1}(m,t)$
  $\rightarrow d_m(r,t)=l(P^\kappa)=l(P^\alpha)+l(P^\beta) \geq l(P^a)+l(P^b)=l(P^k)$
  and $s_m(r,t)=\kappa=\alpha+\beta \geq a+b=k$ if
  $P^\alpha \varepsilon \Delta_{m-1}(r,m)$ and $P^\beta \varepsilon \Delta_{m-1}(m,t)$ .

Now it has been proved that the values $d_m$, $p_m$ and $s_m$ are evaluated correctly by the extended FWA. □

The extended FWA detects for the all-pairs SP a SP with minimum depth as well as the Ford-Bellman-Moore algorithm for the single-source problem does. In contrary, the common FWA can only assure that $P^k \varepsilon \Delta_n(r,t)$.

## III. Determination of a Single-Pair Shortest Path

The detection of a SP from vertex r to t can be divided into two different steps:

- Determination of a tree which contains the SP
- Finding of the SP between the vertices r and t

Obviously, the tree of SP with the root vertex r is an appropriate one for the first part. The tree is determined by its edges (p(r,j),j) evaluated by the FWA. These values can be stored in a linear array of length n to compute the desired path according to the second step.

But problems arise as data sharing is restricted to neighboring processors only in a systolic array and the necessary data may be far apart. Up to n-1 sequential operations each of which requiring O(n) steps in the worst case are needed if no suitable order of edges exists. In contrary to [8] a much simpler sorting scheme than the Euler circuit can be used to receive a time bound of O(n) as the depth s(r,j) of each node j is available from FWA. The following solution strategy is chosen:

- Sorting of the edges (p(r,j),j) according to the depths s(r,j). The following chain is obtained:
  $(p(r,j_1),j_1)\ (p(r,j_2),j_2)\ \ldots\ (p(r,j_n),j_n)$
  with $s(r,j_i) \leq s(r,j_{i+1})$ for all $i=1,\ldots,n-1$
- Scanning of the sorted chain in descending order of depths and selection of the SP from r to t according to the following procedure:

```
procedure path
  for i=n down to 1 do begin
    if $j_i$=t then begin
      $j_i$ ε 'path from r to t'
      t:=p(r,$j_i$)
    end
```

## IV. Systolic Implementation

At first, the solution to the first subproblem is described where Fig.2 shows the interconnection scheme of the array apart from the outputs T1 to T4 and the inputs Φ,In4. As this part is similar to [5] the internal cells are described only. The following notations are used: $R(i,j)=[f_1,f_2,\ldots]$ denotes a register within cell C(i,j) where the datas $f_1,\ldots$ are stored in a common record. R(i,j).k denotes $f_k$. The sign ← denotes an assignment between cells which needs one time step. Corresponding data are transferred. The sign := denotes an assignment and :=: an exchange of data within one cell and one instant.

```
procedure C(l,m)   { 2≤l,m≤n ; $R_{123}$=[d⊙s,p] }
  $R_1$ ← $R_1$(l,m-1)
  $R_2$ ← $R_2$(l-1,m)
  $R_3$ ← $R_3$(l+1,m+1)
  if $R_3$.1>$R_1$.1+$R_2$.1  then  $R_3$:=[$R_1$.1+$R_2$.1 , $R_2$]
```

The operation of this cell corresponds to the 'if'-statement of the FWA. The executuion of the 'and'- and 'or'-operations can be avoided by the concatenation d(i,j)⊙s(i,j) where the bits of s(i,j) are linked less significantly to those of d(i,j). The sequence of input and output operations of the systolic FW-array are similar to [5,6].

### 1. Solution of the single-source SP problem

The first line of the array according to Fig.2 contains in addition to the cells C(1,m) the sorting cells S(1,m). Moreover, all registers of the FW cells contain besides d(i,j), s(i,j), p(i,j) the corresponding node j with R=[d⊙s,p,j]. The output T1 successively carries the edges (p(r,j),j) of the tree of SPs rooted from vertex r in descending order of depths s(r,j).

```
procedure S(1,1)     { In,$R_4$=[φ] ; $R_5$=[s,p,j,φ] }
  $R_4$ ← In  ;  $R_5$ ← $R_3$(2,2)
  if $R_4$=1  then  $R_5$.4:=1
              else  $R_5$  :=[∞,∞,∞,0]

procedure S(1,m)     { 2≤m≤n  ;  $R_{567}$=[s,p,j,φ] }
  $R_5$ ← $R_5$(1,m-1)  ;  $R_7$ ← $R_3$(2,m+1)
  if $R_5$.4=1     then  $R_6$:=$R_7$  ; $R_6$.4:=0
  if $R_5$.1<$R_6$.1 then  $R_5$:=:$R_6$
  if $R_6$.4=1     then  $R_5$.4:=1 ; $R_6$.4:=0

procedure S(1,n+1)  { T1,$R_5$=[s,p,j,φ] }
  $R_5$ ← $R_5$(1,n)   ;   T1:=$R_5$
```

Cell S(1,1) has to be initialized with $\phi$=1 at the time where the elements $\cdot$(r,r) are read in. This correct timing can be achieved also by detecting that $R_5$.3=r in cell S(1,1). This initialization propagates from m=1 to m=n+1.

### 2. Solution of the single-pair SP problem

In contrary to the foregoing array, cell S(1,n+1) marks all edges which belong to the SP from r to t. To this end, the vertex number t has to be read into this cell before the first edge of the single-source tree enters. The variable $\psi$ is $\psi$=1 if the edge is part of the SP from r to t.

```
**procedure** S(1,n+1)  { T1,$R_5$=[s,p,j,$\psi$] ; In4,$R_8$=[p] }
   $R_5 \leftarrow R_5(1,n)$
   **if** 'input operation' **then** $R_8$:=In4
   **if** $R_5$.3=$R_8$.1 **then** $R_8$:=[$R_5$.3] ; $R_5$.4:=[1]
                     **else** $R_5$.4:=[0]
   T1:=$R_5$
```

If the output T1 propagates to the left through the cells S(1,m) it is possible to delete the edges with $\psi$=0 such that the searched edges can be read out from S(1,1) concurrently.

### 3. Solution of the all-pairs SP problem

According to Fig.2 each row contains a copy of the linear sorting array just described. The initialization has to take place if $\cdot$(1,1) enters S(1,1). After that, the initialization propagates down and right in form of a wave front. The outputs Tl concurrently carry the edges of the trees of SPs rooted from vertices l.

If the outputs Tl are propagated to the left, it is possible to store in S(l,m) the SP from l to m in form of a list of edges (O($n^3$) area is needed).

## V. Concluding Remarks

It is noted that the above algorithms can be used to solve other combinatorial optimization problems which require the determination of a shortest paths or breadth-first spanning tree. The determination of the adjacency matrix of the single-source tree of SP is possible too in O(n) time if the cells in the first row and last column of the systolic array have additional functions.

Recently, it has been shown by the authors [9] that algorithms of the form shown in Fig.1 can be carried out on a linear systolic array in O($n^2$) time with O(n) operating cells. The data at the input and output occur in lexicographical order and pipelined arithmetic units can be used.

**Acknowledgement** -- The authors wish to thank Professor R. Saal for his interest and continuous support on the subject of this paper.

## References

[1] L.J. Guibas, H.-T. Kung, and C.D. Thompson: Proc. Caltech Conf. VLSI, 1979, pp. 509-525.

[2] M.J. Atallah and S.R. Kosaraju: Proc. Fourteenth Annual ACM Symposium on the Theory of Computing, 1982, pp. 345-353.

[3] G.D. Lakhani: IEEE Trans. Comput., vol. C-33, pp. 855-857, Sept. 1984.

[4] N. Deo, C.Y. Pang and R.E. Lord: Proc. of the 1980 ICPP, pp.244-253, Aug. 1980.

[5] G. Rote: Computing 34, pp.191-219, 1985.

[6] S.Y.Kung and S.C.Lo: Proc. of ICDD, Oct.1985.

[7] S.E. Dreyfus and A.M. Law, The Art and Theory of Dynamic progamming, New York, NY: Academic Press, 1977.

[8] P.S. Gopalakrishnan, C.V. Ramakrishnan and L.N.Kanal: Proc. of the 1985 ICPP, pp.703-710, Aug. 1985.

[9] U. Schwiegelshohn and L. Thiele, "A linear systolic array for computations on matrices" submitted Int. Journal on Parallel and Distributed Computing, 1986.

```
**procedure** initialization
  **for** all pairs of vertices i,j **do begin**
    $d_0(i,j) := \begin{cases} l(i,j) & \text{if } (i,j)\varepsilon E \\ \infty & \text{otherwise} \end{cases}$
    $s_0(i,j) := \begin{cases} 1 & \text{if } (i,j)\varepsilon E \\ \infty & \text{otherwise} \end{cases}$
    $p_0(i,j) := \begin{cases} i & \text{if } (i,j)\varepsilon E \\ \infty & \text{otherwise} \end{cases}$
  **end**

**procedure** modification
  **for** k:=1 **to** n **do begin**
    **for** all pairs of vertices i,j **do begin**
      **if** { $d_{k-1}(i,j)>d_{k-1}(i,k)+d_{k-1}(k,j)$ } **or**
         ( $d_{k-1}(i,j)=d_{k-1}(i,k)+d_{k-1}(k,j)$ **and**
           $s_{k-1}(i,j)>s_{k-1}(i,k)+s_{k-1}(k,j)$ )
      **then begin**
        $d_k(i,j):=d_{k-1}(i,k)+d_{k-1}(k,j)$
        $s_k(i,j):=s_{k-1}(i,k)+s_{k-1}(k,j)$
        $p_k(i,j):=p_{k-1}(k,j)$
      **end**
      **else begin**
        $d_k(i,j):=d_{k-1}(i,j)$
        $s_k(i,j):=s_{k-1}(i,j)$
        $p_k(i,j):=p_{k-1}(i,j)$
  **end**
```

**Fig.1: Extended Floyd-Warshall algorithm**

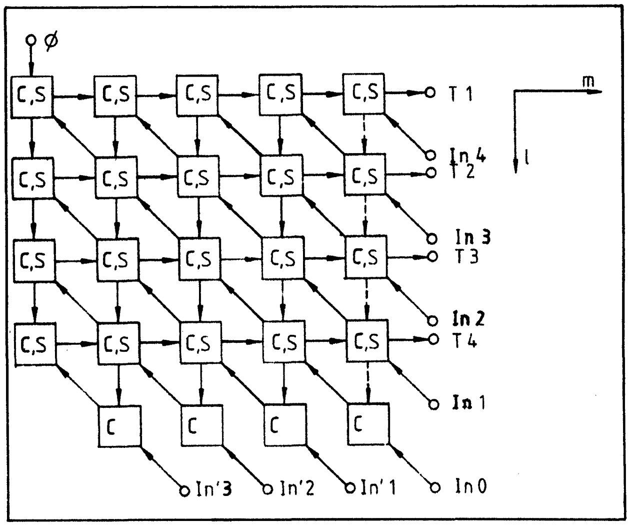


**Fig.2: Systolic array for the all-pairs SP problem**

# SYNTHESIZING NON-UNIFORM SYSTOLIC DESIGNS*

Concettina Guerra
Rami Melhem
Department of Computer Sciences, Purdue University
West lafayette, Indiana 47907

Abstract -- In this paper we propose a method to derive systolic designs with non-uniform data flow. One of the major difficulties in systematic design is in transforming the original sequential specification of a problem into a form suitable to VLSI implementation. Our approach to automatically restructuring a problem is based on a subset of the data dependencies extracted from the original problem specification. By using such dependencies we are able to identify chains of dependent computations which are then converted into recurrence equations. The mapping of the new specification into hardware is also based on data dependencies. We illustrate the methodology by applying it to algorithms using dynamic programming.

## 1. Introduction

The development of efficient VLSI algorithms is relevant to many applications including signal and image processing, which justifies the considerable interest for this topic. In recent years, the problem of synthesizing VLSI design - and systolic design, in particular - has also received much attention. Systematic methodologies for the derivation of systolic algorithms have proved useful both in finding new designs and in verifying the correctness of old ones. Moreover, the possibility of automatically generating a number of viable algorithms for the solution of a given problem enables the selection of an optimal algorithm among a wider set of candidates. The optimality can be based on such parameters as completion time $T$, number of processors $P$, chip area, etc.[9]. Most of the existing synthesis methods tend to minimize the completion time.

Systolic systems generally are characterized by simple processing elements and regular and local communication pattern. However, the definition of a systolic system changes according to different authors. Here we shall refer to synchronous systems, i.e. systems with a global clock for the timing of all processing elements, and assume that all computations are data independent.

The first and probably most challenging part of the systematic design process is in the transformation of the high-level problem specification into a form better suited to VLSI implementation. Such a form can be, for example, a system of first-order recurrence equations, a uniform recurrence equation, or nested loops with constant data dependencies. Often, this transformation is obtained by using techniques similar to those used for software compilers: buffering of variables, addition of new variables, etc. However, it is not well understood how to select a good transformation especially for some complex non-numerical problems. Some authors assume that the problem is already given in the required form, and concentrate on how to map it into hardware.

A fundamental distinction between different approaches to the mapping problem is whether they use data dependence based transformations [1, 3, 8-13] or delay operators applied to the mathematical expression of the algorithm [5]. For a survey of the various methods see [2].

The transformational approach based on data dependencies has been successfully applied to a variety of algorithms characterized by constant data dependencies. For such algorithms linear time and space transformations of the index set into VLSI arrays have been derived [11-13]. The resulting designs have constant and regular data flow. However, a number of more complex problems with non-constant data dependencies require non linear transformations. A typical example is the optimal parenthesization problem, which uses dynamic programming.

A non-linear transformation for dynamic programming was indicated in [12]. However, the paper does not precisely describe how the transformation is obtained from the algorithm specification. Chen [1] presents a methodology for mapping algorithms expressed as systems of first order recurrence relations into systolic arrays. The synthesis procedure allows designs with non-uniform communication patterns. The mapping is done inductively, starting at the boundary conditions. By using this technique, various designs, corresponding to different communication patterns in the systolic array, are derived for dynamic

* This work was in part supported under ONR Contract N00014-85-K-0339

0190-3918/86/0000/0765 $01.00 © 1986 IEEE

programming. The procedure, based on a point-by-point mapping, appears to be quite lengthy. In [10] a mathematical model, based on a sequence notation, is used to represent index transformations. The model provides a precise specification of systolic designs. It allows arbitrary interconnections in systolic networks and is not restricted to linear transformations. The paper does not give a constructive methodology to find the transformations.

This paper presents a systematic method for the design of VLSI algorithms. The method consists of transforming the high-level problem specification into a set of mutually dependent recurrence equations. To select the appropriate transformations we propose a two-step refinement procedure. The procedure first determines a coarse timing function of the computations in the original specification of the problem, based on a maximal set of constant data dependencies; then uses this function to guide the search for an index transformation. Ordered chains of dependent computations are identified in the algorithm, according to such function, and an index transformation is applied to each single chain. The index transformation must be compatible with the timing function, i.e. all the computations whose operands are available first will be performed first. Subsequently, the mapping of the obtained system of recurrence equations into a VLSI network is performed by applying linear time and space transformations separately to each recurrence subject to global constraints. The resulting design may have non-uniform data flow.

This paper is organized as follows. Section 2 describes a method which helps deriving from an abstract algorithm a set of mutually dependent recurrence equations. Section 3 illustrates the method by applying it to dynamic programming. In section 4 time-space transformations are used to map the new specification into a VLSI array.

## 2. Deriving a systolic algorithm from the high-level specification

In this section we discuss how to transform the high-level problem specification to adapt it to a VLSI architecture. The task of enhancing pipelining and local communication in an algorithm is generally accomplished by index transformations which involve adding indices to existing variables in the algorithm, or by renaming variables, or by introducing new variables. The aim of these transformations is to produce a new specification which is in *canonic form*.

*Definition.* A canonic form consists of a structured set of computations written as a recurrence relation or a nested loop. The set includes input statements, output statements, assignment statements, and conditional assignment statements. The recurrence relation or nested loop defines an index set $I^n=\{(i_1,\ldots,i_n) \mid l_1^1 \le i_1 \le l_1^2,\ldots,l_n^1 \le i_n \le l_n^2\}$, subset of the set $Z^n$ of the $n$-tuples of integers. We assume that the following conditions are satisfied:

CA1 - each variable in the algorithm is associated with an index vector $\bar{i}=(i_1,i_2,...,i_n)$, i.e. is an element of an $n$-dimensional array. In other words, there is a one-to-one correspondence between the $n$-tuples in $I^n$ and the dimensions of any array used in the algorithm.

CA2 - if $S$ is an assignment statement indexed by $(i_1,i_2,...,i_n)$, and a variable on the right side of $S$ is indexed by $(j_1,j_2,...,j_n)$, then each $i_k$ may depend only on $j_k$.

CA3 - a variable is used exactly once after it is generated.

It is not always possible to transform a given problem into canonic form. For problems which do not have such a representation we attempt to rewrite them into a form of many modules each being in canonic form, i.e. satisfying condition CA1-CA3.

The canonic form does not explicitly specify any ordering among the computations; the lexicographical ordering in $I^n$ is arbitrary and therefore irrelevant to our purposes. Rather, an implicit partial ordering of the computations is given by the data dependencies. The dependence vector of a variable is defined as the difference of the index vectors of computations where the variable is used and generated [7]. The data dependence vectors $\bar{d}_1,\ldots,\bar{d}_m$ associated to the variables of a recurrence can be represented as a matrix $D=[\bar{d}_1 \cdots \bar{d}_m]$, whose columns are labelled by the variable names. A variable may have many dependencies each corresponding to a different index vector; in this case there will be a column in the $D$ matrix for each pair *(variable, index vector)*, labelled by the pair itself. A partial ordering $>_D$ defined in $I^n$ by the data dependencies is such that $\bar{i} >_D \bar{i}'$ if $\bar{i}=\bar{i}'+\bar{d}_j$ for some $\bar{d}_j \in D$.

Transformations applied to derive a canonic form do not change the algorithm fundamentally, but only the data dependencies among the variables. The new data dependencies in the resulting specification of the problem are therefore not characteristic of the problem itself but of its parallel implementation. Indeed, as is well known, there are in general several ways to transform a given problem specification, according to the previous rules, each way producing a different set of data dependencies. However, not all the possible transformations lead to a feasible VLSI design. Moreover, different transformations may result in different performances. Consider the two following examples:

*Example 1.* The convolution problem is defined by : given a sequence $x_1,\ldots,x_n$ and a set of weights $w_1,\ldots,w_k$, determine the sequence $y_1,\ldots,y_n$ such that:

$$y_i \;=\; \textstyle\sum_{k=1,s} w_k * x_{i-k}$$

Broadcasting of $x$ and $w$ can be eliminated by adding one more index to all variables $x$, $w$, and $y$. At least two different index transformations can be applied which produce the two recurrences in canonic form below: the first is a backward recurrence, where the variable $y_{i,k}$ is defined in terms of $y_{i,k-1}$, and the second is forward recurrence, where the variable $y_{i,k}$ is defined in terms of $y_{i,k+1}$.

$$\begin{aligned} & 0 \le i \le n \ ; \ 0 \le k \le s \\ & w_{i,k} = w_{i-1,k} \\ & x_{i,k} = x_{i-1,k-1} \\ & y_{i,k} = y_{i,k-1} + w_{i,k} * x_{i,k} \end{aligned} \qquad (1)$$

$$\begin{aligned} & 0 \le i \le n \ ; \ 0 \le k \le s \\ & w_{i,k} = w_{i-1,k} \\ & x_{i,k} = x_{i-1,k-1} \\ & y_{i,k} = y_{i,k+1} + w_{i,k} * x_{i,k} \end{aligned} \qquad (2)$$

Data dependencies corresponding to the two recurrences introduce two different partial orderings in $I^2 = \{(i,k) \mid 0 \le i \le n;\ 0 \le k \le s\}$, which translate into two different schedules for the computations and consequently into different systolic designs.

*Example 2* . Recursive convolution

This problem can be expressed as: given a sequence of weights $w_1,..,w_k$, determine the sequence $y_1,..,y_n$ such that:

$$y_i \ = \ \sum_{k=1,s} w_k \ * \ y_{i-k}$$

Of the two recurrences which can be derived by using transformations similar to those used for the convolution problem, only the forward recurrence has to be considered for a systolic implementation. The backward recurrence uses data before they are generated therefore cannot result in any meaningful design.

For more complex problems, such as optimal parenthesization using dynamic programming, selecting a good initial transformation is not an obvious task and in fact straightforward transformations such as those applied to the convolution problem fail. It is clear that this part of the synthesis procedure sometimes requires creativity and programming experience. We suggest a way to assist the human designer in this crucial task. The method we propose here relies on identifying a set of constant data dependencies in the original formulation of the problem.

### A two-step procedure

We assume that the high level specification of a problem is written in the form of a nested loop with the index set $I^n$ defined by:

$$I^n = \{(i_1, \ldots, i_n) \mid l_1^1 \le i_1 \le l_1^2, \ldots, l_n^1 \le i_n \le l_n^2\}.$$

We assume that, unlike the canonic form, the loop contains a variable which is an $s$-dimensional array. More specifically, we let $s=n-1$ and assume that the loop contains an assignment statement of the form:

$$c(\bar{i^s}) = f(c(\bar{i^s} - \bar{d}_1^s), \ldots, c(\bar{i^s} - \bar{d}_m^s)) \qquad (3)$$

where: $\bar{i^s} = (i_1, i_2, \ldots, i_s)$ is an $s$-tuple of indices of the loop;

$\bar{d}_j^s = (a_{j,1}, .., a_{j,t-1}, i_t - i_n, a_{j,t+1}, .., a_{j,s})$ , $j=1,..,m$;

and where $a_{j,l}$ are integer constants. In other words, the index $i_t$ on the left side of (3) is replaced on the right side by the index $i_n$.

Each vector $\bar{d}_j^s$, $(j=1,..,m)$, represents a non constant data dependence for variable $c$, since its $t$-th component is a function of the two indices $i_t$ and $i_n$. We can expand $\bar{d}_j^s$ into a number of data dependence vectors, each corresponding to a specific value of index $i_n$ in the $t$-th component. Then, we associate to each pair (*variable ,index vector*)=$(c, \bar{i}_s)$ the set $D_{\bar{i^s}}^c$ of all the data dependence vectors obtained by expanding $\bar{d}_1^s,..,\bar{d}_m^s$.

Our strategy to select the initial index transformation consists of a two-step procedure. Let $I^s = \{(i_1,\ldots,i_s) \mid l_1^1 \le i_1 \le l_1^2, \ldots, l_s^1 \le i_s \le l_s^2\}$. First, determine from the high-level specification of the problem a coarse timing function $T: I^s \to Z$. This function will be used in the subsequent step of the procedure to guide the search for a schedule of computations indexed by $I^n$. We derive $T$ from a subset of the data dependencies of the algorithm, namely, the subset of constant data dependencies, thus, $T$ provides only a lower bound for an actual timing function for $I^s$. Furthermore, $T$ depends only on the implicit dependencies of the problem since it is derived before any arbitrary ordering of the computations is introduced. The set $D_{\bar{i^s}}^c$ of data dependencies is non constant in the computation space. However, the set $D^c$ defined as the intersection of $D_{\bar{i^s}}^c$ for all $\bar{i^s} \in I^s$ contains only constant data dependencies. For a nested loop with constant data dependencies, a linear (or quasi-affine) timing function can be determined by applying the transformational method described in [11,13]. Thus, if $D^c$ is not empty we can derive a linear timing function $T : I^s \to Z$ which is compatible with the set $D^c$, i.e. $\bar{i^s} >_{D^c} \bar{i'^s}$ implies $T(\bar{i^s}) > T(\bar{i'^s})$. In [11, 13] necessary and sufficient conditions for the existence of a linear timing function are given. In particular, it must be:

$$T(\bar{d}_i) > 0 \ \text{ for } \textit{each } \bar{d}_i \in D^c \qquad (4)$$

System (4) may have no solution or several solutions. In this latter case, the one which minimizes the total execution time (defined as the difference between the minimum and maximum value of $T$) is chosen.

It obvious that if $\tau$ is an actual timing function for $I^s$ then it must be $\tau(\bar{i^s}) \geq T(\bar{i^s})$ for each $\bar{i^s} \in I^s$. Moreover, because of the monotonocity of data dependencies in $D^c_{\bar{i^s}}$ , it must be $\tau(\bar{i^s}) > \tau(\bar{i'^s})$ if $T(\bar{i^s}) > T(\bar{i'^s})$.

The partial ordering defined by $T$ on the set of computations $I^s$ defines a partial ordering in a subset of $I^n$ based on the availability of operands. Consider the set $J^n \subset I^n$ consisting of the $n$-tuples $(\bar{i^s}, i_n)$ for any given $\bar{i^s} \in I^s$ and for $l_n^1 \leq i_n \leq l_n^2$. For any two such $n$-tuples, we introduce the relation $>_T$ defined by:

$$(\bar{i^s}, i_n') >_T (\bar{i^s}, i_n'') \quad \Longleftrightarrow$$

$$Max\{T(\bar{i^s}-\bar{d}_1'),\ldots, T(\bar{i^s}-\bar{d}_m')\} > Max\{T(\bar{i^s}-\bar{d}_1''),\ldots, T(\bar{i^s}-\bar{d}_m'')\}$$

where $\bar{d}_j'$ and $\bar{d}_j''$ $(j=1,..,m)$ are the data dependence vectors in (3) corresponding to values $i_n'$ and $i_n''$, respectively, of index $i_n$. Obviously, the relation $>_T$ is a partial ordering in $J^n$. Thus, $J^n$ can be decomposed into a number of chains, i.e. subsets whose elements are linearly ordered (relative to each other). Of course, there will be only one chain if the relation $>_T$ is linear to start with. Each chain consists of indexed computations which have to be carried out one after the other in a specific linear ordering. Computations belonging to different chains can be carried out independently. Among the many chain decompositions of the partially ordered set $J^n$, we choose the one in which the computations in a chain are also sorted (in either increasing or decreasing order) according to the index $i_n$. Let us denote by $s$ the number of such chains.

At this point, we are ready to restructure the given algorithm. We partition the computations indexed by $J^n$ into $s$ separate recurrences, each corresponding to a distinct chain. In each recurrence the ordering for index $i_n$ is chosen according to its ordering in the chain. Then, we transform each recurrence into canonic form using the three previous rules: 1) adding missing indices, 2) adding new variables, and 3) renaming old variables. In addition we also add statements between recurrences to correlate variables in distinct recurrences. In fact, a recurrence may use a variable generated in another recurrence. This last step may introduce non-constant data dependencies in the system.

In conclusion, the obtained new specification is expressed as a system of $s$ modules, each module being a recurrence equation in canonic form. Non-constant data dependencies may occur between variables of different modules.

## 3.An application to dynamic programming

Consider the dynamic programming technique applied to such problems as optimal parenthesization, and shortest path. Both problems can be expressed by the recurrence:

$$1 \leq i \leq n; i < j \leq n$$

$$c_{i,j} = \min_{i<k<j} f(c_{i,k}, c_{k,j}) \qquad (5)$$

with initial conditions:

$$c_{i,i+1} = c_i \quad 1 \leq i \leq n$$

for some function $f$. Note that no explicit ordering for the indices $i, j$, and $k$ is specified. If we choose the normal lexicographical ordering for the three indices and we apply some index transformation we can derive a canonic form from the above recurrence; however, the systolic implementation of it will have execution time $O(n^2)$ since it does not overlap the computations of $c_{i,j}$ for different $k$.

Let $I^3=\{(i,j,k) \mid 1 \leq i, j \leq n; i<k<j\}$ be the index set of the recurrence above; and let $I^2=\{(i,j) \mid 1 \leq i, j \leq n\}$ be the index set defined by variable $c$. Each pair $(c, (i, j))$ is associated with a distinct set of data dependencies (here represented in matricial form):

$$D^c_{(i\ j)} = \begin{vmatrix} 0 & \text{i-k} \\ \text{j-k} & 0 \end{vmatrix}$$

or in expanded form:

$$D^c_{(i\ j)} = \begin{vmatrix} 0 & \ldots & 0 & 0 & -1 & -2 & \ldots & \text{i-j+1} \\ \text{j-i+1} & \ldots & 2 & 1 & 0 & 0 & \ldots & 0 \end{vmatrix}$$

The intersection $D^c$ of all the set $D^c_{(i,j)}$ $(1 \leq i, j \leq n)$ is non empty and is given by:

$$D^c_{(i\ j)} = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

We derive a timing function $T : I^2 \rightarrow Z$ compatible with $D^c$. Since $D^c$ contains only constant data dependencies, we can apply the methodology discussed in [11] to determine a linear timing function. Conditions (4) applied to the data dependence vectors in $D^c$ give:

$$T_1 > 0 \quad \text{and} \quad T_2 \leq -1$$

The least integer values that satisfy the above equations are:

$$T_1 = 0 \quad \text{and} \quad T_2 = -1.$$

Thus, the optimal time transformation is:

$$T(i,j) = j - i.$$

Next we consider the partial ordering $>_T$ in $J^3=\{(i,j,k) \mid i,j \textit{ are fixed} \text{ and } i<k<j\}$ defined by:

$$(i,j,k') >_T (i,j,k'') \quad \Longleftrightarrow$$

$$Max\{\, T(i,k'), T(k',j)\} \;>\; Max\{\, T(i,k''), T(k'',j)\}$$

Notice that the minimal elements with respect to $>_T$ are:

$(i, j, (i+j)/2)$ if $i+j$ is even or

$(i, j, (i+j-1)/2)$ and $(i, j, (i+j+1)/2)$ if $i+j$ is odd.

By repeatedly finding minimal elements after removing the previous minimal elements from the set we obtain a decomposition of $J^3$ in the two chains below (here we only write the third component of the index vectors):

{if (i+j) is even}

$(i+j)/2, (i+j)/2-1, ..., i+1;$

$(i+j)/2+1, (i+j)/2+2, ..., j-1.$

{if (i+j) is odd}

$(i+j-1)/2, (i+j-1)/2-1, ..., i+1;$

$(i+j+1)/2, (i+j+1)/2+1, ..., j-1;$

We can now rewrite (5) into a form where the execution ordering of index $k$ is specified according to the above chains. The new specification of the problem consists of two recurrences: the first recurrence is a forward recurrence where the index $k$ varies from $(i+j)/2$ to $i+1$ (or from $(i+j-1)/2$ to $i+1$ if $i+j$ is odd); and the second is a backward recurrence where $k$ varies from $(i+j)/2$ to $j-1$ (or from $(i+j+1)/2$ to $j-1$ if $i+j$ is odd) . To transform each recurrence into canonic form some further manipulation is necessary. We first add missing indices to variables on both sides of (5) and then introduce new recurrence variables to eliminate broadcast of a variable to different destinations. We use different sets of variables for the two recurrences. Each recurrence initializes some of its variables to values generated by the other recurrence. Now equations (5) can be converted into the following system of mutually dependent recurrence equations:

```
        for $i$ :=1 to $n-1$ do $a''_{i,i+1,i+1} := c_{i,i+1}$;  $c_{i,i+1,i+1} := c_{i,i+1}$;
        for $l$ := 2 to $n-1$ do
      for $i$ := 1 to $n$ do begin
            $j := i+l$;
            if $(i+j)$=$even$) then begin
A1:             $k :=(i+j)/2$; $a'_{i,j,k} = a''_{i,j-1,k}$ ;
A2:             if $k= i+1$ then $b'_{i,j,k} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$ ;
                $c'_{i,j,k} := f(a'_{i,j,k}, b'_{i,j,k})$; $c''_{i,j,k} := c'_{i,j,k}$
                end
            else begin    {i+j=odd};
                $k :=(i+j-1)/2$; $a'_{i,j,k} := a'_{i,j-1,k}$ ;
                if $k= i+1$ then $b'_{i,j,k} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$ ;
                $c'_{i,j,k} := f(a'_{i,j,k}, b'_{i,j,k})$
                $k := (i+j+1)/2$;
A3:             if $k := j-1$ then $a''_{i,j,k} := c_{i,j-1,j-1}$ else $a''_{i,j,k} := a''_{i,j-1,k}$ ;
A4:             $b''_{i,j,k} := b'_{i+1,j,k}$ ;
                $c''_{i,j,k} := f(a''_{i,j,k}, b''_{i,j,k})$;
                end
            for $k := \lceil (i+j-1)/2 - 1 \rceil$ downto $i+1$ do begin
                $a'_{i,j,k} := a'_{i,j-1,k}$ ;
                if $k= i+1$ then $b'_{i,j,i+1} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$ ;     module 1
                $c'_{i,j,k} := h(c'_{i,j,k+1}, f(a'_{i,j,k}, b'_{i,j,k}))$;
                end;

            for $k := \lfloor (i+j+1)/2 +1 \rfloor$ to $j-1$ do begin
                if $k= j-1$ then $a''_{i,j,k} := c_{i,j-1,j-1}$ else $a''_{i,j,k} := a''_{i,j-1,k}$ ;        module 2
                $b''_{i,j,k} := b''_{i+1,j,k}$ ;
                $c''_{i,j,k} := h(c''_{i,j,k-1}, f(a''_{i,j,k}, b''_{i,j,k}))$:
                end;
A5:         $c_{i,j,j} := h(c'_{i,j,i+1}, c''_{i,j,j-1})$;
            end.
```

From the above specification we extract distinct sets $D_1$ and $D_2$ of local data dependencies for the two modules.

$$D_1 = \begin{array}{|ccc|} c' & a' & b' \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{array} \qquad D_2 = \begin{array}{|ccc|} c'' & a'' & b'' \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{array}$$

Non-constant dependencies between variables of distinct recurrences are defined by statements A1 to A5 in the algorithm. We will refer to such dependencies as global dependencies.

## 4.Mapping the algorithm into hardware

Once the algorithm has been converted into the new form, the mapping into a VLSI array can be accomplished in two steps:

1) Finding for each individual module in the algorithm representation a separate timing function which is compatible with the local data dependencies and also satisfies the constraints imposed by the global dependencies.

2) Finding a space mapping for each module into the processors of a physical network which is compatible with the timing function and satisfies global constraints.

Assume that each processor of a VLSI array is assigned a label $l \in L^{n-1} \subset Z^{n-1}$ . Furthermore, the connection pattern of the network is described by a set of vectors $\Delta = [\delta_1, \delta_2, \ldots, \delta_s]$ which specify the interconnections among the processors. Precisely, $\delta_i$ is the difference of the integer labels of adjacent cells in the network.

The space mapping is a mapping from the set of computations to the set of processors

$$S : I^n \rightarrow L^{n-1}$$

which maps simultaneous computations into distinct processors. Determining a solution for $S$ is equivalent to find a solution to the diophantine equations:

$$SD = \Delta K \qquad (6)$$

for which the matrix $\begin{bmatrix}\tau \\ S\end{bmatrix}$, $\tau$ being the time function, is nonsingular. In (6) $K$ is an integer matrix with positive elements. The equations may not have any solution or have several solutions. If no feasible solution is found, the design procedure is repeated by starting with a different timing function or else a different physical network. If several solutions are possible, the one which is optimal according to some criterion is chosen. A criterion can be, for example, the minimum number of required processors. These issues are discussed in [3].

Referring again to the dynamic programming algorithm, we first seek linear time transformations $\lambda$ and $\mu$ for module 1 and 2, respectively. Furthermore, let $\sigma$ be the timing function for computation in A5 outside the two modules. The linearity requires that for the dependence vectors of the modules it must be:

$$\lambda\,(\bar{d}) > 0 \quad \text{for } \bar{d} \in D_1 \quad \text{and} \quad \mu\,(\bar{d}) > 0 \quad \text{for } \bar{d} \in D_2$$

that is:

$$\lambda_1 \leq -1 \qquad \lambda_2 \geq 1 \qquad \lambda_3 \leq -1$$
$$\mu_1 \leq -1 \qquad \mu_2 \geq 1 \qquad \mu_3 \geq 1.$$

Constraints specified by A1-A5 lead to the following equations:

$$\begin{gathered} \lambda(i, j, (i+j)/2) > \mu(i, j-1, (i+j)/2) \\ \lambda(i, j, i+1) > \sigma(i+1, j, j) \\ \mu(i, j, j-1) > \sigma(i, j-1, j-1) \\ \mu(i, j, (i+j+1)/2) > \lambda(i+1, j, (i+j+1)/2) \\ \sigma(i, j, j) \geq \max[\lambda(i, j, i+1), \mu(i, j, j-1)] \end{gathered}$$

It easy to check that an optimal solution to the above system is given by:

$$\lambda_1 = -1 \qquad \lambda_2 = 2 \qquad \lambda_3 = -1$$
$$\mu_1 = -2 \qquad \mu_2 = 1 \qquad \mu_3 = 1$$
$$\sigma_1 = -2 \qquad \sigma_2 = 1 \qquad \sigma_3 = 1.$$

Hence:

$$\lambda(i,j,k) = -i+2j-k$$
$$\mu(i,j,k) = -2i+j+k$$
$$\sigma(i,j,j) = -2i+2j.$$

The next step is to find a mapping of the computations indexed by $I^3$ into the processors of a VLSI array. We consider a triangular array labelled by $L^2 = \{(x,y)\} \subset Z^2$. The interconnections among cells are specified by the matrix:

$$\Delta = \begin{vmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \end{vmatrix}$$

The mapping function $S : I^3 \rightarrow L^2$ of the computations in the triangular array is obtained with a construction similar to the one for the timing functions. The same space function is determined for the two modules and is given by:

$$S(i,j,k) = (j,i).$$

The resulting design is identical to the one first introduced in [4]. The systolic array and the action of a cell are depicted in figure 1.

## 5.Conclusions

We have suggested a methodology to assist a human designer into the difficult task of designing a VLSI algorithm starting from a sequential specification of a problem. Among the many possible transformations which can make parallelism explicit in a program the method helps selecting a feasible and optimal sequence of transformations. The methodology appears useful for complex nonnumerical problems for which standard restructuring techniques seem inadequate. The class of obtainable designs is not restricted to designs having constant data flow.

## References

[1] Chen, M., "Synthesizing Systolic Designs ", in *Proc. Int. Symp. on VLSI Technology, Systems and Applications*, Taipei-Taiwan, May 85.

[2] Fortes, J.A.B., Fu, K.S., and Wah, B.W., "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays" Tech. Rep., Electr. Eng., Purdue University.

[3] Fortes, J.A.B. and Moldovan, D.I., "Parallelism Detection and Algorithm Transformation Techniques Useful for VLSI Architecture Design" *J. Parallel Distrib. Comput.*, May 1985.

[4] Guibas, L.J., Kung, H.T., and Thompson, C.D., "Direct VLSI Implementation of Combinatorial Algorithms ", *Proc. of Caltech Conf. on VLSI*, 1979.

[5] Johnsson, L. and Cohen, D., "A Mathematical Approach to Modelling Flow of Data and Control in Computational Networks", in *VLSI Systems and Computations*, H.T. Kung , B. Sproull and G. Steele eds., Computer Science Press, pp. 213-225, 1981

[6] Kung, H.T. and Lin, W., "An Algebra for VLSI Algorithm Design", *Proc. Conf. on Elliptic Problem Solvers*, 1983.

[7] Kunh, R.H., "Transforming Algorithms for Single-Stage and VLSI Architectures", *Workshop on Interconnection Networks for Parallel and Distributed Processing*, 1980.

[8] Lam, M. and Mostow, J., "A Transformational Model of VLSI Systolic Design", *Computer*, pp. 42-52, 1985.

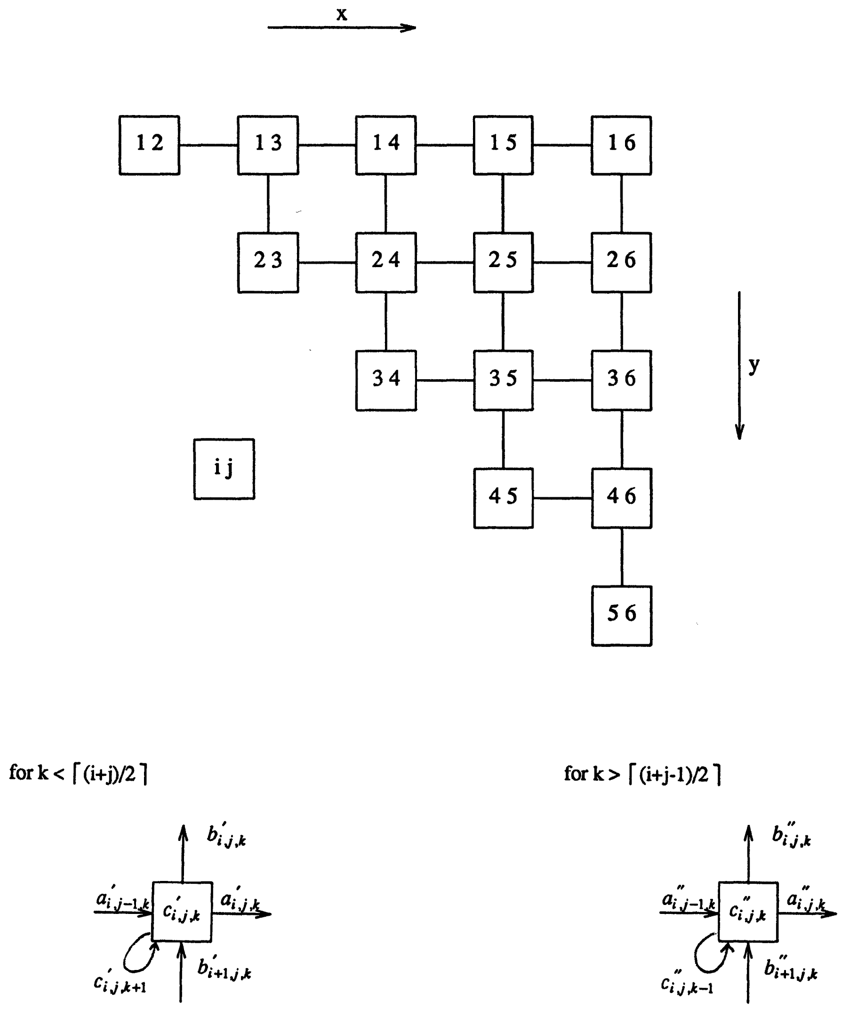
[9] Li, G. and Wah, B., "The design of Optimal Systolic Arrays", *IEEE Trans. on Computers*, C-34, pp. 66-77, 1985.

[10] Melhem, R. and Guerra, C., "The Application of a Sequence Notation to the Design of Systolic Computations", *Techn. Rep. 568*, Dept. Comp. Sc., Purdue University.

[11] Moldovan, D., "On the Analysis and Synthesis of VLSI Algorithms", *IEEE Trans. on Computers*, C-31, pp. 1121-1126, 1982.

[12] Moldovan, D., "On the Design of Algorithms for VLSI Systolic Arrays", *Proc. IEEE*, vol. 71, pp. 113-120, Jan 1983.

[13] Quinton, P., "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations ", *Proc. 11-th Annual Symp. on Computer Architecture*, pp. 208-214, 1984.



for k < ⌈(i+j)/2⌉

for k > ⌈(i+j-1)/2⌉

Fig. 1 A systolic network for dynamic programming

# VERIFICATION OF SYSTOLIC ARRAYS:
# A stream functional approach

Sanjay V Rajopadhye
Department of Computer Science
University of Utah
Salt Lake City, Ut 84112

Prakash Panangaden
Department of Computer Science
Cornell University
Ithaca, NY 14853

## Abstract

We illustrate that the verification of systolic architectures can be carried out using techniques developed in the context of verification of programs. This is achieved by partitioning the original problem into three parts, namely proving the correctness of the data representation, of the individual processing elements, and of a composition of a number of such processors. By expressing a processing element as a function on a stream of data we are able to utilize standard proof techniques from programming language theory. This decomposition leads to relatively straightforward proofs of the properties of the systolic architecture. We illustrate the techniques via a substantial example, namely the proof of the correctness of a linear-time systolic architecture for computing the gcd of polynomials. Although this architecture has been designed a few years ago, a formal proof of correctness has not hitherto appeared in the literature.

## I Introduction

In the last few years there has been much interest in the use of formal techniques for the design and analysis of VLSI circuits. Special formalisms have been invented for expressing transformations and specifications of such circuits. One important aspect of the analysis of circuits is the correctness problem. In this paper we address the correctness problem for an important class of parallel architectures, namely systolic arrays. This problem is decomposed in such a manner that formal techniques from programming language theory can be used in a straightforward manner. The particular techniques we use were invented in the context of proving the correctness of data representations [3] and semantics of functional languages [4].

The individual processors in systolic arrays act synchronously (at least they can be conceptually viewed thus, even though the actual implementation may be asynchronous) on streams of data tokens. As a result their operation can be divided into a three-step *read-execute-write* cycle, where the execute phase does not branch on the *availability* of data values. It has been shown [4] that under such conditions the action of such a process can be described completely by a *stream function*. The action of the entire network is therefore merely the composition of these individual functions. The verification problem is thus reduced to the problem of verifying that the composition of a caerain number copies of a function computes as specified. We shall illustrate the technique by proving the correctness of a linear systolic architecture for computing the greatest common divisor (GCD) of two polynomials [1]. Due to lack of space, we present here only a brief outline of the proof. The interested reader is referred to [8] for full details.

The rest of this paper is organized as follows. In the following section (Sec II) we briefly describe the architecture of the chip. We also express each processor in the array as a stream function, provide a representation mapping from the domain on which the chip operates(i.e. streams of numbers) to the domain of polynomials, and give a representation invariant. The following section, (Sec III) then formulates the actual verificaton problem as three subproblems and addresses each in turn. Finally in Sec IV we compare our technique with other existing techniques for verifying systolic architectures.

## II Description of the gcd chip

The basis of the gcd-chip is Euclid's algorithm which depends on the following lemma. If A and B are two non-zero polynomials as shown below,

$$\begin{aligned} A &= a_i x^i + \ldots + a_1 x + a_0 \\ B &= b_j x^j + \ldots + b_1 x + b_0 \end{aligned}$$

then their gcd satisfies the following invariant.

$$\mathrm{gcd}(A,\ B) = \begin{cases} \mathrm{gcd}((A - [a_i/b_j]x^{i-j} * B),\ B) & \text{if } i \geq j \\ \mathrm{gcd}(A,\ (B - [b_j/a_i]x^{j-i} * A)) & \text{if } j \geq i \end{cases}$$

This serves as the basis for a gcd-preserving transformation that also reduces the degree of one of the polynomials by at least one. It is successively applied to the original pair of polynomials until one of them is reduced to the zero polynomial, at which point the other one is the desired gcd. Let us view a polynomial as a sequence of coefficient-power pairs in decreasing order of power. The first term has a nonzero coefficient, and thus its power is the degree of the polynomial. If $p_1$ and $p_2$ are two such sequences we can express Euclid's

This research was sponsored in part by the Office of the Naval Research under contract N00014-83-K-0317, and in part by a University of Utah fellowship. Travel and other support was obtained from an IBM faculty development grant.

algorithm as in Fig II-1. In our proof of correctness we shall show that the gcd architecture computes exactly this function.

```
gcd(p1, p2) =
if p1 = [0, 0] then p2
elseif p2 = [0, 0] then p1
elseif deg(p1) ≥ deg(p2) then
  gcd((p1-[Q(p1,p2), (deg(p1)-deg(p2))]*p2), p2)
else
  gcd(p1, (p2-[Q(p2,p1), (deg(p2)-deg(p1))]*p1))

where
  deg(p) = second(first(p)) is the degree of p;

  Q(p1, p2) = first(first(p1))/first(first(p2))
is the ratio of the leading coefs of p1 and p2;
and '-' and '*' are operations on polynomials;
first (and second) is a function that
returns the first (second) element of a list.
```

**Figure II-1:** *Euclid's algorithm*

The Kung-Brent architecture consists of 2*m-1 processors cascaded linearly. Each processor has three data input (and output) channels, and one control line `start`. Two of the data channels, `a` and `b` carry streams of numbers, and the third, `d` carries a single integer (aligned with the signal on the `start` line). The processor begins a new cycle when it receives a start bit on the `start` line. Depending on whether or not `d` is non-negative the processor determines whether stream A or stream B is to be reduced (say `A`). It then computes the coefficient `q` (= $a_1/b_1$, while delaying the other stream `B` by one time unit. After this it cycles through the rest of the input stream (i.e. until a new start bit appears) replacing each `a` element by $a_i-q*b_i$, and passing stream `B` unaltered (except for the delay). We shall consider the processor operating on only one pair of polynomials. From the description of each processor, the equivalent stream function (called *reduce*, see Fig II-2) can easily be determined, using the technique described in [7].

```
reduce(sA, sB, d) =
if first(sA)=0 or (first(sB)≠0 ∧ d≥0) then
 [f(rest(sA),rest(sB),first(sA)/first(sB))^0,sB,d-1]
else
 [sA,f(rest(sB),rest(sA),first(sB)/first(sA))^0,d+1]

where f is defined as
  f(s1, s2, q) = if s1 = [] then []
else
  (first(s1)-q*first(s2))^f(rest(s1), rest(s2), q)
  Note: ^ is an infix cons operator
```

**Figure II-2:** *The function reduce*

In order to prove the correctness of the gcd chip we must now demonstrate that the function *reduce*, applied 2*m-1 times to input streams of length m yields the gcd of the polynomials that the input streams represent. Before formulating this problem we must set up some basics. Firstly, we must give a mapping, *rep* from the data-domain on which the chip (i.e. *reduce*) operates (treams) to the abstract domain of interest (polynomials). Secondly, we must specify a representation invariant, *R* that must be satisfied by elements of the data domain. This is done as follows. We define *rep* in terms of another function *prep* which takes an integer *d* and a single stream of numbers *s* (which has no leading zeroes) and yields a polynomial.

```
prep(s, d) =
    if d = 0 then
        if s=[] then [0,0] else [first(s), 0]
    elseif s=[] then [0,d]^prep(s, d-1)
    else [first(s), d]^prep(rest(s), d-1)
```

Using this we now define *rep* (Fig II-3). `AllZero(s)` is a predicate asserting that all elements in s are zero, *StripTrailingZeroes*(s) is a function that removes all trailing zeroes from a stream s (provided `AllZero(s)` is false) and *Final*(i, s) is the stream consisting of the last i elements of a stream s.

```
rep(sA, sB, d) =
if AllZero(sA) then
  [[0,0], prep(StripTrailingZeroes(sB),
                    length(StripTrailingZeroes(sB))-1)]
elseif AllZero(sB) then
  [prep(StripTrailingZeroes(sA),
          length(StripTrailingZeroes(sA))-1), [0,0]]
elseif first(sA)=0 then rep(rest(sA), sB, d-1)
elseif first(sB)=0 then rep(sA, rest(sB), d+1)
else
  rep'(StripTrailingZeroes(sA),
          StripTrailingZeroes(sB), d)

where
rep'(s1, s2, d)=
if d > length(s1) - length(s2) then
  [prep(s1, length(s2)+d-1),
    prep(s2, length(s2)-1)]
elseif d = length(s1) - length(s2) then
  [prep(s1, length(s1)-1), prep(s2, length(s1)-1)]
elseif d < length(s1) - length(s2) then
  [prep(s1, length(s1)-1),
    prep(s2, length(s1)-d-1)]
```

**Figure II-3:** *The representation mapping, rep*

The representation invariant consists of four conjuncts $R_1$, $R_2$, $R_3$ and $R_4$ which are defined as follows.

```
R1 ≡ length(sA) = length(sB)              (= n, say)
R2 ≡ (first(sA)≠0 ∨ first(sB)≠0)
#i(R3) ≡ d≥0 ⇒
    (n≥d ∧ AllZero(Final(d, sB))) ∨ AllZero(sB)
R4 ≡ d<0 ⇒
   (n≥-d ∧ AllZero(Final(-d, sA))) ∨ AllZero(sA)
```

## III Problem decomposition and proof

With this machinery in place, the actual proof of correctness is achieved by demonstrating that *reduce* satisfies the following conditions.

- It must preserve the invariant $R$ on every invocation, i.e.

$$R(s_A, s_B, d) \Rightarrow R(reduce(s_A, s_B, d)) \qquad (1)$$

- It must preserve the gcd of the polynomials obtained by applying *rep* to its input and output streams, i.e.

$$gcd[rep(reduce(s_A, s_B, d))] = gcd[rep(s_A,s_B,d)] \qquad (2)$$

- The computation defined by *reduce* "terminates", i.e. if the length of the original streams is m the exactly 2*m-1 applications of *reduce* (denoted by $reduce^{2*m-1}(s_A, s_B, d)$) cause one of the polynomials to become the zero polynomial[1]. This means that

$$first(rep(reduce^{2*m-1}(s_A, s_B, d))) = [0,0]$$
$$\vee\ second(rep(reduce^{2*m-1}(s_A, s_B, d))) = [0,0] \qquad (3)$$

The proof of Theorem 1 involves fairly straightforward manipulation after substituting the definition of *reduce* in the equation. Next, proving that the function *reduce* preserves the gcd of the polynomials involves a case by case analysis on the first elements in the streams $s_A$ and $s_B$. Although the prof is somewhat long it is fairly simple conceptually. Finally, to prove Theorem 3 we also perform a similar casewise analysis. We first show that if one of the polynomials that $[s_A, s_B, d]$ represent is the zero polynomial, then further application of *reduce* leaves the streams (and hence the polynomials they represent) unchanged. Then we show that associated with each $[s_A, s_B, d]$ is an integer[2], N which decreases by exactly one on every application of *reduce*. When N becomes zero one of the polynomials is the zero polynomial. The interested reader is referred to [8] for full details.

## IV Conclusions

We have presented by means of an example how well-known techniques from programming language theory can be profitably applied to the verification of systolic architectures. An additional contribution of this paper is the proof of correctness of a hitherto unproved systolic array. Other researchers have addressed the problem of reasoning about systolic architectures [2, 5, 6]. In Chen's approach space-time recursion equations are set up and their fixed point [9] yields the function describing the network. By using our approach we do not need to explicitly reason about the time component. Successive time instants are represented by successive appearances of tokens on a stream. Melhem [5] describes a formalism involving graphs with colored arcs which is very similar to ours. The processing elements are also modelled as (essentially) stream functions, though they are referred to as operators. The overall network behavior is obtained by solving a system of *difference equations.* Purushothaman [6] also has a similar approach, where the verification problem is also reduced to the solution of recurrence equations. In both these methods the equations are set up using the token-wise behavior of the individual processing elements, and standard techniques for manipulating functions are not exploited. Their use of difference/recurrence equations implies that both, the data representation and processor behavior issues have to be handled simultaneously. These issues are cleanly separated in our approach.

## Acknowledgments

The authors would like to thank Kevin Karplus for helpful criticism of an early draft and Richard Fujimoto and Ashok Samal for comments on the presentation.

## References

**1.** Brent, R. P. and Kung, H. T. Systolic VLSI Arrays for Linear-Time GCD Computation. VLSI 83, Aug, 1983, pp. 145:154.

**2.** Chen, Marina C. *Space-Time Algorithms: Semantics and Methodology.* Ph.D. Th., California Institute of Technology, Pasadena, CA, May 1983.

**3.** Hoare, C. A. R. "Proof of Correctness of Data Representations". *Acta Informatica 1* (1972), 271-281.

**4.** Kahn, Gilles. The Semantics of a Simple Language for Parallel Processing. Proceedings of IFIP, IFIP, Aug, 1974, pp. 471-475.

**5.** Melhem, Rami G. and Rheinboldt, Werner C. "A Mathematical Model for the Verification of Systolic Networks". *SIAM Journal of Computing 13*, 3 (August 1984), 541-565.

**6.** Purushothaman, S. *Verification of Systolic Architectures.* Ph.D. Th., University of Utah, Salt Lake City, Utah 84112, December 1985.

**7.** Rajopadhye, Sanjay V. A formal basis for synthesizing systolic arrays: PhD Thesis proposal. UUCS-84-010, Universityof Utah, Computer Science Department, November, 1984.

**8.** Rajopadhye, Sanjay V., Panangaden, Prakash. Verification of Systolic Arrays: A stream functional approach. UUCS-85-001, University of Utah, Salt Lake City, Ut 84112, March, 1985.

**9.** Joseph Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory.* MIT Press, 1977.

[1] Note that this formulation of termination is slightly different from the termination in the classical sense. This is because our hardware architecture has a constant number of processors, and thus applies the function *reduce* exactly 2m+1 times, regardless of whether it is redundant or not.

[2] N represents the sum of the number of leading zeroes in one of the streams and the degrees of the two polynomials

# SYNTHESIZING VLSI ARCHITECTURES: DYNAMIC PROGRAMMING SOLVER

(preliminary version)

**Marina C. Chen**

**Department of Computer Science**
**Yale University**
**New Haven, CT 06520**
chen-marina@yale

**Abstract** In this paper, a design methodology for synthesizing efficient parallel algorithms and architectures from problem definitions specified in the language **Crystal** is presented. First, transformations on a problem definition to programs with *bounded order* and *bounded degree* are introduced. This stage of transformations aims to reduce the dominant costs in the underlying hardware which might be incurred by a direct parallel implementation of the problem definition. At the second stage, pipelining is automatically incorporated into an algorithm so that the hardware resources can be most effectively utilized. The methodology allows the derivation of systolic algorithms and architectures in a unified framework based on **Crystal.** Since **Crystal** is a general purpose language for parallel proramming, new synthesis techniques and insights into problems can be integrated readily within the existing framework.

## 1. Introduction

VLSI technology provides a natural medium for parallel processing, from the level of switching elements, to logic gates, to functional and control units, and to architectures and algorithms. To expoit its potential fully, parallelism must be used at increasingly higher levels. Design automation, consequently, must go beyond the level of taking as givens architectural level specifications, and should head towards compiling or synthesizing efficient parallel algorithms and architectures. The technology, on the other hand, constrains the way in which parallel computation can be organized. Dominant costs at the technological level often have profound implications in the designs at algorithmic and architectural levels. Clearly, to achieve an efficient design, one must take advantage of what the technology can offer, and minimize the costs associated with the constraints it imposes upon the design.

In this paper, we describe a design methodology that takes into account the dominant costs in the underlying technology and minimizes such costs to yield efficient parallel algorithms and architectures. The design process starts with a problem definition specified in the language **Crystal** [4]. The definition is interpreted as a parallel algorithm, however naive and inefficient it might be. From this naive algorithm, the dominating costs it might incur in the underlying parallel hardware, such as that of communications and fan-ins and fan-outs, are examined. We then improve the algorithm by a series of transformations that results in a *bounded order* and **bounded degree** program which has reduced hardware costs. Once such a program is obtained, it goes through another stage of transformation, called *space-time mapping*, by which pipelining is automatically incorporated into the algorithm and the hardware resources are fully utilized. Algorithms are classified as either uniform or non-uniform because space-time mapping for uniform algorithms can be obtained by an expedient procedure.

Throughout this paper, the methodology is applied to the dynamic programming problem, which turns out to be a uniform algorithm. The well-known systolic algorithm of Guibas, Kung and Thompson [7] and others are generated as a result. The rest of this paper is organized as follows: in Section 2, a mathematical definition of dynamic programming is given. In Section 3, its interpretation as a parallel algorithm is described. In Section 4, transformations for reducing the number of fan-ins and fan-outs are presented. In Section 5, transformations that reduce long-range communications are illustrated. In Section 6, space-time mapping for uniform algorithm is described. A general inductive method for finding space-time mapping for non-uniform algorithms can be found in [5]. In Section 7, optimization of control signals is illustrated. Finally, some related work is discussed in Section 8.

## 2. A General Definition of Dynamic Programming

A large class of optimization problems can be solved by dynamic programming. The definition of such problems can be posed in a general form where $C(i,j)$ is some cost function which is to be minimized.

$$C(i,j) = \begin{cases} s > 0 \rightarrow \min\limits_{i<k<j} \{F(C(i,k), C(k,j))\} \\ \qquad \text{where } F \text{ is some function on the costs} \\ s = 0 \rightarrow C_i \text{ for some individual cost} \end{cases}$$

$$\text{where } s = j - i - 1, \tag{2.1}$$

and $i$ and $j$ are integers in the range $0 < i < j \leq n$, for some constant $n$.

## 3. Parallel Interpretation of Algorithms

The straightforward definition given in Equation (2.1) is

0190-3918/86/0000/0776 $01.00 © 1986 IEEE

in the form of a **Crystal** recursion equation with *recursion variables* $i$ and $j$. It can be interpreted as a parallel algorithm as follows: Each pair $(i,j)$ in Equation (2.1) is interpreted as a process in an ensemble of parallel processes denoted by the set $P_1 \overset{\text{def}}{=} \{(i,j) : 0 < i < j \leq n\}$. The local processing at each process $(i,j)$ is, in this example, the function $\min_{i<k<j}\{F(x_k, y_k)\}$ that takes $j - i - 1$ pairs of arguments. The communication between processes is specified, for instance, by the pairs $(i,k)$ and $(k,j)$ appearing on the right-hand side of Equation (2.1) to indicate that the computation of $C(i,j)$ at process $(i,j)$ needs the results from processes $(i,k)$ and $(k,j)$.

The ensemble of processes and its data flow can be depicted by a DAG (Directed Acyclic Graph) as shown in Figure 1. It consists of nodes, where each node corresponds to a process $(i,j)$ in $P_1$, and directed edges, each of which comes out of a node whose corresponding process appears on the right-hand side of Equation (2.1), and goes into a node whose corresponding process appears on the left-hand side of the equation. The directed edges of the DAG define the data dependency relation of the algorithm. We say that a process **u** immediately precedes **v** ($\mathbf{u} \prec \mathbf{v}$), or **v** immediately depends on **u** ($\mathbf{v} \succ \mathbf{u}$), if there is a directed edge from **u** to **v**. The transitive closure "$\overset{*}{\prec}$" of this relation, called "precede", is a partial order, and there is no infinite decreasing chain from any node **v**, nor is there an infinite number of processes that immediately precede **v**. Those nodes that have no incoming edges are called *sources.*

Processes are parallel in nature. A computation starts at the sources which are properly initialized, and is followed by other processes each of which starts execution when all of its required inputs, or dependent data become available. The parallel execution of the naive dynamic programming definition starts with all processes such that $s = 0$, followed by processes with increasing $s$ as illustrated in Figure 1.

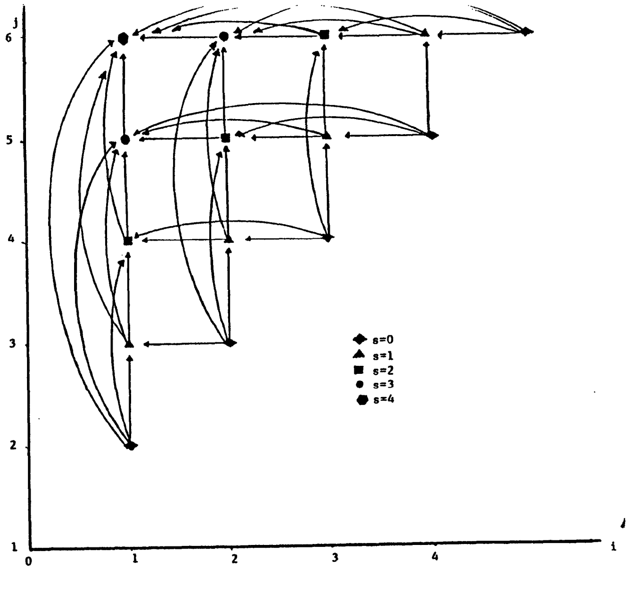


**Figure 1:** The DAG describing the data dependency of dynamic programming.

An immediate, very naive implementation of the parallel interpretation uses one processor for each process, and altogether $O(n^2)$ parallel processors are needed. The number of time steps needed to compute the result is at best $n - 1$ since any $C(i,j)$ cannot be computed until $C(i,j-1)$ and $C(i+1,j)$ are computed.

## 4. Problems with Large Number of Fan-ins and Fan-outs

Due to the inherent physical constraints imposed by the driving capability of communication channels, power consumptions, heat dissipations, memory bandwidth, etc., it is reasonable to assume that data can only be sent or received simultaneously to or from a small number of destinations or sources. Putting such constraints in algorithmic terms: the number of "fan-ins" and "fan-outs" of a datum must be small, where *fan-in degree* of a datum is defined to be the number of data items on which it depends, and *fan-out degree* is the number of data items dependent upon it. If what we are interested in are practical and efficient algorithms, the fan-in and fan-out degrees should be taken into account in measuring the complexity.

It can be easily seen that the fan-in degree of $C(i,j)$, which appears on the left-hand side of Equation (2.1), is $2s$, where $s \overset{\text{def}}{=} j - i - 1$. Conversely, for any value appearing as $C(i,k)$ on the right-hand side of Equation 2.1, that value is needed by $C(i,j)$ for $j = k+1, \ldots, n$ (there are $n - k$ such terms). The same value also appears as $C(k,j)$ in the equation and it is needed by processes $(i,j)$ for $i = 1, \ldots, k-1$ (there are $k - 1$ such terms). Since both $s$ and $k$ grow with the problem size $n$, the number of fan-ins and fan-outs therefore grows linearly with the size of the problem.

To avoid the high cost incurred at the hardware level, the large fan-in and fan-out degrees must be reduced. In the following transformations, the original fan-in and fan-out degrees are reduced from $O(n)$ to a constant while the complexity of the original algorithm is maintained the same as before.

### 4.1. Reducing fan-out degrees

The idea used in reducing the fan-out degree is quite simple:

**Proposition 4.1.** *Let $z(l)$ for $u \leq l \leq v$ where $u$ and $v$ are integers and $u < v$ be $v - u + 1$ variables to which a value $x$ shall be assigned, i.e., $z(l) = x$ for $u \leq l \leq v$ (where the fan-out degree of $x$ is $v - u + 1$). Then the assignments of these variables can be performed instead by*

$$z(l) = \begin{cases} l = u \rightarrow x \\ l > u \rightarrow z(l-1) \end{cases} \qquad (4.1)$$

*or by*

$$z(l) = \begin{cases} l = v \rightarrow x \\ l < v \rightarrow z(l+1) \end{cases} \qquad (4.2)$$

*for $u \leq l \leq v$*

*where the fan-out degrees of $x$ and $z(l)$ for $u \leq l \leq v$ are 1.*

To decrease the fan-out degree of $C(i,k)$, let the variables at processes $(i,j)$ for $k \leq j \leq n$ to which $C(i,k)$ will be assigned be denoted by $a(i,j,k)$. Similarly, let the variables at process $(i,j)$ for $1 \leq k < i$ to which $C(k,j)$ is to be assigned be denoted by $b(i,j,k)$. Applying Equation (4.1) of the proposition to $a(i,j,k)$ and Equation (4.2) to $b(i,j,k)$, we obtain

$$
\begin{aligned}
a(i,j,k) &= \begin{cases} j = k \to C(i,k) \\ j > k \to a(i,j-1,k) \end{cases} \\
b(i,j,k) &= \begin{cases} i = k \to C(k,j) \\ i < k \to b(i+1,j,k) \end{cases} \qquad (4.3) \\
&\text{for } 1 \le i < k < j \le n.
\end{aligned}
$$

Now the $n-k$ fan-out degree of $C(i,k)$ to all processes $(i,j)$, where $k < j \le n$, is reduced to 1, and so is the $k-1$ fan-out degree of $C(k,j)$ to all processes $(i,j)$ for $1 \le i < k$. We may now replace the high fan-out degree values $C(i,k)$ and $C(k,j)$ on the right-hand side of Equation (2.1) by the low fan-out degree values $a(i,j,k)$ and $b(i,j,k)$:

$$
C(i,j) = \begin{cases} s > 0 \to \min\limits_{i<k<j} \{F(a(i,j,k), b(i,k,j))\} \\ s = 0 \to C_i. \end{cases} \qquad (4.4)
$$

### 4.2. Reducing the fan-in degrees

The reduction of fan-in degree of a value computed by a process relies on the associtivity of the operation being computed.

**Definition 4.1.** An operation "$\bigoplus$", where $z = \bigoplus_{u<l<v} x(l)$, on $v-u-1$ arguments, is associative if it can be replaced by the composition of the following sequence of binary associative opertions "$\oplus$" on variables $z(l)$, $u < l < v$:

$$
z(l) = \begin{cases} l = u+1 \to x(u+1) \\ u+1 < l < v \to z(l-1) \oplus x(l) \end{cases} \qquad (4.5)
$$

and $z = z_{v-1}$.

Similarly, the same operation can be achieved by a different sequence using both binary and ternary associative operations.

**Proposition 4.2.** *If "$\bigoplus$" is an associative operation on $v-u-1$ arguments $x(l)$ for $u < l < v$, where $z = \bigoplus_{u<l<v} x(l)$, then it can be replaced by a sequence of binary and ternary operations on variables $z(l)$, for $m < l \le v$ of the following form:*

$$
z(l) = \begin{cases} l = m \to x(m) \oplus x(u+v-m) \\ m < l < v \to z(l-1) \oplus x(l) \oplus x(u+v-l) \\ l = v \to z(l-1) \end{cases} \qquad (4.6)
$$

*where* $m = \lceil \frac{v+u}{2} \rceil$

*and* $z = z_v$.

Since the operation "min" for computing $C(i,j)$ is associative, let the high fan-in degree value $C(i,j)$ at process $(i,j)$ be computed instead by the sequence $c(i,j,k)$ for $m < k < j$, where $m \overset{\text{def}}{=} \lceil \frac{i+j}{2} \rceil$. Now apply Proposition 4.2 to Equation (4.4), and it is transformed to

$$
c(i,j,k) = \begin{cases} s = 0 \to C_i \\ s > 0 \to \\ \quad \begin{cases} k = m \to \min(F(a(i,j,k), b(i,j,k)), \\ \qquad\qquad F(a(i,j,i+j-k), b(i,j,i+j-k)) \\ m < k < j \to \\ \qquad \min(c(i,j,k-1), F(a(i,j,k), b(i,j,k)), \\ \qquad\qquad F(a(i,j,i+j-k), b(i,j,i+j-k)) \\ k = j \to c(i,j,k-1), \end{cases} \end{cases} \qquad (4.7)
$$

and $C(i,j) = c(i,j,j)$ for all $(i,j) \in P_1$.

The application of Proposition 4.2 is motivated by the following: Due to the data dependency, the value $F(a(i,j,k), b(i,j,k))$ for $i < k < j$ cannot be computed before at least $max(k-i, j-k)$ time steps. A pair with the least delay is the one with $k = \lfloor \frac{i+j}{2} \rfloor$ and/or $k = \lceil \frac{i+j}{2} \rceil$, where the delay is about half of the interval size. Consequently, the most efficient sequence of binary or ternary operations will start with this pair, and will be followed by other pairs in the order of increasing amount of delay.

## 5. Communcations and Bounded-order Recursion Equations

The following two equations

$$
\begin{aligned}
a(i,j,k) &= \begin{cases} j = k \to c(i,k,k) \\ j > k \to a(i,j-1,k) \end{cases} \\
b(i,j,k) &= \begin{cases} i = k \to c(k,j,j) \\ i < k \to b(i+1,j,k) \end{cases} \qquad (5.1) \\
&\text{for } 1 \le i < k < j \le n,
\end{aligned}
$$

are obtained from Equation (4.3) by substituting $C(i,k)$ with $c(i,k,k)$, and substituting $C(k,j)$ with $c(k,j,j)$. The system of

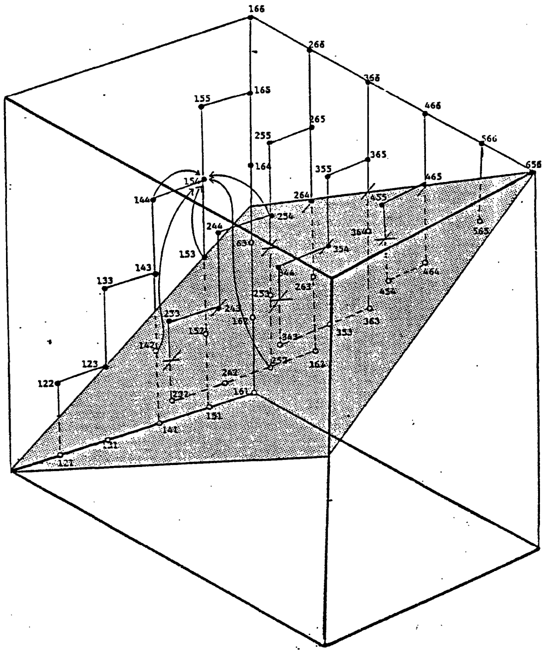


**Figure 2:** Process structure $P_2$ in which the fan-in and fan-out degreess are constant, but in which there are still long range communications

recursion equations consisting of Equations (5.1) and Equation (4.7) is a new algorithm for dynamic programming. The new ensemble of processes $P_2 \overset{\text{def}}{=} \{(i,j,k) : 0 < i < k < j \leq n\}$ is now three dimensional, as shown in Figure 2. The added one dimension and the increased number of processes are for the purpose of decreasing fan-in/fan-out degrees. A remaining criterion for a good parallel algorithm is that all communications between processes defined by a system of recursion equations should also be local. A few definitions are called for:

**Definition 5.1.** The path length between two processes $\mathbf{v} \overset{\text{def}}{=} (v_1, \ldots, v_m)$ and $\mathbf{u} \overset{\text{def}}{=} (u_1, \ldots, u_m)$ is defined to be $|v_1 - u_1| + \cdots + |v_m - u_m|$.

**Definition 5.2.** A communication between two processes $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u} \prec \mathbf{v}$, is local if their path length is bounded by a fixed constant.

**Definition 5.3.** The *order* of a system of recursion equations is defined to be the maximum path length between any pair of processes $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u} \prec \mathbf{v}$.

Similar to the high fan-in and fan-out degrees, the extra delay introduced by the communications of the high-order terms in the equations can undermine the efficiency of an algorithm. We therefore require that the order of a system of equations be either constant or grow very slowly with respect to the problem size.

### 5.1. Localizing communications by domain contraction

Note that Equation (4.7) contains terms $a(i,j,i+j-k)$ and $b(i,i+j-k,j)$, which in turn immediately depends on processes $a(i,j-1,i+j-1-k)$ and $b(i+1,j,i+1+j-k)$ by Equations (4.3). The order of the system of equations is therefore $2(|k+1-\frac{i+j}{2}|)$ for $i < k < j$, which grows with the problem size $n$. To eliminate the high order terms, further transformations are required.

The path length $2(|k+1-\frac{i+j}{2}|)$ between two communicating processes is a linear function of $i$, $j$ and $k$. The goal is to transform this linear function into a constant one. The most obvious choice is to set $k = \frac{i+j}{2}$, in which case the path length becomes 2. Geometrically, we can see that the coordinates of the high order terms above are symmetric to those of the low-order terms $a(i,j,k)$ and $b(i,j,k)$, with respect to the plane $k = (i+j)/2$, shown in Figure 2 as a shaded plane. The algebraic transformation now corresponds to the contraction of the original domain of processes by one half along the plane, so that $(i,j,k)$ and $(i,j,i+j-k)$ become a single process $(i,j,k)$.

Let processes $(i,j,k)$, such that $m \leq k \leq j$, be referred to as being in the upper half of the process structure $P_2$. Similarly, let those processes, such that $i \leq k \leq, m'$ be referred to as being in the lower half of $P_2$.

Let the new process structure be the upper half of $P_2$, i.e., $P_3 \overset{\text{def}}{=} \{(i,j,k) : 0 < i < j \leq n \text{ and } m \leq k \leq j\}$, as shown in Figure 3. The data streams $a$ and $b$ of those processes originally residing in the lower half must be given new names so that they can be differentiated from those of the processes originally in the upper half. To achieve this, let $d$ and $e$ be new data streams to replace $a$ and $b$ for the processes of the lower half as shown in Figure 3.

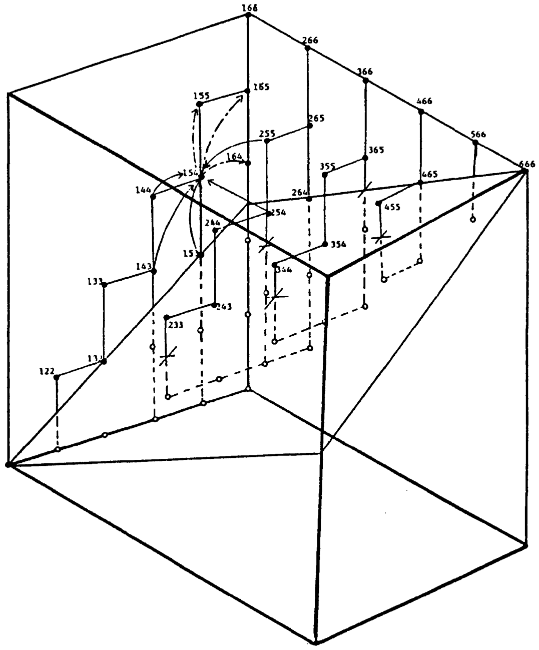


**Figure 3:** Process structure $P_3$, in which all data dependencies are local

### 5.2. Algebraic manipulation to obtain low order equations

First, Equation (5.1), defined for $(i,j,k)$, $i \leq k \leq j$, is now split into two set of equations, one for the case $m \overset{\text{def}}{=} \lceil \frac{i+j}{2} \rceil \leq k < j$ where $(i,j,k)$ resides in the upper half of $P_2$ and the other for the case $i < k \leq m' \overset{\text{def}}{=} \lfloor \frac{i+j}{2} \rfloor$ where it resides in the lower half. Since, for those processes $(i,j,k)$ that reside in the lower half, $a$ and $b$ will be renamed by $d$ and $e$, we must be aware of the case when a process $(i,j,k)$ is in the lower half of $P_2$ while process $(i,j-1,k)$ might be in the upper half, such as in the case when $k = m'$ and $i+j$ is even (or $s \overset{\text{def}}{=} j-i-1$ is odd). Another case to watch for is when $(i,j,k)$ is in the upper half while $(i+1,j,k)$ might be in the lower half; it occurs when $k = m$ and $i+j$ is even (or $s$ is odd). Two equations in (5.1) are now split into four equations, where the above two cases are singled out:

upper half:

$$a(i,j,k) = \begin{cases} j = k \rightarrow c(i,k,k) \\ m \leq k < j \rightarrow a(i,j-1,k). \end{cases}$$

lower half:

$$a(i,j,k) = \begin{cases} (k = m') \wedge (odd(s))) \rightarrow a(i,j-1,k) \\ \qquad\text{(upper half)} \\ (k = m') \wedge (even(s))) \rightarrow a(i,j-1,k) \\ i < k < m' \rightarrow a(i,j-1,k). \end{cases} \tag{5.2}$$

upper half:

$$b(i,j,k) = \begin{cases} (k = m) \wedge (odd(s)) \rightarrow b(i+1,j,k) \\ \qquad\text{(lower half)} \\ (k = m) \wedge (even(s)) \rightarrow b(i+1,j,k) \\ m < k < j \rightarrow b(i+1,j,k). \end{cases}$$

lower half:

$$b(i,j,k) = \begin{cases} k = i \to c(k,j,j) \\ i < k \le m' \to b(i+1,j,k). \end{cases}$$

Next, every $a(i,j,k)$ in the lower half is replaced by $d(i,j,i+j-k)$, where $(i,j,i+j-k)$ is now the upper half. Similarly, every $b(i,j,k)$ in the lower half is replaced by $e(i,j,i+j-k)$.

$(i,j,k)$ lower half

$$d(i,j,i+j-k) = \begin{cases} (k = m') \wedge (odd(s))) \to a(i,j-1,k) \\ \qquad \text{(upper half)} \\ (k = m') \wedge (even(s))) \to \\ \quad d(i,j-1,i+(j-1)-k) \\ i < k < m' \to d(i,j-1,i+(j-1)-k). \end{cases}$$

$(i,j,k)$ lower half

$$e(i,j,i+j-k) = \begin{cases} k = i \to c(k,j,j) \\ i < k \le m' \to e(i+1,j,(i+1)+j-k). \end{cases} \tag{5.3}$$

Finally, a "substitution of variable" is performed on the above two equations, which replaces $i+j-k$ by $k'$:

$$d(i,j,k') = \begin{cases} (k' = m) \wedge (odd(s))) \to a(i,j-1,i+j-k') \\ (k' = m) \wedge (even(s))) \to d(i,j-1,k'-1) \\ m < k' < j \to d(i,j-1,k'-1). \end{cases}$$

$$e(i,j,k') = \begin{cases} k' = j \to c(i,j,j) \\ m \le k' < j \to e(i+1,j,k'+1). \end{cases} \tag{5.4}$$

Since $k'$ is a bounded variable, replace it by $k$ throughout Equations (5.4).

### 5.3. Resulting System of Equations

The resulting algorithm is the following system of recursion equations:

$$a(i,j,k) = \begin{cases} j = k \to c(i,k,k) \\ m \le k < j \to a(i,j-1,k) \end{cases}$$

$$d(i,j,k) = \begin{cases} (k = m) \wedge (odd(s))) \to a(i,j-1,k-1) \\ (k = m) \wedge (even(s))) \to d(i,j-1,k-1) \\ m < k < j \to d(i,j-1,k-1) \end{cases}$$

$$b(i,j,k) = \begin{cases} (k = m) \wedge (odd(s)) \to e(i+1,j,k+1) \\ (k = m) \wedge (even(s)) \to b(i+1,j,k) \\ m < k < j \to b(i+1,j,k) \end{cases}$$

$$e(i,j,k) = \begin{cases} k = j \to c(i,j,j) \\ m \le k < j \to e(i+1,j,k+1) \end{cases}$$

$$c(i,j,k) = \begin{cases} s = 0 \to C_i \\ s > 0 \to \\ \quad \begin{cases} k = m \to \min(F(a(i,j,k),b(i,j,k)), \\ \qquad\qquad F(d(i,j,k),e(i,j,k)) \\ m < k < j \to \\ \quad \min(c(i,j,k-1), F(a(i,j,k),b(i,j,k)), \\ \qquad\qquad F(d(i,j,k),e(i,j,k)) \\ k = j \to c(i,j,k-1) \end{cases} \end{cases} \tag{5.5}$$

## 6. Space-time Mapping to Incorporate Pipelining

From the algorithm (5.5) derived above, we further improve the efficiency of the algorithm by incorporating pipelining automatically. From the stand-point of implementation, a process **v** in a system of recursion equations will be mapped to some physical processor $s$ during execution, and once the process is terminated, another process can be mapped to the same processor. In fact, such re-use of the resource is the essence of ***pipelining.*** We call each execution of a process by a processor an ***invocation*** of the processor. Let $t$ be an index for labeling the invocations so that the processes executed in the same processor can be differentiated, and let these invocations be labeled by strictly increasing non-negative integers. Then for a given implementation of a program, each process **v** has an alias $[s,t] \overset{\text{def}}{=} f(\mathbf{v})$, telling when (which invocation) and where (in which processor) it is executed. The key to an efficient parallel implementation of an algorithm is to find an appropriate one-to-one function $f$ that maps a process **v** to its alias $[s,t]$ such that $t$ will be non-negative and $t_2 > t_1$ if $\mathbf{v}_1 \prec \mathbf{v}_2$, where $[s_1,t_1] \overset{\text{def}}{=} f(\mathbf{v}_1)$ and $[s_2,t_2] \overset{\text{def}}{=} f(\mathbf{v}_2)$.

### 6.1. Data dependency vectors

To find such a mapping, we require that the domains of the recursion variables of a **Crystal** program be vector spaces, and that the appropriate vector addition and scaler vector product be defined. Though each of the recursion variables $i$ and $j$ in Equation (2.1) assumes an integer value, its domain can be extended to the set of rationals. For instance, an $m$ dimensional process structure is now embedded in an $m$-dimensional vector space over the rationals. From now on, we may refer to the $m$-tuple of values of recursion variables identified with a process as a vector. As suggested by [12], a data dependency vector plays an important role in mapping algorithms to parallel processors. Since the vector addition is defined and the data dependency of two vectors takes on an exact meaning, a data dependency vector can be formally defined:

**Definition 6.1.** A ***data-dependency vector*** is the difference $\mathbf{v} - \mathbf{u}$ of vector **v** and vector **u**, where $\mathbf{u} \prec \mathbf{v}$ (**u** immediately precedes **v**).

### 6.2. Basis communication vectors

As described above, each program defines a set of data dependency vectors. On the other hand, each network topology defines a set of ***basis communication vectors.*** For instance, in an $n$-dimensional hypercube, a processor has $n$ connections to its nearest neighboring processors. Each of the $n$ communication vectors (one for each connection), has $n+1$ components. The first $n$ ones indicate the movement in space and the last one indicate movement in time, which is always positive (counting invocations). These $n$ communication vectors, together with the communication vector $[0,0,\ldots,0,1]$, representing the processors's communication of its current state to its next state, form the basis communication vectors. In an $n$-dimensional network, there can be more than one set of basis communication vectors. Taking a two-dimensional hexagonal network as an example, a diagonal connection has a communication vector $[1,1,1]$. The set of vectors $\{[1,0,1],[0,1,1],[1,1,1]\}$ serves as bases as well as the set $\{[1,0,1],[0,1,1],[0,0,1]\}$. For any $n$-dimensional network which has nearest neighbor connections and is regularly connected and indefinitely extensible, all possible sets of basis communication vectors can be obtained by the enumeration of its symmetry groups Lin and Mead [10].

### 6.3. Uniformity of a parallel algorithm

The concept of uniformity is introduced to characterize parallel algorithms so that an expedient procedure can be applied to a *uniform algorithm* to find the space-time mapping from processes to processors. Let $\mathbf{d}_i$, $1 \le i \le k$ denote the data dependency vectors appearing in a program.

**Definition 6.2.** A uniform algorithm is one in which a single set of basis vectors $\mathbf{b}_j$, $1 \le j \le m$, can be chosen so as to satisfy the mapping condition that every data dependency vector $\mathbf{d}_i$ appearing in the algorithm can be expressed as a linear combination of the chosen basis vectors with non-negative coordinates, in other words, there exists $\mathbf{b}_j$ and $\alpha_j \geq 0$, $1 \le j \le m$, such that

$$\mathbf{d}_i = \alpha_1 \mathbf{b}_1 + \alpha_2 \mathbf{b}_2 + \cdots + \alpha_m \mathbf{b}_m, \text{ for } 1 \le i \le k.$$

### 6.4. Motivation for the mapping condition

The mapping condition is motivated by the possibility of using as the space-time mapping a linear transform from the basis data dependency vectors to the basis communication vectors. Each basis communication vector $\mathbf{e}_j$ corresponds to a nearest neighbor communication on a network of processors, and its time component is always 1. When the mapping between the two sets of basis vectors is determined, then the communication vector $\mathbf{c}_i$ to which a data dependency vector $\mathbf{d}_i$ corresponds is also determined, i.e., $\mathbf{c}_i = \sum_{j=1}^{m} \alpha_j \mathbf{e}_j$ if $\mathbf{d}_i = \sum_{j=1}^{m} \alpha_j \mathbf{b}_j$.

If a data dependency vector has a term with a negative coordinate $\alpha_{j1}$ in its linear combination, then $\alpha_{j1}\mathbf{e}_j$ represents a communication that takes negative time steps, a situation that is not feasible in any physical implementation. In the case of a non-uniform program, more than one set of basis data dependency vectors must be chosen so as to satisfy the mapping condition, and the space-time mapping may become non-linear. In this case, the inductive mapping procedure becomes necessary.

Examples of using such a simple procedure to find linear mappings of processes to parallel architectures for matrix products, LU decomposition, array multipliers, etc., can be found in [2, 3]. Most of the systolic algorithms reported in the literature can be obtained this way, and new systolic algorithms are discovered due to the ability to generate systematically all possible sets of basis communication vectors.

### 6.5. A uniform dynamic programming algorithm

Let $\mathbf{d}_i$, $i = 1, 2, \ldots, 5$ denotes the data dependency vectors of Systems 5.5:

$$\mathbf{d}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{d}_2 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}, \mathbf{d}_3 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \qquad (6.1)$$
$$\mathbf{d}_4 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{d}_5 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Let the following vectors be the chosen basis data dependency vectors:

$$\mathbf{z}_1 \stackrel{\text{def}}{=} \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix}, \mathbf{z}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{z}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

and let matrix $Z$ be defined as $Z \stackrel{\text{def}}{=} (\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3)$, then

$$\mathbf{d}_i = Z\mathbf{a}_i, \text{ where}$$
$$\mathbf{a}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{a}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{a}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix},$$
$$\mathbf{a}_4 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{a}_5 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Note that every component of $\mathbf{a}_i$ is non-negative, thus System 5.5 of dynamic programming is a uniform algorithm. Choose the set of basis communication vectors as

$$\mathbf{c}_1 \stackrel{\text{def}}{=} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \mathbf{c}_2 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \mathbf{c}_3 \stackrel{\text{def}}{=} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \text{ and } C \stackrel{\text{def}}{=} (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3). \qquad (6.2)$$

The linear mapping $T$ from the basis dependency vector to the basis communication vector is then

$$T = CZ^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 1 & 1 \end{pmatrix}.$$

**Proposition 6.1.** *The space-time mapping from each process $(i, j, k)$ in $P_3$ of System (5.5) to an invocation of a processor $[x, y, t]$ is a linear mapping $T$, in the matrix notation where each process and invocation is written as a column vector, or written in a functional notation as*

$$[x, y, t] = f(i, j, k) = [-i, j, -2i + j + k] \text{ where}$$
$$-n \le x < -1, 1 < y \le n, 2 \le t < 2(n-1).$$

*The inverse mapping from the image of $f$ to the process structure, denoted by $f^{-1}$, is*

$$(i, j, k) = f^{-1}(x, y, t) = (x, y, 2x - y + t),$$

*specifying which process is being executed at a particular time step $t$ of processor $(x, y)$.*

Similarly, other space-time mappings can be derived from other sets of basis communication vectors which are given below in a matrix notation with each basis communication vector written as a column vector:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}.$$

The set of basis communication vectors in Equation (6.2), or in each of the above sets of vectors, results in a different systolic architecture, as demonstrated below.

### 6.6. Space-time Recursion Equations (STREQ)

A system of *space-time recursion equations* (STREQ), which describes a target systolic architecture or algorithm can be obtained from an original program and a space-time mapping by algebraica manipulation. The following STREQ that describe the resulting design are obtained by substituting the inverse mapping $f^{-1}$ of $i$, $j$, and $k$, the mappings $[x_l, y_l, t_l]$, $l = 1, 2 \ldots, 5$ of dependency vectors $\mathbf{d}_l$, into the original system of recursion equations, and by renaming $\hat{a}(x,y,t) = a(i,j,k)$, $\hat{b}(x,y,t) = b(i,j,k)$, $\hat{c}(x,y,t) = c(i,j,k)$, $\hat{d}(x,y,t) = d(i,j,k)$, and $\hat{e}(x,y,t) = e(i,j,k)$. Note that as a result, the substitution of the predicates $s = 0$ becomes $z = 1$, where $z \overset{\text{def}}{=} y - x$, $k = m$ becomes $t = \lceil\frac{3(y-x)}{2}\rceil = \lceil\frac{3z}{2}\rceil$, $k = j$ becomes $t = 2z$, $m < k < j$ becomes $(\lceil\frac{3z}{2}\rceil \leq t < 2z)$, etc.

$$\hat{a}(x,y,t) = \begin{cases} t = 2z \to \hat{c}(x,y,t) \\ \lceil\frac{3z}{2}\rceil \leq t < 2z \to \hat{a}(x,y-1,t-1) \end{cases}$$

$$\hat{d}(x,y,t) = \begin{cases} (t = \lceil\frac{3z}{2}\rceil) \wedge (even(z))) \to \hat{a}(x,y-1,t-2) \\ (t = \lceil\frac{3z}{2}\rceil) \wedge (odd(z))) \to \hat{d}(x,y-1,t-2) \\ \lceil\frac{3z}{2}\rceil < t < 2z \to \hat{d}(x,y-1,t-2) \end{cases}$$

$$\hat{b}(x,y,t) = \begin{cases} (t = \lceil\frac{3z}{2}\rceil) \wedge (even(z)) \to \hat{e}(x+1,y,t+1) \\ (t = \lceil\frac{3z}{2}\rceil) \wedge (odd(z)) \to \hat{b}(x+1,y,t-2) \\ \lceil\frac{3z}{2}\rceil < t < 2z \to \hat{b}(x+1,y,t-2) \end{cases}$$

$$\hat{e}(x,y,t) = \begin{cases} t = 2z \to \hat{c}(x,y,t) \\ \lceil\frac{3z}{2}\rceil \leq t < 2z \to \hat{e}(x+1,y,t+1) \end{cases}$$

$$\hat{c}(x,y,t) = \begin{cases} z = 1 \to C_x \\ z > 1 \to \\ \quad \begin{cases} t = \lceil\frac{3z}{2}\rceil \to \\ \quad \min(F(\hat{a}(x,y,t),\hat{b}(x,y,t)), F(\hat{d}(x,y,t),\hat{e}(x,y,t)) \\ (\lceil\frac{3z}{2}\rceil \leq t < 2z) \to \\ \quad \min(\hat{c}(x,y,t-1), F(\hat{a}(x,y,t),\hat{b}(x,y,t)), \\ \quad F(\hat{d}(x,y,t),\hat{e}(x,y,t)) \\ t = 2z \to \hat{c}(x,y,t-1) \end{cases} \end{cases} \tag{6.3}$$

## 7. Target Systolic Architecture

### 7.1. Processor and time requirements

The design consists of a triangular array of $(n-1)(n-2)/2$ processors which complete the computation in $2(n-2)$ time steps, as indicated by the range of $x$, $y$ and $t$ in Proposition 6.1. Notice that the space-time mapping has achieved the goal of decreasing the complexity of processor requirement from $O(n^3)$ of the naive interpretation of System (5.5), to $O(n^2)$ in the new algorithm by only increasing the time complexity with a constant factor, in this case, 2. Thus we see pipelining in the resulting algorithm and the successful sharing of a single processor among $O(n)$ processes.

### 7.2. I/O and storage requirements

In the resulting algorithm, if any of the space components of a process on the left-hand side of a Equation, which is the identifier of a processor, differs from that on its right-hand side (the identifier of another processor), then the corresponding datum is an input to the processor, such as $\hat{a}[x, y-1, t-1]$ in Equation (6.3), which is from the processor below it and which takes 1 time step to arrive. If the processor identifiers on both sides of an equation are the same, then the corresponding datum is a stored value such as $\hat{c}[x, y, t-1]$ in Equation (6.3).

The space-time mapping in Proposition 6.1 yields the well-known systolic architecture for dynamic programming [7]. It can be seen in Figure 4 that a process $(i,j,k)$ is mapped to processor $(-i,j)$ at time step $t = -2i + j + k$. Such an invocation may have inputs from invocations both at time $t-1$ and $t-2$, corresponding to the fast registers and the slow registers described in [7]. All signals controlling the loading and unloading between registers of different speed can be systematically derived by techniques of compiler optimization as described below.

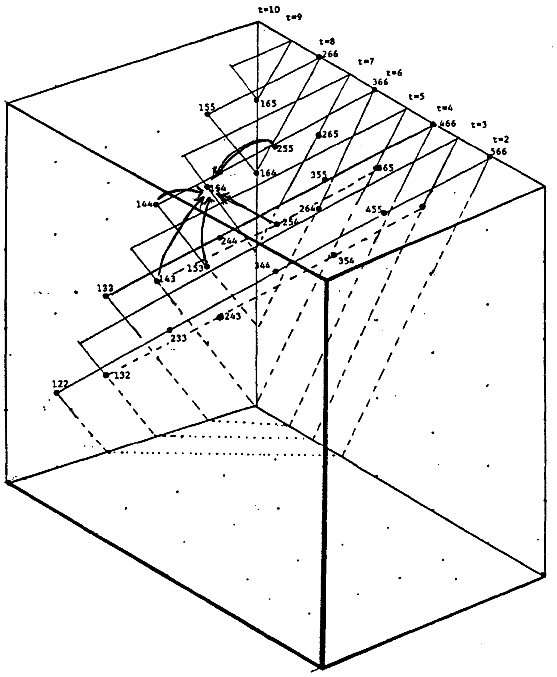


Figure 4: A series of parallel planes indicates the time steps of the space-time mapping. Processes on the same plane are mapped to the same invocation number $t$. (The processor number of a process in this case is its projection onto the bottom plane with the sign of $x$ changed).

### 7.3. Control requirements

The predicates in the STREQ are implemented differently depending on different target implementation media. In a multiprocessor machine, the "processor id" $(x,y)$ can be stored and an "invocation counter" keeps track of $t$ until conditions such as $t = 2z$ is satisfied. In a VLSI implementation, however, it is too costly to store these integers and dedicate hardware to perform the tests. The following optimization of STREQ becomes necessary.

### 7.4. Optimization performed on STREQ

A better design can be obtained by replacing the expensive computations of the predicates by transferring a one-bit control signal, as illustrated by [7]. For a predicate that is independent of $t$, there is no concern, since it can be hardwired into the design. For any predicate that is dependent on $t$, it must be substituted by one that is independent of $t$. Since a communication always both moves in space and takes time to complete, it can be used to "compute" expressions of the space-time recursion variables. For instance, an expression $t = x - 1$ can be computed by a data stream which carries the value "true" and "false".

$$q(x,t) = \begin{cases} (t=0) \wedge (x=1) \rightarrow true \\ \quad \text{(establish the truth initially for the test)} \\ (t>0) \wedge (x=1) \rightarrow false \\ (t>0) \wedge (x>1) \rightarrow q(x-1, t-1) \end{cases}$$

It can be easily seen that such signal can be used to compute any predicates that is linear with respect to the recursion variables. Of all the control signals, the one to implement the test of predicate $t = \lceil \frac{3z}{2} \rceil$ is of particular interest. The predicate contains a piece-wise linear expression, and the resulting signal travels in those processors with even $s$ in half the speed as they do in those processors with odd $s$. This particular type of signal is described incorrectly as a linear signal that moves "two cells every three time units" in [7] as opposed to the above-mentioned piece-wise linear signal.

Central to an optimizing compiler for parallel systems is the making of trade-offs between communications and computations, which can be carried out symbolically in quite an elegant fashion, as illustrated here.

## 8. Related Work

Many attempts in pursuit of systematic methods for designing systolic computations have appeared in the literature. In the classification of systolic designs from the geometric point of view, Lin and Mead [10] classify systolic arrays using symmetry groups, but there is no attempt made to obtain new designs. Cappello and Steiglitz [1] view systolic designs as affine transforms from combinational designs, however, their method does not give clues to finding the appropriate transforms.

Moldovan [12, 13] notes that data dependency vectors are what suggest clues to potential linear transforms. Miranker and Winkler [11] describe a similar method, and they have observed that all 2-dimensional regularly connected arrays can be embedded in a hexagonal array, called a universal array. Quinton [14] starts with a specification called a system of uniform recurrence equations and applies affine and linear transforms to obtain a systolic array. The class of systolic computations that can be described by uniform recurrence equations is limited to those that have only one of their data streams dependent upon other data streams. Li and Wah [8] describe a method that first finds an appropriate placement of one of the input streams and then perform the linear transforms. Delosme and Ipsen [6] describe how to implement a hyperbolic Cholesky solver by a linear transformation from a system of recurrences.

All of the above four methods use (quasi-)linear or affine mappings from a restricted class of algorithms to a new design. The class of designs that can be obtained by these methods are thus subject to restrictions such as uniform recurrent (a more restrictive definition than the uniformity defined in this paper), synchronous, regular, having uniform data flow, etc. The capabilities of these methods are, at best, equal to the mapping procedure for the uniform algorithms described above. However, instead of using a simple, constant time procedure consisting of enumerating the basis communication vectors by subgroup enumerations and finding mappings between the two sets of basis vectors, they find linear mappings by searching through the space of possible linear transforms or input placements. The search requires a time at least polynomially of the size of the problem [8]. None of these methods is able to deal with the problem of the large number of fan-ins and fan-outs.

Li and Wah [9] have classified the problems that can be solved by dynamic programming into several classes and have proposed systolic processors for solving them. The formulation given in this paper corresponds to the most general class — in their terminology, the polyadic-nonserial class. In their treatment, however, the issue of synchronization and timing, which makes the systolic design in [7] particularly interesting, is not addressed.

## 9. Concluding Remarks

The objective of this work is the seeking of parallel programs that effectively utilize underlying technologies. Due to the complexity that might arise in dealing with hundreds of thousands of autonomous parallel processing elements, we find design methodology for this process necessary in order to take the enormous burden off the designers. Program synthesis, at the very basic level, relies on a formal notation for describing the problem, and henceforth on manipulating the descriptions to yield efficient parallel programs. This paper describes two stages of transformations of programs described in the parallel language **Crystal**. The first stage proceeds from a definition of a problem to a program that has fixed fan-in and fan-out degrees and uses only local communications. The second stage incorporates pipelining into this program by space-time mapping. Optimization of control signals is also illustrated.

It is essential that the language **Crystal** be amenable to algebraic manipulation; hence, techniques developed for algorithm transformation can be carried out formally, and potentially by an automatic process. The procedure of space-time mapping for uniform algorithms and the test for the existence of a linear mappings are computational attractive and their complexities are independent of the problem sizes.

### References

[1] Cappello, P.R. and Seiglitz, K., *Unifying VLSI Array Designs with Geometric Transformations,* IEEE Proceedings of the International Conference on Parallel Processing, (1983).

[2] Chen, M. C., The Generation of a Class of Multipliers: a Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers,* October 1985, pp. 116–121.

[3] ————, Synthesizing Systolic Designs, *Proceedings of the Second International Symposiumon VLSI Technology, Systems, and Applications,* May 1985, pp. 209–215.

[4] ————, A Parallel Language and Its Compilation to Multiprocessor Machines, *The Proceedings of the 13th Annual Symposium on POPL,* January 1986, pp. 131–139.

[5] ————, Transformations of Parallel Programs in Crystal, *The Proceedings of the IFIP 86, Dublin, Ireland,* September 1986.

[6] Delosme J-M and Ipsen, Ilse, An illustration of a methodology for the construction of efficient systolic architecture in VLSI, *Proceedings of the Second International Symposiumon VLSI Technology, Systems, and Applications,* May 1985, pp. 268–273.

[7] Guibas, L. J. Kung, H. T. and Thompson, C. D., Direct VLSI Implementation of Combinatorial Algorithms, *Proc. Caltech Conf. VLSI,* 1979.

[8] Li, G.-J. and Wah B. W., *The Design of Optimal Systolic Arrays,* IEEE Transactions on Computer, C-34/1 January (1985), pp. 66–77.

[9] ————, Systolic Processing for Dynamic Programming Problems, *Proceedings of the 1985 International Conference on Parallel Processing,* 1985, pp. 434–441.

[10] Lin, T.Z. and Mead, C.A., *The Application of Group Theory in Classifying Systolic Arrays,* Display File 5006, Caltech, Mar 1982.

[11] Miranker, W. L., Spacetime Representations of Computational Structures, *Computing,* 1984, pp. 93–114.

[12] Moldovan, Dan I., On the Design of Algorithms for VLSI Systolic Arrays, *IEEE Transaction on Computer,* 1983.

[13] Moldovan, Dan. I., *ADVIS: A Software Package for the Design of Systolic Arrays,* Proceedings of ICCD, (1984).

[14] Quinton, P., Automatic synthesis of systolic arrays from uniform recurrent equations, *Proceedings of 11th Annual Symposium on Computer Architecture,* 1984, pp. 208–214.

# APPLICATION OF PARALLEL PROCESSING TO FAULT SIMULATION

Dr. Prabhu Goel, Dr. Chi-Lai Huang, and Robert E. Blauth

Gateway Design Automation Corporation
Littleton, MA 01460

## Abstract

As an alternative to using special-purpose hardware to accelerate the fault simulation process for digital logic design, a new approach is proposed to perform fault simulation using multiple workstations or processors in parallel. By dividing the faults equally among those processors, it is possible to achieve a linear performance improvement for circuits of 5,000 gates and up. The empirical results obtained from implementation of this approach on ELXSI-6400 super mini-computers and Apollo workstations corroborates the expectation.

## 1. Overview

The increasing complexity of digital systems has forced digital designers to use sophisticated computer-aided design (CAD) tools. One of the most widely used ([1]) class of CAD tools is digital logic simulators which are used both for verifying the behavior of conceptual designs of digital systems and for developing tests for digital systems. These simulators are commonly referred to as verification simulators or fault simulators ([2]) depending upon the intended application.

There has been an increasing trend ([3][4][5]) towards applying parallel processing to accelerate logic simulators. Usually, a special hardware engine is proposed and built to implement the simulator algorithm. Hardwiring or microcoding one specific algorithm may increase the performance from ten to one hundred times. The major drawbacks to this approach are the extra hardware cost and the difficulty of interfacing with other tools. Sometimes the performance gained through one step may be lost in the whole design procedure. In this paper, we shall describe one method of applying parallel processing to achieve acceleration of fault simulation. Our method permits us to use a large number of low cost general purpose computers in a parallel processing mode to achieve very high speed fault simulation at very low cost compared to the use of hardware accelerators or expensive mainframe computers. As distinguished from the application of other parallel processing techniques ([3][4][5]) applied to digital simulation, our technique exploits general purpose computers rather than special purpose hardware. Empirical results show that even with a large number of processors, we are able to achieve nearly linear acceleration of fault simulation as a function of the number of processors. With "N" processors, each operating at 1 Mips, we can achieve close to the performance obtainable from a single processor with a processing power of "N" Mips.

## 2. Status of Fault Simulators

Fault simulators are used for developing manufacturing and field service tests with a particular view to detection and diagnosis of defects in digital systems. They are used for evaluating the effectiveness of candidate tests in screening digital systems for defects. The same simulators can also be used for developing diagnostic information that is used in conjunction with the tests to help locate defects caused in manufacturing or defects that develop in the field. The diagnostic information developed from the use of fault simulators can then be used in the repair of defective products as well as in the adjustment of the manufacturing process.

Defects in digital systems can be quite varied ([6]), e.g. shorts, opens, pinholes in oxide, improper alignments, etc.. However, it becomes impractical to simulate this diversity of defects when applying a fault simulator to digital systems of any reasonable complexity. The classical stuck fault model is most commonly used in conjunction with fault simulators since single stuck faults are relatively easy to simulate and empirical evidence suggests that this model is highly effective in practice when applied to digital systems. Since a large number of single stuck faults can exist in a digital system, the computer resources needed to simulate these faults for very large digital systems can be excessive ([14]).

Innovations in fault simulation algorithms have made it practical to apply fault simulation to digital systems consisting of tens of thousands of logic gates. Starting with serial fault simulation ([7]) involving simulation of one single stuck fault at one time, there has been an evolution of techniques that permit simulation of several faults at one time. Parallel fault simulators ([8]) enable a fixed number of faults to be simulated in parallel, typically 31 faults at one time for a computer with 32 bit words. Deductive ([9]) and concurrent ([10]) fault simulators permit simultaneous simulation of a much larger number of faults even more efficiently ([14]) than with parallel fault simulators.

More recently there have been significant developments in application of statistical ([11]) and approximate ([12]) methods to achieve additional speed-up in the fault simulation process. There is clear recognition of the value of approximate and statistical methods of fault simulation in the design and test of digital systems. However, the issue of whether or not one needs accurate fault simulation is very much open ([13]) and subject to both intense controversy as well as to the application at hand and, as such, will not be further elaborated upon in this paper.

Digital systems being designed today are large enough to create compute-time bottlenecks for accurate fault simulation ([14]) even on the most advanced computers. Rigorous design for testability techniques help to significantly alleviate the problem bottlenecks. In spite of this, few large digital systems are being designed with testability enforced rigorously. Hence, there is a continuing need to accelerate fault simulators.

## 3. Acceleration of Fault Simulation

Special purpose computers ([3][4][5]) have been developed expressly for accelerating verification simulation and extended to accelerate fault simulation as well. These special purpose computers, commonly referred to as hardware accelerators, are just beginning to enter the marketplace for fault simulation. Current hardware accelerators exploit special purpose hardware as well as micro-code to directly implement the

0190-3918/86/0000/0785 $01.00 © 1986 IEEE

simulation algorithms.

Another way to improve simulation speed is to use several general purpose processors to execute the same simulation algorithm. Rather than each processor simulating a piece of the algorithm, each processor can apply the complete algorithm on separate pieces of data. For logic simulation, each processor can simulate part of the circuit ([3]). However, it is not practical to obtain a linear performance improvement with an increasing number of processors by this method since there is no way to ensure an even distribution of simulation events across the multiple processors. In fact, the inter-processor communication overhead can quickly overtake the benefits realized from parallelism. A second approach is to let parallel processors simulate the entire circuit with independent subsets of the stimulus vectors. This is feasible only when the set of stimulus vectors is such that independent subsets of stimulus vectors can be identified with each subset not dependent on the circuit state created by preceding stimulus vectors. In general, without explicit designer assistance, it would be infeasible to attempt this form of stimulus vector partitioning for sequential logic circuits. In combinational logic circuits however, each stimulus vector can be viewed as being completely independent of the preceding stimulus vectors. Hence, the technique of parallel processing of vectors is quite viable for combinational logic circuits. In a recent publication ([15]), this technique has been applied by exploiting the bit-wise parallelism inherent within all general purpose processors. In ([15]), 256 stimulus vectors at a time are simulated in parallel with serial fault simulation to achieve significant speed-up in accurate fault simulation for large combinational logic circuits. The same publication also cited an application of the same approach to verification simulation of scan design logic which, for most verification purposes, can be treated in a manner similar to combinational logic circuits.

Fault simulation lends itself to multi-processing even more so than verification simulation. Besides a plurality of gates and stimulus vectors, fault simulation also has another dimension - a plurality of faults. The time used by a fault simulator can be summarized by the following formula:

$$T=TP+TG\bullet P+F\bullet TF\bullet P/2$$

where:

TP is the time spent for the model building and the post-processing,

P is the total number of patterns to be simulated,

TG is the average time to simulate the good machine for one pattern,

F is the total number of faults to be simulated,

TF is the average time to simulate one faulty machine for one pattern.

Here we assume that a fault will be dropped from further simulation if it is detected in one pattern. By assuming all faults are detected uniformly in these P patterns, a fault will be simulated only P/2 patterns on average.

Usually TF is about 0.01 to 0.1 of TG for a concurrent fault simulator. For a circuit of 5,000 gates, there are about 10,000 stuck-at faults to be simulated. Hence, the third factor in the equation is about 50 times the second for this circuit. This ratio will be increased proportionally with the size of the circuit.

In a multi-processor computer, each processor can simulate a subset of the total faults set. At the end of the simulation, one processor can merge the data generated by the other processors and produce the combined simulation results for viewing by the designer. During simulation time, very limited communication is needed between the parallel processors. In this way, each processor will spend about 1/Nth of the third factor on faulty machine simulation. As such, one would expect to achieve an almost linear performance improvement with increased numbers of parallel processors, even up to hundreds of processors for a 50,000-gate fault simulation. Results achieved by us from a practical implementation of these concepts corroborates the expectation.

## 4. AIDSSIM-A: Accelerated Fault Simulation With Parallel Processing

AIDSSIM ([16]) is a concurrent fault simulator which has been used in the industry for several years. Based on AIDSSIM, a new multi-processor fault simulator, AIDSSIM-A has been implemented on ELXSI super mini-computers and on a multi-processor Apollo Domain network. The basic algorithm can be summarized as follows:

Step 1.
The main processor, called parent processor, accesses model and stimulus vectors for the circuit to be simulated.

Step 2.
The parent processor activates several children processors while passing along the circuit information.

Step 3.
Each processor (both parent and child) selects an independent non-overlapping subset of the total fault set; only one fault subset is to be simulated by one processor.

Step 4.
During fault simulation of each subset of faults, the corresponding processor creates the simulation data that is written out to unique files.

Step 5.
As each processor finishes simulating the stimulus vectors and its unique fault subset, it signals the parent processor and returns summary information to the parent while terminating its own processing.

Step 6.
The parent processor merges the simulation data created by each of the children and produces the final result.

Since fault simulators spend the majority of their time in the simulation of Step 4, the algorithm provides an almost linear reduction in elapsed time by using several processors running in parallel. Whether the algorithm can achieve the full linear performance acceleration depends on the effectiveness of the host multi-processing computer in reducing the communication overhead. The ELXSI-6400 ([17]) is a super-minicomputer with closely-coupled processors. In contrast, Apollo computers ([18]) are microcomputers linked with a relatively slower speed inter-processor communication network. In spite of significant architectural differences, both the ELXSI and the Apollo systems show an almost linear performance gain with increasing number of processors while running AIDSSIM-A.

We have experimented with different approaches to partitioning the total fault set into subsets for parallel processing but have not observed any striking difference from one to the other. The results reported in this paper have been achieved by a fault set partitioning that is quite arbitrary and is done as follows. The fault set is assumed to have an arbitrary order. Each successive group of "n" faults is assigned by associating one fault to each of the "n" processors.

## 5. AIDSSIM-A Implementation and Results on an ELXSI-6400

ELXSI-6400 is a multi-processor system with up to ten processors. The system can support up to 192 M-bytes of real memory. Four billion bytes of virtual memory are available per processor. Its GIGABUS, a synchronous, 64-bit data bus operating at 320 M-bytes per second, allows data sharing among processors with a minimum overhead. With cache memory, ELXSI-6400 allows memory segments to be shared among processors without any noticeable overhead. Taking this into consideration, AIDSSIM-A implementation on the ELXSI shares the circuit model's data structure between the multiple processors. Only the single parent processor is used to perform good machine simulation over a small number of consecutive stimulus vectors. The good machine simulation result for each group is saved in common memory for access by all the children processors. When there is not enough memory space to hold good machine simulation data for the next group of stimulus vectors, the parent processor will enter a wait state. The parent processor's wait state is exited only when the children processors have processed the good machine data for a previous group, thereby making memory available for the new group. When a child processor finishes processing with the current good machine data, the child will also enter a wait state until new good machine data is available from the parent. Provision of shared memory to hold good machine data for up to 10 stimulus vectors reduces the number of wait states for each processor. Further, to minimize wait states for the children processors, the parent processor is made to run faster than all children processors by having the parent simulate only half the number of faults simulated by the children processors.

We have processed several circuits on an ELXSI-6400 computer. Figure 1 shows the results for a circuit containing 6600 gates. There are 23741faults; 8404 faults are equivalence faults. After 104 test stimulus vectors, 16031 faults are detected. The fault coverage is 67.52%. Our results are reported from an ELXSI-6400 with 32 M-bytes of main memory and 4 processors. Note that the results go up to 9 processors. We used virtual processors to go beyond the 4 processors on the machine that was available to us. The total processor (cpu) time is the summation of the individual processors' times. Note that the total time increases only about 10% from one processor to nine processors. As can be seen, the average processor time decreases inversely with the number of processors. There is about a 25% difference between the fastest and the slowest processors running independent fault subsets. The effective elapsed time reduction has to be determined from the run time associated with the slowest of the parallel processors.

## 6. AIDSSIM-A On Apollo

Apollo computers are based on the Motorola 68010/68020 microprocessors as well as on proprietary bit-slice processors. They are linked by the high-speed Apollo Domain network which allows communication data rates of up to 10 M-bits per second. Hundreds of Apollo computers can be linked in a network.

Due to the slower communication speed on an Apollo network, (compared to an internal bus on an ELXSI-6400), the AIDSSIM-A implementation on the Apollos does not transfer data between parent and children processors during the simulation step (Step 4). At the beginning of the simulation, the parent processor will send the model and fault data to all the children processors. Each processor will have its local model data structure, but the stimulus vectors file is shared. At the end of simulation, the parent processor accesses the simulation result files of children processors and produces the final result.

The circuit used to produce results shown in Figure 1 was also processed with AIDSSIM-A on an Apollo network with six processors. The results are shown in Figure 2. The Apollo processors used are DN460 or DN660, each ranging from 4 M-bytes to 16 M-bytes of physical main memory. Since only six processors were available, AIDSSIM-A results in Figure 2 are limited to a maximum of six processors. As can be seen, the total processor (cpu) time undergoes relatively small change as we increase the number of processors. For the circuit used, we determined the communication overhead to be about 5 to 10 seconds per processor. The average processor time also decreases inversely with increasing number of processors.

## 7. Conclusions and Future Extensions

AIDSSIM-A has demonstrated that fault simulation can be accelerated almost linearly with an increasing number of processors in a multi-processing environment. A major advantage of this technique is in permitting the exploitation of a large number of low cost processors to provide a powerful cost/performance alternative to the use of special purpose processors or hardware accelerators. We further intend to gather data from running circuits with one hundred thousand gates. Since Apollo networks permit hundreds of processors in the network, it will be quite interesting to see how AIDSSIM-A performs with a very large number of processors. More work is required to permit recovery from the failure of one or more processors being used in the network or in a multi-processing environment. It would also be desirable to dynamically determine the existing/projected workload on various processors in a network and allow for migration of the processing workload from high utilization processors to those processors which have low current utilization by other users. In an environment where some of the processors being operated in parallel are of unequal compute power, it will be necessary to change the allocation of fault subsets and allow for the slower processors to be synchronized with the faster processors without slowing down the latter.

## References

[1] Beresford, R. "The Role of CAE: Views from Users", presented at Design Automation Conference (6/85) published by CMP Publications, Manhasset, NY as part of "CAE: The User, the Industry", 9pp.

[2] Goel, P. and Moorby, P.R., "Fault Simulation Techniques for VLSI Circuits", VLSI Design, (7/84) 22-26pp.

[3] Pfister, G.F., "The Yorktown Simulation Engine, Introduction", 19th Design Automation Conference, (1982) 51-54pp.

[4] Siegel, S. and Kaszynski, M.E., "The Design of a Logic Accelerator", VLSI Systems Design, (10/85) 76-80pp.

[5] Blank, T., "A Survey of Hardware Accelerators Used in Computer-aided Design", Design and Test of Computers, (8/84) 21-89pp.

[6] Timoc, C. et al, "Logic Models of Physical Failures", Proc. of International Test Conference, (10/83) 546-553pp.

[7] Roth, J.P. et al, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits", IEEE Trans. Electron. Computers, Vol EC-16, (10/67), 527-580 pp.

[8] Seshu, S., "On an Improved Diagnosis Program", IEEE Trans. on Electronic Computers, Vol EC-14, (1965) 76-79pp.

[9] Chang, H.Y.P. et al, "Comparison of Parallel and Deductive Fault Simulation Methods", IEEE Trans. on Computers, Vol C-23, No. 11, (11/74) 1132-1139pp.

[10] Ulrich, E.G. and Baker, T., "The Concurrent Simulation of Nearly Identical Networks", Proc. of Design Automation Conference, (1973)145-150pp.

[11] Jain, S.K. and Agrawal, V.D., "STAFAN: An Alternative to Fault Simulation", Proc. of 21st Design Automation Conference, (1984)18-23pp.

[12] Brglez, F., "A Fast Fault Grader: Analysis and Applications", Proc. of International Test Conference, (1985) 785-794pp.

[13] Panel Session #21, International Test Conference, (1985)

[14] Goel, P., "Test Generation Costs Analysis and Projections", Proc. of 17th Design Automation Conf., (1980) 77-84pp.

[15] Waicukauski, J.A., et al, "A Statistical Calculation of Fault Detection Probabilities by Fast Fault Simulation", Proc. of International Test Conference (1985) 779-784pp.

[16] AIDSSIM User Manual, Gateway Design Automation Corp., Littleton, MA.

[17] ELXSI System 6400 Introduction, ELXSI, San Jose, CA.

[18] "Getting Started With Your Domain System", Apollo Computers, Chelmsford, MA.

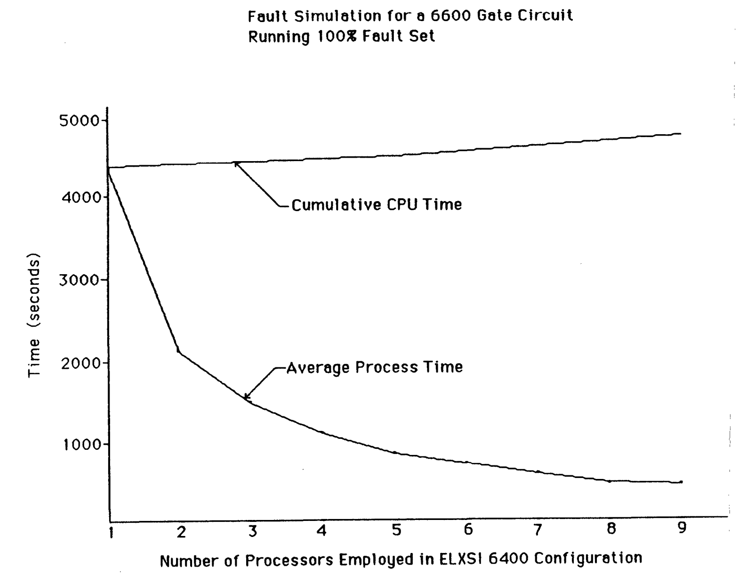


Figure 1. AIDSSIM-A Performance on ELXSI Multiprocessor

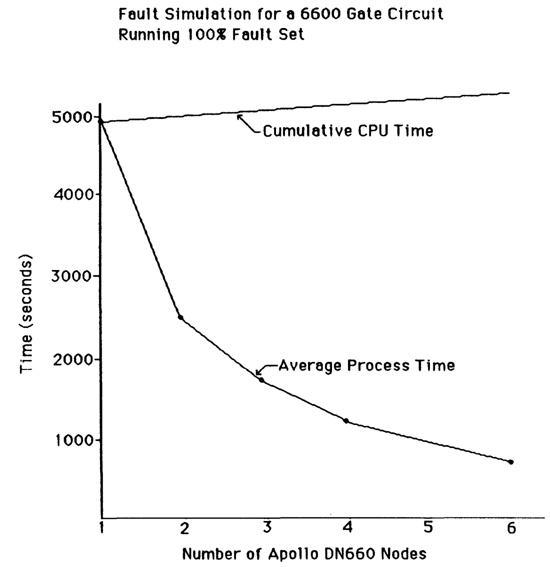


Figure 2. AIDSSIM-A Performance on Apollo Network

# A SLICING ALGORITHM OF CONCURRENCY MODELING
# BASED ON PETRI NETS

*Carl K. Chang and Huiyu Wang*

**Department of Electrical Engineering and Computer Science**
**University of Illinois at Chicago**
**Box 4348, Chicago, IL 60680**

## *ABSTRACT*

**Since its birth two decades ago, the Petri Nets theory has been widely applied to system modeling. In particularily, Petri Nets are highly suitable for modeling and analyzing distributed systems due to its inherent property of concurrency. This paper describes an algorithm which can slice out all concurrency sets of a Petri Nets model. The concurrency set is defined as the set of paths in different processes which should be executed concurrently. Determining the concurrency sets of a concurrent or distributed system model is very significant in performing analysis. The time complexity of the algorithm is $O((n/N)^N)$. Here n is the number of transitions of the model and N is the number of processes of the model. An example of the application of the algorithm in testing distributed software systems is also presented.**

## I. INTRODUCTION

Petri Nets is a well-known theory concerned with the concurrency or parallelism property of different kinds of systems in the area of computer science and other related fields [1]. As the application of Petri Nets are continously emerging, many variations and extensions of Petri Nets have also been developed, such as Modified Petri Nets [2], Discrete Time Stochastic Petri Nets [3], extended Petri Nets [4], etc. Certainly, they provide more powerful tools in modeling and analyzing a variety of systems.

In general, the problem of analyzing a concurrent system is to analyze the interactions among different processes in the system. For the Petri Nets model, the problem is the analysis of the dynamic relations among the transitions in different processes. Typically, firing the transitions in one process can affect the execution of transitions in other processes. For example, in Figure 1, there are two paths in process 1: P11=(t11, t12, t14), P12=(t11, t13, t14) and two paths in process 2: P21=(t21, t22, t23), P22=(t21, t23, t24). P11 and P12, P21 and P22 are mutual-exclusion paths, i.e. execution of one path will suspend another path. In Figure 1, if path P12 in process 1 is expected to be executed, the path P21 in process 2 should be executed concurrently, otherwise the transition t13 will never be enabled. We define the set of paths in different processes which should be executed concurrently as *concurrency set.* Therefore, P12 and P21 are in same concurrency set. Identifying concurrency sets in Petri Nets models is critical to the analysis of these models.

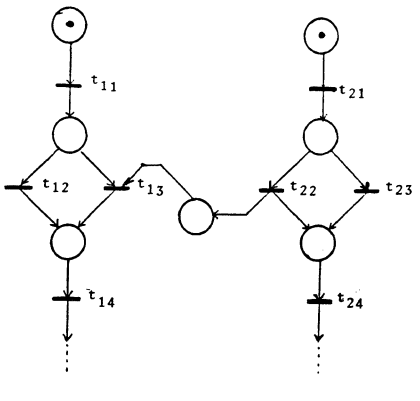


Figure 1. A concurrency set

Upon the modeling of a distributed software system, several processes can be executed concurrently on different processors. In regard to the testing problem, we want to select a set of input data to execute these processes concurrently in order to verify the correctness of the program [5]; for example, to detect deadlock. Because there may exist more than one path in each process, we have to determine which paths should be executed concurrently (i.e. in the same concurrency set). To date, few research results have been reported in this area.

In section II, a slicing algorithm is presented. The algorithm is able to slice out all concurrenccy sets by examining the Petri Nets model. The complexity of the algorithm is discussed in Section III. An example is given in Section IV. Finally, Section V presents the concluding remarks.

## II. SLICING ALGORITHM

Before describing our slicing algorithm, we need the following definitions.

***Definition 1:*** In a Petri Nets model, if transition t is in Process i and I(t), input place of t, is in process j or O(t), output place of t, is in Process k, i ≠ j or i ≠ k, then t is termed a *communication transition.*

***Definition 2:*** A path in a process which contains one or more communication transitions is termed a *communication path.*

***Definition 3:*** For Process i and Process j, if there exists t in Process i, such that O(t) is in Process j, then Process i and Process j are *interrelated,* or Process i has a *relation* with Process j.

***Definition 4:*** A *concurrency set* is a set of communication paths, which are selected from some interrelated processes and can be executed concurrently.

The slicing algorithm can slice out all concurrency sets in a Petri Nets model. In this algorithm, we first find a base path which covers at least one communication transition and put it into the concurrency set S. We mark those transitions in other processes which have relations with this base path. Then we scan each process to select a path which covers all the marked transitions. This path may generate new communication transitions which have relations with it in either the previous (been-scanned) processes or the succeeding (not-scanned yet) processes. If this path does not involve any new communication transitions having relations with the previous processes or these transitions are already in the concurrency set, then we put this path into the concurrency set and mark those transitions having relations with the succeeding processes. Otherwise, if this path involves new communication transitions having relations with a previous process, say x, then we have to give temporary marks to these related transitions and backtrack to process x to find a new path to cover both marked and temporarily marked transitions. If we can find such a path, then we replace the one already in the concurrency set by

0190-3918/86/0000/0789 $01.00 © 1986 IEEE

this new path and mark again the transitions in other processes. Otherwise, we erase temporary marks and try to find a new path other than the old one which was already in the concurrency set. Afterwards, we restart the scanning process from x, until all processes have been scanned and a concurrency set has been found. This procedure is repeated until all communication transitions are included in certain concurrency sets.

Here we should consider a special case in which there is no concurrency set in the model. For example, in Figure 2, transition $t_{j1}$ and $t_{j2}$ are mutually exclusive. There exists only one path which includes $t_{j1}$ in $process_j$. $t_{j1}$ has a relation with $t_k$ in $process_k$ and $t_k$ has a relation with $t_{j2}$ in $process_j$. When we backtrack to $process_j$ we can find neither a path which covers both $t_{j1}$ and $t_{j2}$ nor a new path which covers $t_{j1}$. In this case, the algorithm should terminate. Otherwise it will enter an infinite loop.

Details of the algorithm now follow.

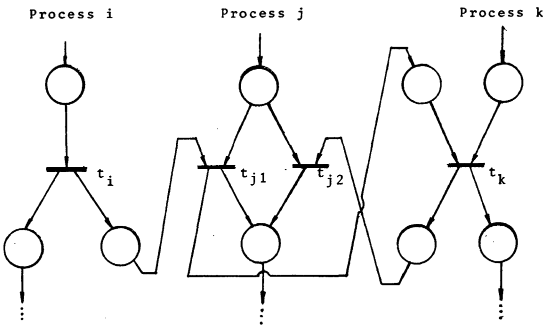


Figure 2. A special case: there is no concurrency set.

***Input:*** P[1], P[2], ..., P[N] : Process 1 to Process N.
CS = { t| t ∈ T, t is a communication transition}

***Output:*** S[1], S[2], ..., S[I] : A set of concurrency sets.

***Variables:*** M = { t| t∈T, t has a mark}
TM = { t| t∈T, t has a temporary mark}
WS = { t| t∈CS and t is in current process being scanned}
N = number of processes in the model.

```
Algorithm_Slicing (P[1],...,P[N],CS : in; S[1],...,S[I] : out)
  for j <-- 1 to N do if There exist more than one path in P[j]
                          then changable[j] <-- true
                          else changable[j] <-- false;
  I <-- 0;  terminate <-- false;
  while CS ≠ ∅ and terminate = false do
      I <-- I+1;      S[I] <-- ∅;
      M <-- ∅;        TM <-- ∅;
      WS <-- ∅;   WS <-- WS U {t}; /* pick up a t from
      CS which belongs to P[1] */

      Procedure_findpath (P[1], WS : in; PA : out);
      /* input process P[1], WS to find a path PA in P[1]
      which covers all communication transitions in WS. If
      there is no such path, PA = ∅ */

      S[I] <-- S[I] U PA;
      M <-- M U {t|t∈P[x], x=1 and t has relation with
      PA};

      Procedure_scanning (P[1], ..., P[N] : in; CS, M,
      TM, S[I] : in&out);
      /* scanning all processes according to the base path
      to find a concurrency set */

      CS <-- CS - CS∩S[I]
  end while
end slicing
```

The procedure *findpath* is to find a path in the given process such that this path can cover all communication transitions in WS. WS is the working set which contains the marked communication transitions in the process to be dealt with. The *Procedure_findpath* is not complicated. Here we will not give a detailed description.

The procedure *scanning* is central to the algorithm. Executing this procedure once, we can get a concurrency set. The following is the *Procedure_scanning:*

```
Procedure_scanning (P[1], ..., P[N] : in; CS, M, TM, S[I]
: in&out)
  i <-- 2;     flag <-- true;
  while i < N  and  terminate = false do
      if flag = true
          then WS <-- P[i]∩M;
              if WS ≠ ∅
                  then Procedure_findpath (P[i], WS
                  : in; PA : out)
                  else i <-- i + 1
          else PA <-- ∅
      end if
      if WS ≠ ∅ and PA = ∅
          then i <-- No. of nearest changable process;
              flag <-- false;
      if WS ≠ ∅ and PA ≠ ∅
          then if (there is no t in previous processes which
          have relations with PA)
                  or (these t are in S[I])
                  then S[I] <-- S[I] U PA;
                      i <-- i + 1;
                      M <-- M U {t|t∈P[x] (x>i), and
                      t has relation with PA};
                      flag <-- true
                  else TM <-- TM U {t|t∈P[x] (x<i) and
                  t has relation with PA};
                      i <-- the least No. of process x
                      (x<i) which has relation with
                      PA;
                      S[I] <-- S[I] - all paths in
                      processes j, j>i;
                      flag <-- false
              end if
      end if
      if flag = false
          then WS <-- P[i]∩(M U TM);

              Procedure_findpath (P[i], WS : in; PA
              : out);

          if PA = ∅
              then TM <-- TM - TM∩P[i];
                  if there is no new path which
                  covers WS
                      then terminate <-- true
                      else flag <-- true
              end if
      end if
  end while
end scanning.
```

## III. TIME COMPLEXITY

*Algorithm_Slicing* is basically a backtracking algorithm. Finding a concurrency set consists of slicing out N paths from N processes. Because there would be several paths in each process, say k, the total combinations of these paths is $k^N$. The backtracking in this algorithm is necessary to find a new path combination. Hence the worst case is that the algorithm tries all possible combinations.

If there are n transitions in the Petri Nets model and N processes, we assume that n transitions are evenly distributed in N processes. That is, each process has n/N transitions. Now the problem is how to determine the number of paths in each process. Let us consider the situation in Figure 3. In the upper half, each place has two output transitions and in the lower half part, each pair of transitions has one output place. Thus the number of paths in this model equals the number of transitions at the $m_{th}$

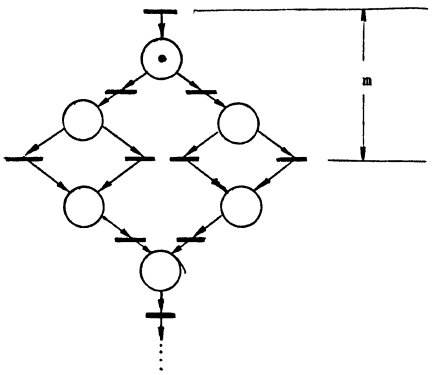


Figure 3. Maximum munber of paths in a Petri Net model

level. Let x denote the number of transitions in this model. We take this model as consisting of two binary trees. Thus we have the equation:

$$2(2^{m} - 1) - 2^{m-1} = x \qquad (1)$$

$2^{m} - 1$ is the number of transitions in a binary tree with depth m. $2^{m-1}$ is the number of transitions at the $m_{th}$ level. Solving equation (1):

$$\begin{aligned}
&2^{m+1} - 2 - 2^{m-1} = x\\
&2^{m-1}(2^{2} - 1) = x + 2\\
&2^{m-1} = (x + 2)/3\\
&m - 1 = lg(x + 2)/3\\
&m = lg(x + 2)/3 + 1
\end{aligned}$$

Thus the number of paths (i.e. number of transitions in the $m_{th}$ level) is:

$$2^{m-1} = 2^{(lg(x+2)/3+1)-1} = 2^{lg(x+2)/3} = (x + 2)/3$$

number of paths with x transitions = (x + 2)/3 (2)

The number of paths in a process calculated by equation (2) is at least more than the general case, because we assume each process has one entry and one exit.

Now we can continue to discuss the time complexity of our algorithm. Since each process has n/N transitions, the number of paths in each process is (n/N + 2)/3, so that the worst case is:

$$[(n/N + 2)/3]^{N}$$

Thus the time complexity
$T(n) = [(n/N + 2)/3]^{N} = O((n/N)^{N})$.

***Discussion:***

There are two factors affecting the time complexity. One is n, the total number of transitions, and the other is N, the number of processes. If N has been given (i.e. N is a constant), then T(n) becomes simpler. The time increases as n becomes larger. But in another case, the problem will be more complicated. That is, if n has been given, the complexity varies as the ratio n/N changes. Therefore, N will affect the complexity considerably. Now let us analyze the complexity function $(n/N)^{N}$

Let $y = (n/N)^{N}$

$$\begin{aligned}
(\,ln\, y)' &= (N\, ln\, n/N)'\\
y'/y &= (N\, ln\, n/N)'\\
y' &= (N\, ln\, n/N)'y\\
&= [\,ln\, n/N + N(-n/N)/(n/N)](n/N)^{N}\\
&= (\,ln\, n/N - 1)(n/N)^{N}
\end{aligned}$$

Let y = 0, that is $(\,ln\, n/N - 1)(n/N)^{N} = 0$

Because in our problem $0<N<n$, so $(n/N)^{N} = 0$

$$\begin{aligned}
&(\,ln\, n/N - 1) = 0\\
&ln\, n/N = 1\\
&n/N = e\\
&N = n/e
\end{aligned}$$

when $N < n/e$ $y' > 0$ i.e. $(n/N)^{N}$ is increasing as N is getting larger,

$N > n/e$ $y' < 0$ i.e. $(n/N)^{N}$ is decreasing as N is getting larger.

Figure 4 shows the curve of the complexity function $(n/N)^{N}$ (when n is given).

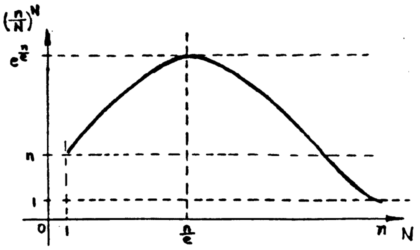


Figure 4. Thc curve of the complexity function

For example, if n = 100, $T(100) = (100/N)^{N}$ The following table gives the time comlexity at different values of N.

| N | 5 | 10 | 20 | 25 | 37 | 40 | 50 |
|---|---|---|---|---|---|---|---|
| $(n/N)^{N}$ | $3.2x10^{6}$ | $1.0x10^{10}$ | $9.45x10^{13}$ | $1.13x10^{15}$ | $9.47x10^{15}$ | $8.2x10^{15}$ | $1.13x10^{15}$ |

Table 1. Example values of the complexity function

When N = 100/e = 36.79, $((n/N)^{N})' = 0$.

Because N is an integer, we consider N = 37 as an extreme value. From Table 1, we can find that when N < 37, $(n/N)^{N}$ is increasing; and when N > 37, $(n/N)^{N}$ is decreasing.

## IV. AN EXAMPLE

In this section, we shall show how to apply our algorithm to the testing of a distributed software system. As mentioned in the INTRODUCTION section, in order to generate test data for the system, we should first determine which parts of the system must be executed during the testing. There may exist a set of concurrency sets in the system model. We have to determine these concurrency sets, before we can generate different sets of input data to execute these concurrency sets.

Figure 5 gives an example of a distributed software system modeled on Petri Nets. There are three processes in the system. Now we apply our slicing algorithm to this Petri Nets model to obtain all concurrency sets.

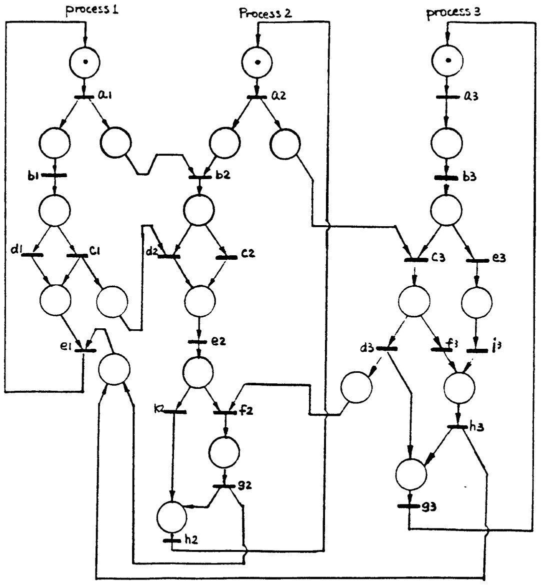


Figure 5. A system model in Petri Nets

*Input* : P[1], P[2], P[3]
CS = {a1, c1, e1, a2, b2, d2, f2, g2, c3, d3, f3}
All three processes are changable.

1. First select the base path (a1, b1, c1, e1) in process 1.
   S = { (a1, b1, c1, e1) }

2. mark communication transitions in process 2.
   M = { b2, d2}; WS = { b2, d2}

3. Find a path in process 2 to cover WS.
   { a2, b2, d2, e2, k2, h2 }
   Because this path only has a relation with process 3, we put it into S1 :
   S1 = { (a1, b1, c1, e1), (a2, b2, d2, e2, k2, h2) }
   mark C3 in process 3.
   M = { b2, d2, c3 }, WS = { c3 }.

4. Find a path in process 3 to cover WS :
   { a3, b3, c3, d3, g3 }
   Because this path has the relation with transition f2 in process 2 and f2 is not in S1, we go back to process 2 and give a temporary mark to f2 :
   TM = { f2 }, WS = { b2, d2, f2 }.

5. Find a new path in process 2 to cover WS :
   { a2, b2, d2, e2, f2, g2, h2 }
   Because this path has the relation with e1 in process 1 but e1 is in S1, so put this path into S1 and delete the old one.

   S1 = { (a1, b1, c1, e1), (a2, b2, d2, e2, f2, g2, h2) }
   mark transition c3 in process 3 :
   M = { b2, d2, c2 }, WS = { c3 }.

6. Find a path in process 3 to cover WS :
   { a3, b3, c3, d3, g3 }
   Because the transition f2 has a relation with this path is in S1 , we put this path into S1 :
   S1 = { (a1, b1, c1, e1), (a2, b2, d2, e2, f2, g2, h2), (a3, b3, c3, d3, g3) }.

7. Now we have found a concurrency set S1. Delete all communication transitions in S1 from CS :
   CS = { f3 }.

Repeat this procedure: select base path in process 3:
( a3, b3, c3, f3, h3, g3 );
We can easily get another concurrency set S2:
S2 = { (a1, b1, d1, e1), (a2, b2, c2, e2, f2, g2, h2), (a3, b3, c3, f3, h3, g3)}

Now CS = $\emptyset$, the algorithm terminates. There are two concurrency sets S1 and S2 in this Petri Nets model.

## V. CONCLUSION

A slicing algorithm of concurrency modeling based on Petri Nets is presented in this paper. Its application has been discussed and an example of an application in distributed software testing is given. The time complexity of the algorithm is also analyzed. This slicing algorithm will be a very useful tool for analyzing distributed systems modeled in Petri Nets.

## REFERENCES

[1] James L. Peterson, **"Petri Net Theory and the Modeling of Systems"**, *Pretice-Hall, Inc.* 1981.

[2] Stephen S. Yau and Mehmet U. Caglayan, **" Distributed Software System Design Representation Using Modified Petri Net "**, *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 6, Nov. 1983, pp. 733 - 745.

[3] Michael K. Molloy, **"Discrete Time Stochastic Petri Nets"**, *IEEE Transactions on Software Engineering,* Vol. SE-11, NO. 4, April 1985, pp. 417 - 423.

[4] M. Tamer Ozso, **"Modeling and Analysis of Distributed Database Concurrensy Control Algorithm Using an Extended Petri Net Formalism"**, *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 10, Oct. 1985, pp. 1225 - 1239.

[5] Lee J. White and Edward I. Cohen, **"A Domain Strategy for Computer Program Testing"**, *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 3, May 1980, pp. 247 - 257.

# PARALLEL ENTITY CENTERED SIMULATION ON THE BUTTERFLY COMPUTER

John B. Gilmer, Jr.
The BDM Corporation
7915 Jones Branch Drive
McLean, Virginia 22102

George Hartwig
Louise Kokinakis
US Army Ballistic Research Laboratory
Aberdeen Proving Ground, Maryland 21005

Abstract--Simulation, which is very often compute bound on uniprocessor computers, is an excellent candidate application for parallel processing. This paper describes the implementation of a simple military simulation on the BBN Butterfly parallel computer. The entity centered nature of the problem will be described, followed by an overview of the model that represents the essential characteristics of similar full scale simulations. Implementation issues on the Butterfly and performance will be given, with particular attention to scaling up to the 124 node size of the available machine.

## 1. Background

The performance problem in military simulation is illustrated by the CORBAN (Corps Battle Analysis) simulation, which runs only about twice as fast as real time for a large scenario [1]. Since the period simulated is measured in days, this is too slow for operational use in decision aids. This is very unlikely to be achieved at reasonable cost without resort to parallelism. The effort reported in this paper investigated the potential for speedup by using a small, simple simulation having similar structure to CORBAN, implemented on the Butterfly computer. A full report on the effort has been prepared [2].

The CORBAN simulation represents ground combat and some aspects of air combat, including perception, attrition, movement, and decisionmaking processes [3][4]. A simulated scenario includes many entities, each representing a military unit, which may have a variety of types of weapons. Each such entity may only engage enemy units which it can detect.In addition, combat simulations include decisionmaking algorithms and movement algorithms that cause entities to change location and hence the combat relationships among them. A time stepped approach is used in CORBAN, so that each $\Delta t$ every unit perceives, shoots, makes decĩsions, and moves. Entities directly observe and modify state variables of each other. This style of programming maps well onto a shared memory parallel computer such as the Butterfly where all state variables can be maintained in global memory. [5]

## 2. The Butterfly Computer

The Butterfly is a shared memory parallel computer having up to 256 nodes, each within an 8 Mhz 68000 microprocessor, a 2901 bit slice node controller, and 1 Mbyte of memory. Instructions referencing memory located locally take about 1 1/2 microseconds, while those referencing memory on another node take about 5 microseconds. References to other nodes are through an omega, or "Butterfly," network topology from which the machine takes its name. Each link has a capacity of 32 mbits/sec. The machine is programmed by the user as an MIMD (Multiple Instruction stream, Multiple Data stream) shared memory machine [6].

## 3. The Simulation

With a parallelism approach that processes entities on different processors, potential for problems occurs where more than one entity may access the same data simultaneously during perception, combat processes, movement. Decisionmaking is a function internal to an entity, so it has no intrinsic requirement to reference other entities. Only the perception, combat, and movement functions were included in this simple simulation since they are the problem areas. This simulation named "Zipscreen," was written in C with the addition of parallel constructs available through the BBN "Uniform System" package which supports allocation of global memory blocks and scattered matricies, and parallel process generation [7].

The data structures of the program are shown in Figure 1. As in CORBAN, units are represented by blocks of data of varying length, which contain a number of common attributes in the "unitsb" structure and a list of assets in the concatenated "asitem" structures. Units are accessed using the "unarry" array. Each asset item specifies the type and quantity of each asset. The variable "hexloc" specifies an octal hex location address. These index into the array "phexar" which contains pointers to the linked list of units for each hex. The hex address "newhex" is the new hex a unit moves to during a time step.

The program structure is illustrated in Figure 2. The "runsim" routine cycles through all of the time steps. The "GenOnIndex" is a mechanism provided by the Butterfly "Uniform System" to initiate a task for each unit. The "ciunit" routine calls "percev," the perception routine, to generate a target list of adjacent enemy units. The "hxadd" function is used to generate the hex addresses to be searched. If

0190-3918/86/0000/0793 $01.00 © 1986 IEEE

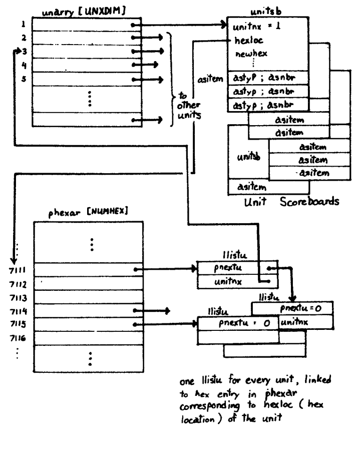


Figure 1. Data Structures

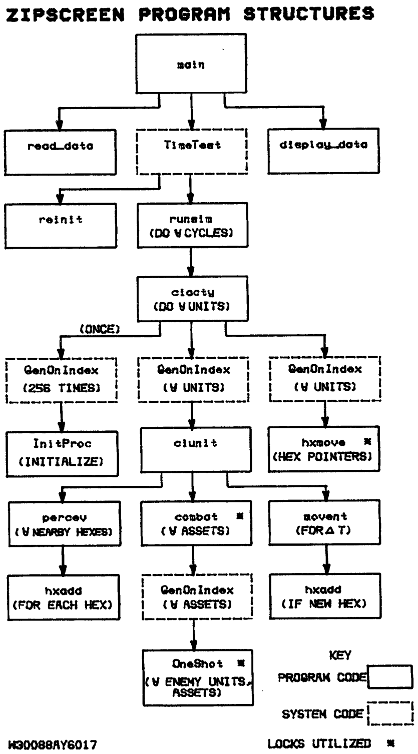
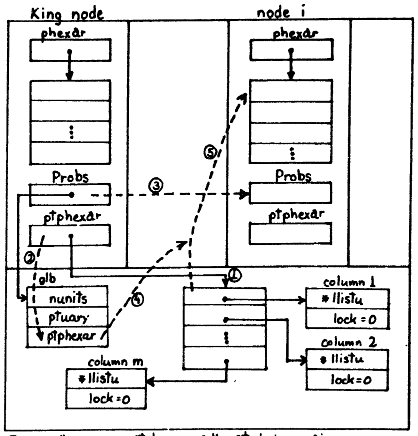


Figure 2. Parallel Program Structures

targets are found, the subroutine "combat" is called. The movement routine then updates the unit's position. After "ciunit" is completed for all units, "hxmove" is called for all units to update the occupancy lists. The combat routine makes separate calculations of effects for each of the weapons in the unit's asset list. A separate task, "OneShot," is spawned for each asset. Each weapon then calculates attrition against each asset of each target unit. The "movent" routine causes each unit to move at a constant speed in the direction "hexdir." When the unit moves into another hex, "hexdir" is added to the hex address "hexloc" to determine the new hex location "newhex." The original data structure for hex access had to be modified to use the "AllocateScatterMatrix" function of the Uniform System. For the hex pointer array, "phexar," the columns correspond to the hex numbers. A second element in the column is used for the lock. The matrix column pointer array is duplicated on all of the processors. Figure 3 illustrates how this is done.



① scatterarray ptphexar allocated in main
② ptphexar inserted into glb by ciacty
③ Probs, The pointer to glb, passed via GenOnIndex
④ ptphexar retreived from glb by InitProc at node i
⑤ column pointers copied to local array by block move in InitProc

Figure 3. Distribution at ScatterMatrix Pointers

## 4. Performance and Conclusions

The original simulation, with no parallelism, required 29.81 seconds for a run with 99 units for five time steps. When parallelism was added for units but the machine was constrained to use only one processor, the run time was 30.59 seconds, a 3.4% increase in run time. Later, with concurrent processing of assets, the single processor run time increased to 31.00 seconds, for a total of 3.5% overhead. A version of the simulation that did not include movement was developed and run on the large 124 node butterfly. Figure 4 shows the results. The results are sensitive to the amount of combat. With an earlier scatter algorithm that permitted only 62 of the 800 units to engage in combat, efficiency for 124 nodes was still relatively high at 56%.

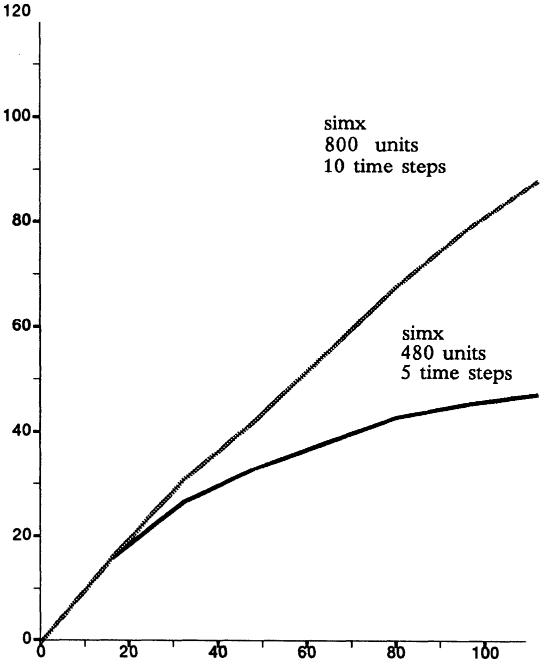


Figure 4. Zipscreen Performance on the Butterfly

In later histogramming, movement processing was found to consume a very small amount of the computational resources. Figure 4 also illustrates speedup curves using the most complete version of the simulation. Note that for the 480 unit case on 124 processors, removal of initialization processing raises the 124 node efficiency to 45%. Typical CORBAN runs range from about 500 to 1000 units. These two cases bound the range of performance that might be achieved without the use of optimized scheduling. Histograms indicated that load balancing was the dominant cause of inefficiency.

The Butterfly computer running the Zipscreen simulation demonstrates the feasibility of parallel combat simulations of the type similar to CORBAN, having entity centered, time stepped architecture. Good scaling with additional processors was demonstrated for up to 124 processors, with efficiency limited by the task breakdown implied by data.

An effort is currently underway to demonstrate parallel simulation on a larger, more detailed scale by implementing a subset of the CORBAN simulation on the BBN Butterfly and a hypercube architecture simulation. These projects are being pursued by the US Army Ballistic Research Laboratory, Los Alamos National Laboratory and The BDM Corporation, under the sponsorship of the Defense Advanced Research Projects Agency.

## 5. Acknowledgements

The simulation demonstration effort described here was largely pursued as part of a one-week Butterfly training course in August 1985. The BBN personnel rendered valuable assistance. This effort was sponsored by the Defense Advanced Research Projects Agency.

## 6. References

1. Tim Hawkins, report on performance of the CORBAN simulation running on a 12 1/2 Mhz 68000 (no wait states) system on a scenario comprising about 800 entities, August 1985.

2. John B. Gilmer, Jr., Report on the BBN Butterfly Multiprocessor Workshop (August 12-16, 1985): A Battle Simulation Application, The BDM Corporation, BDM/R-85-0980-TR, Sept. 13, 1985.

3. Deep Attack Programs Office Corps Battle Analysis Final Report, Volume II, The BDM Corporation, BDM/W-85-0732-TR, 30 June 1985.

4. John B. Gilmer, Jr., "Dynamic Variable Resolution in the Quickscreen Combat Model," Proceedings of the Winter Simulation Conference, IEEE, pp 597-602, Dec 1984.

5. John B. Gilmer, Jr., "Multiprocessor Computer Architecture for Entity Centered Simulations," Proceedings of the 13th Pittsburg Simulation Conference, Instrument Society of America, 1981, pp 513-518.

6. Crowther, Goodhue, Starr, Thomas, Milliken, and Blackadar, "Performance Measurements on a 128-node Butterfly Parallel Processor," Proceedings of the 1985 International Conference on Parallel Processing, Aug 20-23, 1985, pp 531-540.

7. "The Uniform System Approach to Programming the Butterfly Parallel Processor," Version 1, Dec. 19, 1985. BBN Laboratories, Inc.

# A TECHNIQUE FOR SIMULATING SIMD SIGNAL PROCESSING ALGORITHMS ON THE PASM PARALLEL PROCESSING SYSTEM

**Edward C. Bronson**
**James T. Kuehn**[†]
**Leah H. Jamieson**

**School of Electrical Engineering**
**Purdue University**
**West Lafayette, IN 47907 USA**

**Abstract** -- This paper describes a new technique to analyze data-independent SIMD algorithms to obtain performance measures of their execution on a parallel processing system. The execution time of an SIMD algorithm is the greatest amount of time required by the control unit and any one processing element to complete execution. The operations performed by the control unit and this single processing element constitute the longest processing path or critical path through the parallel algorithm. A procedure is described to extract the critical path and to simulate the execution of the critical path on an SIMD control unit and a single processing element. This is a general technique that combines a complete parallel architecture hardware model with serial processing tools to obtain very detailed SIMD algorithm performance information. The simulation studies presented here are for SIMD signal processing algorithms written in Parallel-C and executed on the PASM parallel processing system.

## I. Introduction

An SIMD machine typically consists of a *control unit (CU),* an interconnection network, and *N processing elements (PEs)* where each PE is a processor-memory pair. The CU broadcasts instructions to the PEs and all enabled PEs execute the same instruction at the same time. *Masking* operations can be performed by the CU to enable or disable any set of PEs within the machine. The execution time of an SIMD algorithm is the greatest amount of time required by the CU and any one PE to complete execution of the algorithm. The operations performed by the CU and this single PE constitute the longest processing path or *critical path* through the parallel algorithm. Performance measures of a parallel algorithm such as execution time, throughput, utilization, and speedup can be determined by extracting the critical path and simulating a CU and a PE executing it. This paper describes a new technique to extract, compile, and simulate the critical path of an SIMD algorithm.

Many parallel processing systems are currently being designed or are under development. During the early stages of these systems parallel hardware and a complete set of parallel software tools are often not available. The simulation technique presented here is important because the critical path through the algorithm is a serial program and thus can be processed and simulated using existing serial processing tools. Simulation of the critical path permits analysis of all major components of an SIMD algorithm including enabling and disabling PEs, interprocessor communications, and CU operations. This technique is being used to design a real-time distributed speech understanding system [1].

Section II discusses the PASM architecture which is used as a model in this paper, the SIMD signal processing algorithms to be analyzed, and aspects of the Parallel-C programming language in which the algorithms are coded. Section III details the critical path analysis procedure. As an example of this procedure, Section IV presents the analysis of a fast Fourier transform algorithm.

## II. Architecture and Algorithms

### PASM Parallel Processing System

*PASM* [10] is a partitionable SIMD/MIMD system which can be structured as one or more independent SIMD and/or MIMD machines. This dynamically reconfigurable multiprocessor system is being designed at Purdue University to serve as a research tool in speech and image processing applications. PASM consists of *N* PEs, *Q Microcontrollers (MCs),* a partitionable multistage interconnection network, multiple secondary storage units, and additional processors for job scheduling and I/O.

N/Q PEs are associated with each MC to form an *MC-group.* An SIMD machine *partition* of MN/Q PEs is formed by combining the efforts of M MC-groups. The MCs act as CUs for their PEs in SIMD mode. The PASM operating system will schedule and protect system resources to support a multi-user environment. A prototype of the PASM parallel system with $N = 16$ PEs and $Q = 4$ MCs is being constructed. The design uses the Motorola MC68010 microprocessor [5] as the computation engine of the PE, MC, and support processors.

### Parallel-C Signal Processing Algorithms

Many common signal processing algorithms used in speech and image processing such as fast Fourier transforms *(FFTs),* digital filtering, power spectrum estimation, convolution, and histogramming [7, 8] have very regular data-independent control structures. It is this type of algorithm that this work addresses.

The SIMD signal processing algorithms are written in Parallel-C [4], an extension to the C programming language [3]. Parallel-C provides a "high-level" medium to describe a parallel algorithm without excessive emphasis on the details of the machine's hardware. The language extensions for SIMD mode processing include the definition of parallel variables, functions, and expressions; a scheme for accessing parallel variables by using *selectors;* and extended control structure semantics. Selectors are special variables which are used to define the set of PEs that will perform a parallel operation. As in serial C, scalar variables, functions, and operations may also be defined. Variables are declared in Parallel-C by the new keywords *parallel*, *scalar*, and *selector.*

[†] current address: Institute for Defense Analyses - Supercomputing Research Center, 4380 Forbes Blvd., Lanham, MD 20706 USA

0190-3918/86/0000/0796 $01.00 © 1986 IEEE

Algorithms in Parallel-C can be written for whatever number of processors seems "natural" without regard for the number of physical processors that may actually be available at run time.[‡] If the algorithm's ***extent of parallelism*** exceeds the physical machine size, multiple ***virtual processing elements (VPEs)*** can be emulated on each real ***physical processing element (PPE)*** to create a ***virtual machine size (VMS)*** equivalent to the extent of parallelism. For each VPE emulated within a PASM PPE, a ***virtual microcontroller (VMC)*** is emulated within the associated ***physical microcontroller (PMC)***. If the extent of parallelism in an algorithm is $E$ but only $H$ PPEs are available in hardware, the operating system will map E/H VPEs into each real PPE and E/H VMCs into each PMC. Therefore, each of the M MC-groups in the machine partition emulates E/H virtual MC-groups.

Conceptually, the ME/H MC-group emulations described above are separate processes, M of which are scheduled to run at a time. These processes are forced to perform a ***context switch*** at certain synchronization points in the algorithm to ensure correct emulation of the SIMD machine. For example, consider a synchronized SIMD inter-processor communication step in which all source VPEs write data to destination VPEs through the interconnection network. All VPEs must send data to their destinations before any is allowed to retrieve the data. A synchronization point is required in the network transfer routine to force each virtual MC-group to context switch. When E/H context switches have occurred all VPEs can continue.

## III. Analysis Procedure

The execution time of an algorithm is obtained by extracting the critical path from the Parallel-C source program, compiling the resulting serial C program, assembling the code, adding the PASM library functions, and simulating the algorithm.

### Critical Path Extraction

The source code of an algorithm written in Parallel-C describes each scalar and parallel operation and the appropriate masking information. The Parallel-C Critical Path *(PCCP)* preprocessor accepts the Parallel-C source program and converts it into an equivalent critical path C program using the PASM hardware machine size (Q MCs, N PEs) and the machine partition size (M MCs, MN/Q PEs) to specify the conversion. The critical path conversion process mimics some of the operations that would be performed by a Parallel-C compiler.

PCCP extracts the critical path of an algorithm by transforming each Parallel-C input line into its critical path equivalent. As variables are declared, PCCP maintains a list of each variable name and type while converting the declaration into equivalent C language notation. Scalar variable declarations remain the same except for the elimination of the *scalar* keyword. Parallel variable declarations are converted into normal C language variables. Selector variables are not declared in the critical path code. PCCP uses the variable information to differentiate between scalar and parallel expressions and to analyze parallel variable selector information.

A Parallel-C expression containing only scalar variables is equivalent to its critical path counterpart and no changes are necessary. Conversion of an expression containing parallel variables requires several steps. Initially, selector information is extracted from the expression. The selector information within a parallel statement specifies the MC masking operations required to execute the statement. It is necessary for the MC to perform masking operations only if the new selector differs from the selector of the previously executed parallel statement. The number of selectors, the selector type (select a fixed group of VPEs or a group of VPEs that varies depending on the value of a program variable), and the surrounding algorithm control structure may also indicate the necessity to perform MC masking operations. PCCP tracks selector information and generates special instructions for the MC to perform masking when required. After generating the masking information the parallel expression is converted by removing all selector information.

The PCCP preprocessor performs all of the operations of the standard C preprocessor [3]. This permits the inclusion of special optimized algorithm macros and assembly language code to improve algorithm performance. The preprocessor also permits the selective elimination of sections of Parallel-C source code from the final program. With the exception of the expressions that translate into MC control instructions, removing source code permits a closer dissection of the execution times of the operations which comprise the algorithm such as masking operations time, operating system overhead, and the time for inter-processor communications.

### Simulation

The resulting serial C source program text is prepared for simulation by an MC68010 C compiler, assembler, and linker-loader. PASM library functions to perform C startup, initialization, mathematics, interconnection network, PE masking, and context switching operations are linked to the algorithm code.

The output of the linker-loader is an object file and is interpreted by the simulator. The simulation program is used to simulate the CU and a PE executing the critical path of the parallel algorithm. The simulator is a general purpose uni/multi-processor hardware description and simulation program. It allows the modeling of a wide variety of machine interconnection structures and processing models.

To obtain the algorithm critical path execution time the critical path VPE and associated VMC are simulated. When the machine partition size is less than the extent of parallelism, the critical path execution time must be multiplied by E/H to obtain the complete algorithm execution time. This is accurate because the overhead of the operating system to perform context switching is included in the PASM library functions and hence in the critical path simulation results.

## IV. Example Algorithm Analysis

As an example of the detailed algorithm performance measures obtainable by the critical path extraction technique, the analysis results for a 512-point complex decimation-in-time FFT algorithm are presented. The PE model used in this simulation uses the Motorola MC68010 microprocessor and MC68881 floating-point coprocessor as the execution unit [5, 6]. A multistage cube interconnection network [9] is used.

Parallel SIMD algorithms to perform FFTs have been described in [2]. The principal components of the algorithm include computation steps called ***butterfly operations,*** interconnection network transfers, PE masking

[‡] This approach can be used for an operating system implementation in which multiple jobs may be active simultaneously and the available resources cannot be predicted at compile time. In real-time systems, programs would be coded for a specific number of processors and would be compile-time bound to those processors.

operations, and algorithm control. The structure of the 512-point algorithm consists of 9 stages each containing 256 butterfly operations. Each butterfly operation requires 2 complex floating point additions and 1 complex floating point multiplication. The extent of parallelism in the algorithm is 256 PEs. The complex data, D(i), $0 \leq i < 512$, is distributed 2 data items per VPE. At the start of the algorithm VPE i contains D(i) and D(i + 256). At algorithm completion VPE i contains data item br(2i) and br(2i + 1) where br(i) indicates the bit-reverse of i. The algorithm is optimized by pre-calculating the FFT twiddle factors, masking operation constants, and interconnection network transfer settings.

Figure 1 is a graph of the cumulative execution times of the operations that comprise the algorithm as a function of the number of PPEs within the machine partition. Starting from the bottom of Figure 1, each solid line indicates the algorithm execution time resulting from the addition of the labeled operation to the operations graphed below. The top curve on the graph was obtained by simulating the complete FFT algorithm. The second curve from the top was obtained by using PCCP to eliminate the code that performs the operating system functions. The difference between the top two curves is the operating system execution time. The difference between the next two curves is the execution time of the butterfly operations. Additional performance measures such as overhead ratio, speedup, and throughput [11] can be obtained by combining these execution time results. Processor utilization [11] can be inferred by using PCCP to eliminate portions of the algorithm corresponding to each PE mask setting, obtaining the execution time for the remaining PEs, and comparing these times to the execution time of all PEs.

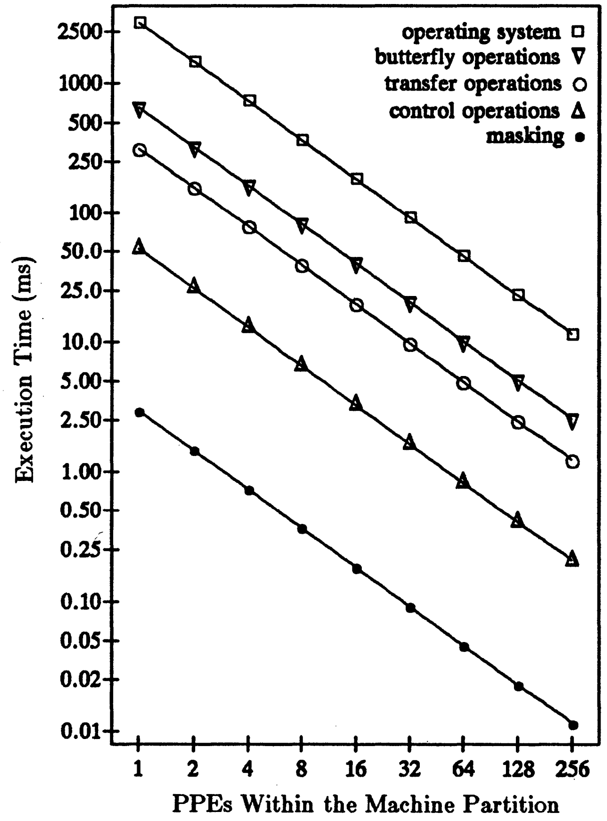


Figure 1 The cumulative execution times for the operations that comprise a 512-point complex fast Fourier transform algorithm.

## V. Summary

A new method to obtain execution time related performance measures of certain data-independent SIMD algorithms running on a parallel processing system has been described. Performance measures such as execution time, speedup, throughput, and utilization can be obtained by extracting the critical path from a Parallel-C algorithm and simulating its performance. The critical path analysis procedure is described and the analysis of an FFT algorithm is presented.

The technique allows detailed analysis of parallel algorithms at an early stage in the development cycle of a parallel system while the software and hardware tools are still under construction. Analysis of the critical path permits a more detailed algorithm analysis than is possible by purely analytical techniques because it includes actual compiler and operating system overhead details such as register allocation, function call overhead, high-level language initialization, context switch overhead, and specific details about the hardware implementation. Although PASM has been used as the specific model in this work, critical path analysis is a general technique and can be applied to other SIMD architectures executing a Parallel-C-like high-level language.

## VI. References

[1] E. C. Bronson and L. J. Siegel, "A parallel architecture for acoustic processing in speech understanding," *1982 Int. Conf. Parallel Processing,* Aug. 1982, pp. 307-312.

[2] L. H. Jamieson, P. T. Mueller, Jr., and H. J. Siegel, "FFT algorithms for SIMD parallel processing systems," *J. Parallel and Distributed Computing,* Vol. 3, No. 1, March 1986, pp. 48-71.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[4] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," *1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 232-235.

[5] Motorola, *MC68000 16/32-Bit Microprocessor Programmer's Reference Manual,* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[6] Motorola, *MC68881 Floating-Point Coprocessor User's Manual,* Motorola, Austin, TX, 1985.

[7] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

[8] A. Rosenfeld and A. C. Kak, *Digital Picture Processing,* Academic Press, New York, NY, 1982.

[9] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing,* D. C. Heath, Lexington, MA, 1985.

[10] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comp.,* Vol. C-30, Dec. 1981, pp. 934-947.

[11] L. J. Siegel, H. J. Siegel, and P. H. Swain, "Performance measures for evaluating algorithms for SIMD machines," *IEEE Trans. Software Eng.,* Vol. SE-8, July 1982, pp. 319-331.

# MULTI GRID ALGORITHMS IMPLEMENTED ON EGPA MULTIPROCESSOR

J. Volkert and W. Henning

Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung (III)
Martensstr. 3 / D-8520 Erlangen / F.R.G.

## Abstract

This paper presents the numerical solution of partial differential equations for simulation of physical phenomena on a class of multiprocessors, called EGPA-systems. The used multigrid methods are well-suited for the considered EGPA systems. An implementation example is given. Communication mechanisms are explained. The results are discussed in respect to efficiency and speed up. At last a view at the future work is taken.

## Introduction

For many years only computer scientists have been interested in multiprocessors consisting of more than just a few elements. But with the appearance of the Hypercube, such a system is commercially available. In the field of vector-computers the trend to multiprocessors is also discernible. Till now, these supercomputers were offered with only a small number of processors, but the number is continuously increasing.

At the University of Erlangen-Nürnberg, West-Germany, multiprocessors have been investigated for several years. We are particularly interested in a certain class called EGPA (Erlangen General Purpose Array). Since we presented the features of such systems in former papers [1-3] we will only give a short description of the EGPA architecture in chapter II. The main part of this paper deals with software aspect.

Two EGPA systems were realized. The first one, a small prototype (5 processors), was built in the late seventies. In chapter III we will have a short look at the other system which is presently running and consists of 21 processors.

With both multiprocessors, many algorithms were implemented. Encouraging results [4-6] demonstrate that EGPA computers are well-suited for a broad spectrum of applications. One area is the numerical simulation of phenomena in physics, chemistry etc. As a rule, the corresponding mathematical models use partial differential equations. In order to find a good solution of such an equation, the discretization must be fine and as a consequence, the required computer performance in FLOPS is very high. The emergence of multigrid methods and the advance of vector-processors dramatically improved the situation. Suddenly, many but not all phenomena can be successfully investigated by simulation. A lot of supercomputer applications have emerged, and a worldwide interest in multigrid algorithms and their implementation is rapidly growing.

Since the main objective of multiprocessors is high speed computation, such a system must be suitably structured to fit these algorithms. In chapter IV we present a general method for mapping multigrid algorithms onto EGPA systems. A simple example, the Poisson-Equation is used in chapter V to demonstrate in more detail the implementation technique and related problems. The measured speedups for this type of equation together with the results for the more complicated Steady Stokes Equations are discussed. Some reasons for the loss in relation to the theoretically maximal values are given.

In the last chapter we give a short survey of our recent research.

## EGPA systems:

EGPA systems form a subclass of multiprocessors with the following features [ 3 ]:

Tight coupling: The architecture consists of processor-memory-modules (PMM) which are connected in two-dimensional orthogonal grid-like structures. Each processor has access to the memories of the four adjacent PMMs (bidirectional connections) and each memory can be accessed by neighbouring processors through a multiport control device.

Restricted neighbourhood: By coupling each PMM with four neighbours the local complexity remains constant throughout the processor plane, whereas the global complexity grows proportionally to the number of PMMs in the system. The principle of constant local complexity and linearly growing global complexity can be realized in various size independent topologies.

Regularity and homogeneity: In most cases a rectangular array of processors is most appropriate, as it can be derived from many algorithmic schemes from relaxation methods. Regularity regarding the interconnection structure, and homogeneity regarding the PMM-structure are mainly directed to the requirements of data-exchange in multiprocessors.

Hierarchy: Two or more arrays of PMMs constitute a hierarchical system through "vertical" connections. Each processor (master) - except those at the lowest level - has access to the memories of four subordinated PMMs (slaves, unidirectional connections).

For larger systems it is unnecessary to have many hierarchical layers. Therefore a possible EGPA computer may have the following structure (s. fig. 1a):

0190-3918/86/0000/0799 $01.00 © 1986 IEEE

- The number of PMMs in the lowest layer is four times larger than the number of PMMs in the second layer. Each PMM in the second layer is connected to a disk via I/O and has access to the memories of the slaves.
- The number of PMMs in the third layer will decrease more strongly to produce for instance a tapering ratio of 1:16. Then for hardware reasons, buscoupling should be preferred memory coupling.

In such a system there is a functional partitioning. The lowest layer is the main worker layer. The next layer has to perform mainly operating system functions, e.g. to manage the data swapping from or to the disks. The processor (or processors) in the top layer constitute an interface to a host (or a workstation). All the upper layers are used for broadcasting (or collecting) data or programs.

### The EGPA system built with DIRMU modules

DIRMU (DIstributed Reconfigurable MUltiprocessor) is a kit for building multiprocessors. The basic element, a PMM consists of two subunits: the P-sub-module (8086/87 with 320 kB private memory and 8 exit-ports) and the M-sub-module (64 KByte 8-port memory). Two PMMs can be connected by plugging a cable into an exit-port of the first module and into a memory port of the other module. Therefore, every combination of memory coupled processors is possible, whereby each processor can have up to seven neighbours (1 exit-port and 1 memory port are used to allow the P-sub-module access its own multiport memory). One possible architecture is the EGPA system shown in fig. 1b.

This system was used for the implementation - among others - of the multigrid algorithms described in the following. The DIRMU kit and the measured results are presented in several papers [5-7] in more detail. For a better understanding the following describes briefly the addressing technique. In this case we want to build the 3-processor-system shown in fig. 2 top. The corresponding DIRMU systems looks like fig. 2 bottom and the addressing is explained in fig. 3.

### General Multigrid Approach on EGPA systems

Let us assume that we are searching for a solution u ($\underline{x}$) of

$$L^u = f, \underline{x} \in B,$$
$$C (u), \underline{x} \in \partial B,$$

where L is a (not necessarily) linear differential operator and f is a function defined on the region B contained in $R^d$. C stands for a condition which u has to fulfil at the boundary of B. Henceforth. We assume that B is (or is almost) the unit cube. Where necessary, this can be achieved by a suitable transformation of the equations [8].

Such equations are usually solved by discretization. For this purpose we choose a grid G with mesh size h. Then we solve the approximate system by relaxation

$$L_h u_h = f_h, \underline{x} \in B \cap G$$

If h is small enough and the relaxation process is executed sufficiently often, then the approximate solution $u_h$ is a good approximation of u. The main problem is that the rate of convergence decrease strongly with decreasing h for all known relaxation methods.

The situation can be essentially improved by using multi grids $G_i$, with $h = 2^{-i}$. A multigrid procedure is given by a predefined sequence of operations of the type R, E, L, $I_1$, $I_2$. The following rules must be obeyed:

- Besides $G_1$, each grid under treatment is relaxed by operation R.
  In $G_1$ the equations are solved exactly (operation type E)
- In case of a transition from $G_i$ to $G_j$, then $i = j \pm 1$.
- In case of a transition from $G_{i-1}$, the residues $(f_h - L_h U_h)$ are restricted (operation L).
- In case of a transition from $G_i$ to $G_{i+1}$:
  a) if $G_{i+1}$ was already used, corrections must be interpolated and added to the existing approximations of the discrete n-values.
  b) if $G_{i+1}$ was not yet used, then the corresponding u-values must be interpolated (operation $I_2$).

There are many possible orders through which the grids can be passed. One example is the V-cycle shown in fig. 4a) and a full multigrid cycle based on V-cycles (fig. 4b).

Multigrid methods are well-suited to EGPA systems because all the operations are homogeneous and local. Thus, a general approach for mapping such an algorithm onto an EGPA system can be given.

We assume an EGPA system with 1<m layers $L_1$ (top), $L_2$, $L_3$, ..., $L_e$ (bottom). Mapping the algorithm onto the computer will be effected in the following way:

- Assign each grid to the finest of all layers which possesses no more modules than the grid has points. This rule avoids situations where, for a calculation, a processor needs data which it can not access directly. Another consequence is that as many grids as possible can be computed by the processors in the lowest layer - so that the computing power is used very efficiently (see fig. 5).
- If the problem is 2-dimensional, each grid plane is partitioned and the parts are assigned to the modules of the corresponding processor layer in a natural way (fig. 6a). Partitioning is achieved in a load balancing manner. This means that all parts should be as equal as possible. Only parts with points of the boundary should be smaller to compensate higher computation load. Three dimensional grids are projected into a 2-dimensional grid which is divided and assigned to processors in the same manner as described before. Each processor manipulates the 3-dimensional column of grid points projected onto the assigned part (see fig. 6 b). In any case, each processor has direct access to all needed data (fig. 6a).

It should be mentioned that the influence of the boundary must be estimated by the programmer. Automatic partitioning methods are not known at present. The computational efficiency can be improved only by analysing execution times obtained with test data.

At runtime, each processor executes a program which differs from the mono-program only in index boundaries and in additional synchronisation parts. If all is correctly implemented the behaviour of the multiprocessor program is completely equivalent to that of the monoprocessor program. How such a correct program can look will be shown in the next chapter.

## An implementation example

In the following, we only discuss the parts of the program which differ from that known from monoprocessors. For demonstration purposes we chose the relatively simple Poisson equations with Dirichlet conditions.

$$\frac{\partial^2}{\partial^2} u(x,y) - \frac{\partial^2}{\partial x^2} u(w,y) = f(x,y),(x,y)\varepsilon(0,1)^2=G,$$

$$u(x,y) = g(x,y),(x,y)\varepsilon\partial G.$$

Let for $h = 2^{-k}$

$$G_h = (x_i,y_j),\ x_i = ih;Y_j = jh;i,j = 0,1...,2^k,$$

A possible discretization of the equation is

$$(4\ u_{i,j} - u_{i,j-1} - u_{i,j+1} - u_{i-1,j} - u_{i+1,j})/h^2=f_{i,j}$$

where

$u_{ij} = u(x_i,y_j)$ and $f_{i,j} = f\ (x_i,\ y_j)$.

We parallelize a full multigrid algorithm (s. fig. 4b) implemented by Witsch [9] with the following operations:

Relaxation: Each grid point is relaxed in red black order

with $u_{i,j} = (h^2 f_{i,j}+u_{i,j-1}+u_{i,j+1}+u_{i-1},+u_{i+1,j})/4$

Red black order means: All grid points are coloured red or black so that the whole grid looks like a checkerboard. All red points are relaxed before all black points. The main advantage is that it is unimportant in which order the reds or the blacks are relaxed. The reason is that the new red values depend only on black values and vice versa.

Restriction: full injection
Interpolation 1: bilinear
Interpolation 2: of 4th order.

We now explain the control flow of the user program for solving Poisson equation. For sake of simplicity we assume that exactly one grid level is manipulated by a processor layer. The following control variables are used

$$\text{deep} = \begin{cases} 0 & \text{grid level is reached for the first time, e.g. position ✱ in fig. 4} \\ 1 & \text{grid level is reached for the second time, e.g. position ▲ in fig. 4} \\ 2 & \text{otherwise} \end{cases}$$

$$\text{deepest} = \begin{cases} \text{true} & \text{finest grid} \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\text{end} = \begin{cases} \text{true} & \text{stop} \\ \text{false} & \text{go on} \end{cases}$$

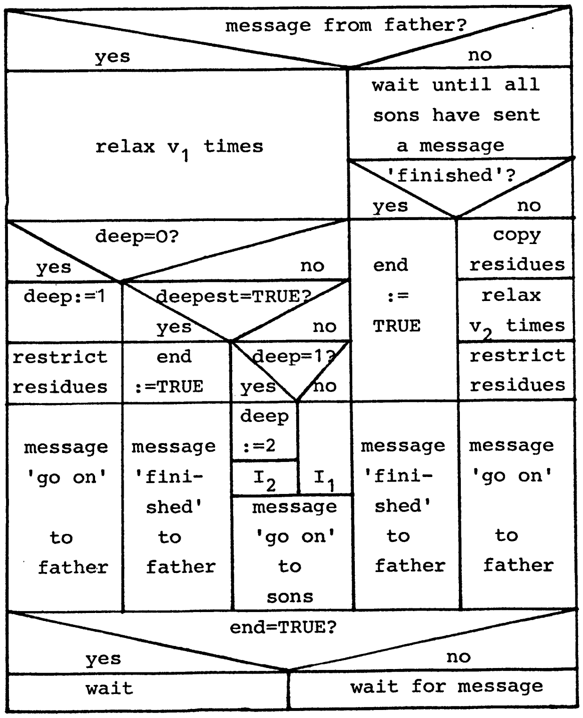
top processor:

The corresponding process is initialized by the system. This process initializes the son processes at each of its slave processors. Then the following loop is executed:

```
exact solution in G1;
interpolate into son memories;
messages to sons,
wait for messages from all sons,
if message = "finished" then stop;
end loop.
```

All other processors execute the same program:

```
Initialisation: deep: = 0; deepest: = true/false
                V1:  = number of relaxation steps
                       befor interpolation
                V2:    number of relaxation steps
                       after restriction,
                initialize sons.
```



Although the above diagram is self-explaining, we have to clarify some features.

The interpolations $I_1$ and $I_2$ produce data, which have only effects on the contents of the memories of the next deeper PMM layer. In the first case the corrections are added to the existing values of the grid points, in the other case the grid is generated for the first time. In contrast to that, the restriction operation writes the residues into the memories of the same layer, since the processors of a layer have no direct access to the memories in the next higher layer. Therefore, the PMMs of the higher layer first have to copy the data into their memories.

While the above diagram shows synchronisation between layers, the following program demonstrates synchronisation between processors of the same level.

We only look at the procedure "relax" which is always called $V_1$ ($V_2$) times in sequence.

```
BEGIN
   WHILE  blackrel & north ≠ blackrel OR
          blackrel & west  ≠ blackrel OR
          blackrel & south ≠ blackrel OR
          blackrel & east  ≠ blackrel

   DO wait (t) OD;
   relaxed;
   redrel: = redrel + 1;
   WHILE redrel & north ≠ redrel OR
         redrel & west  ≠ redrel OR
         redrel & south ≠ redrel OR
         redrel & east  ≠ redrel
   DO wait (t) OD;
   relaxblack;
   blackrel: = blackrel + 1
   END
```

To understand the routine we have to remember that this program is running at any processor. This is the reason why we write relative programs (relative to an assumed PMM location within the system). We achieve this by using variables with relative names. Two examples are "blackrel" and "redrel". "redrel" ("blackrel") is the name of a set of variables. In each PMM, there is an incarnation of "redrel" located in the multiport memory. Such a set is relatively addressed. Each processor writes or reads the incarnation into its multiport memory by using the name "redrel". The corresponding incarnation of the western neighbour (according to the gridlike interconnection structure) is accessed by "redrel & west". With DIRMU (fig. 3) we explain these facts from a hardware point of view. Let us assume that the connection from a processor to the multiport memory of its western neighbour is plugged into exitport 3 and its own multiport memory is accessible by addresses of port 0. Furthermore we assume that "redrel" has the address 8·64 K+d. Then the address of "redrel & west" is 11·64 K+d. This address is transformed to 8·64 K+d by the multiport of the western PMM and this is the address of the variable "redrel" from the point of view of the western processor.

In the above routine the variables "redrel" respectively "blackrel" indicate how often the redpoints resp. the blackpoints were relaxed. Therefore this variables are set to zero during the initialisation. The first WHILE guarantees that the data needed for relaxing the red points already exists in the adjacent PMMs. The second WHILE is necessary to be sure that for the relaxation of the black points all needed red points being in the PMMs of the neighbours are updated.

Now we discuss an improvement of the main routine given by the diagram. The synchronization between layers is achieved by messages. In the implemented program we realy used spinlocks, because this technique is much quickier than message passing (μsec in comparison to msec). This technique we illustrate for the synchronisation between layers (grids).

For controlling the states of the sons and of the neighbours, we introduce the variables.

```
 an = activity number
lan:= last activity number
```

A father-process signals the next activity to the sons by incrementing their activity number

```
an & sons + : = 1
```

(we use this as abbreviation for the real statements

```
an & son 1: = an & son 1 + 1;
       .
       .
       .
an & son 4: = an & son 4 + 1;).
```

(Here we address the incarnations of "an" laying in the memories of the slaves by adding & son i equivalent to the technique how we select the incarnation in the memories of the neighbours. It is also a kind of relatively programming.)

Now let us assume that the sons have copied the last value of "an" into "lan". If a son-process is ready for a new activity, he executes a waiting-loop

```
WHILE an = lan DO wait (t) OD;.
```

After incrementing "an" by the father the processor executes the next statement after this WHILE. For instance, he can control the states of the neighbours by another waiting-loop:

```
WHILE an & neighbours ≠ an DO wait (t) OD
      (an & neighbours stands for
       an & NORTH ≠ an OR ... OR an&WEST≠an)
```

There are a few possibilities for upwards-synchronization (from son to father).

If the process terminates the current activity, he can e.g. change the sign of "an" (an:=-an). Then the father asks for the values of "an" in the sons memories.

```
WHILE an & sons > o DO wait (t) OD;
```

The father process leaves this waiting-loop only in the case, that all son-processes have been carried out the requested activity.

In such a way we synchronize within most of our programs. We found this time saving technique very good and all users preferred it. There is one disadvantage in comparison to message passing: there is no system control of the synchronization. To overcome this we used hardware monitors for tracing. Finally we want to emphasize one fact in context with our Poisson program. No matter which technique we use, only local synchronization takes place. This is important, because a global synchronization would be much more time-consuming.

The possibility to synchronize locally (practiced in most of our programs) and the possibility to program relatively are two very important features of our system.

## Some results

We implemented two multigrid algorithms on the EGPA system build with 21 PMMs (see fig. 1b) [12,13]. In both cases, the processor at the top operates only on grid 0 and the four processors of the middle layer manipulate grid 1. All other grids were assigned to the 16 processors of the bottom layer. The results are given in table 1.

### Some comments to the results:

With increasing number of grids the speedup increases, since the synchronization and communication part of the program decreases in comparison to the calculation part. If the problem is too small, it will make no sense to use such an multiprocessor. Some lower bounds for the size, which a problem should have, are given by Mierendorff and Kolp [11] They showed that in case of a 3 dimensional multigrid algorithm the problem size has to be O ($p^3$); when $p^2$ is the number of processors in the lowest layer. The results for the Steady State Stokes Equation are better than those for Poisson, since in the former case there is more work to do.

Since not all processors are always busy - only one layer is working at a time - the maximal speedup cannot be equal to the number of processors. The ideal speedup gives the upper bound which can maximally be achieved under ideal circumstances. The difference between measured speedup and ideal speedup reflects the loss.

This loss is not effected by the time for the proper synchronization action but by the waiting times. These periods of idleness are caused by unbalanced load, which is obtaind by the fact that it is impossible to divide the used grids into parts with equal load.

## Future work

We introduced EGPA systems. We demonstrated the programming of such systems.

For two multigrid algorithms the speedups measured at our 21-processor system were given. Together with former enjoying results in context with many other problems we were encouraged to continue the investigation of such systems.

In order to increase the performance of the DIRMU-PMM we decided to improve the hardware by adding a powerful coprocessor. First measurements with a first prototype yielded speedups of more than 100 in comparison to the existing system. In the next time we will test whether this good results can be generalized to a whole system [7].

In this context we will investigate the I/O-problem which will occur if the data to be handled are too many and swapping algorithms have to be used. Provided that our theoretical estimations are correct and we believe this, then we will be able to manage this problem with additional hardware without slowing our system down.

## References

[1] W. Händler, U. Herzog, F. Hofmann, H.J.Schneider, Multiprozessoren für breite Anwendungsgebiete: Erlangen General Purpose Array. GI/NTG-Fachtagung "Architektur und Betrieb von Rechensystemen", Informatik-Fachberichte Springer Verlag Berlin Heidelberg New York, 78, 195-208 (1984)

[2] A. Bode, G.Fritsch, W. Henning, J.Volkert, High performance multiprocessor systems for numerical simulation. Proc. SCS 85/First International Conference on Supercomputing Systems, St. Petersburg, Florida, 1985

[3] A.Bode, G. Fritsch, W. Händler, W. Henning, F. Hofmann, J. Volkert, Multigrid oriented computer architecture, 1985 Int. Conf. on Parallel Processing, 89-95

[4] G. Fritsch, W. Kleinöder, C.U. Linster, J. Volkert, EMSY 85 - The Erlangen Multiprocessor System for a Broad Spectrum of Applications, Proc. 1983, Int. Conf. on Parallel Processing, 325-330 and in: Supercomputers: Design and Applications (K. Hwang, ed.), IEEE Comp. Soc. (1984)

[5] W. Henning, J. Volkert, Programming EGPA systems. Proc. IEEE Computer Society Press 1985. 5th Int. Conf. Distributed Computing Systems, 552-529

[6] W. Händler, E. Maehle, K. Wirl, The DIRMU Testbed for High Performance Multiprocessor Configurations. Proc. SCS '85, First International Conference on Supercomputing Systems, St. Petersburg, Florida 1985

[7] A. Bode, Ein Mehrgitter-Gleitkomma-Zusatz für den Knotenprozessor eines Multiprozessors. In: U. Trottenberg, Wypior (eds).: Rechnerarchitekturen für die numerische Simulation auf der Basis superschneller Lösungsverfahren I, GMD Studien Nr. 88, 153-60 )1984)

[8] U. Trottenberg, Private Communication

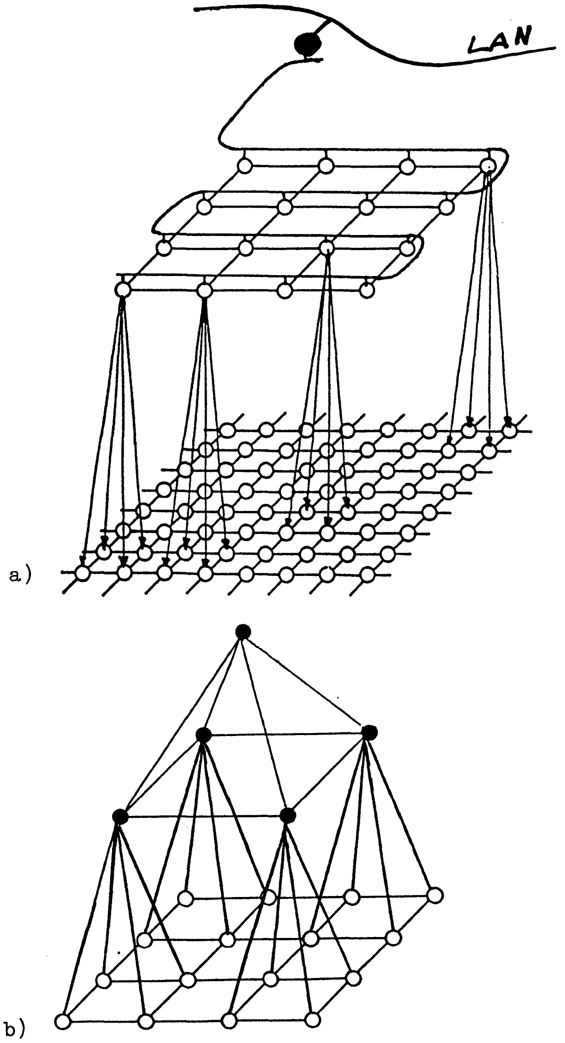
[9] H. Förster, K. Witsch, Multigrid software for the solution of elliptic problems on rectangular domains: MGØØ. In: Multigrid Methods. Proceedings of the Conf. held at Köln-Porz, Nov. 23-27, 1981 (W. Hackbusch, U. Trottenberg, eds.). Lecture Notes in Mathematics. Springer Verlag Berlin, 960, 1-176 (1982)

[10] A. Brandt, Multigrid Techniques, 1984 Guide with Applications in Fluid Dynamics. GMD-Studien 85, St. Augustin, 1984

[11] O. Kolp, H. Mierendorff, Comparison of some Parallel Computer Architectures Applied to Multigrid Algorithms. Proc. Parallel Computing 85, Berlin, 1985.

[12] L. Geus, Parallelismus eines Mehrgitterverfahrens für die Navier-Stokes-Gleichungen auf EGPA-Systemen. Berichte des IMMD 16, 13, Erlangen, 1985

[13] L. Geus, W. Henning, W. Seidl, J. Volkert, MGØØ Implementierungen auf EGPA-Multiprozessoren. GMD-Studien 102, St. Augustin, 121-137 (1985)



Fig, 1; EGPA systems

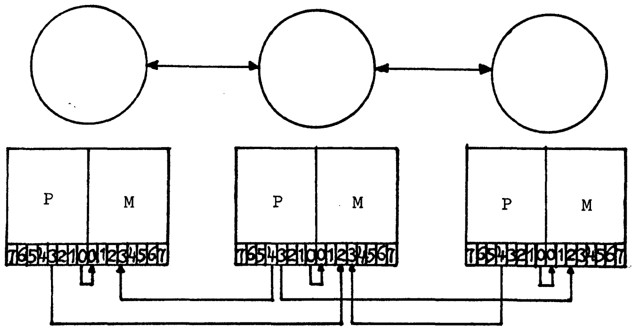


Fig. 2: top : connection structure of a multiprocessor

bottom: realized with DIRMU PMM's

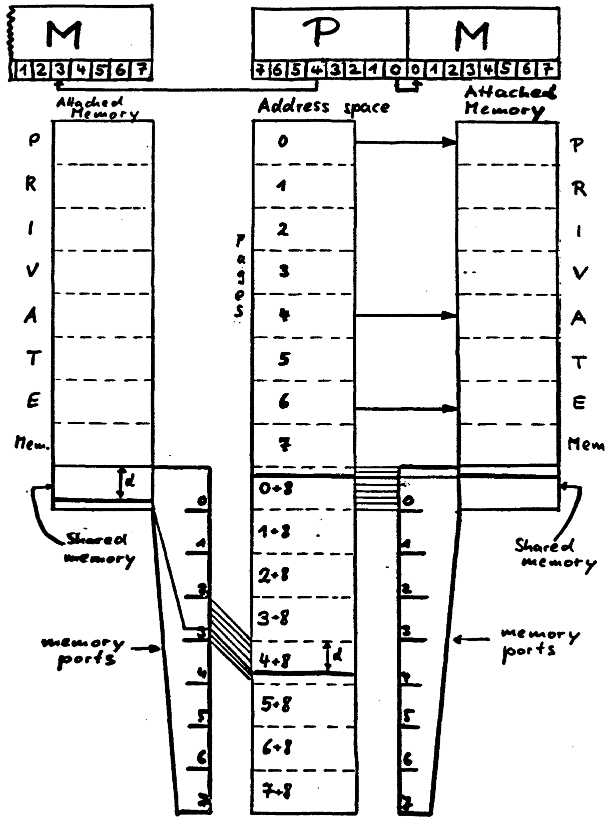


Fig. 3: Addressing scheme for private and shared memory in the DIRMU system.

The private memory is addressed without address transformation (pages 0-7). Each PMM accesses a shared memory plugged into exit-port i by addresses corresponding to page 8+i. The PMM of which the shared memory is a part sees the addresses as belonging to page 8. Therefore each address (8+i)·64K+d (displacement) is transformed to 8·64 K+d.

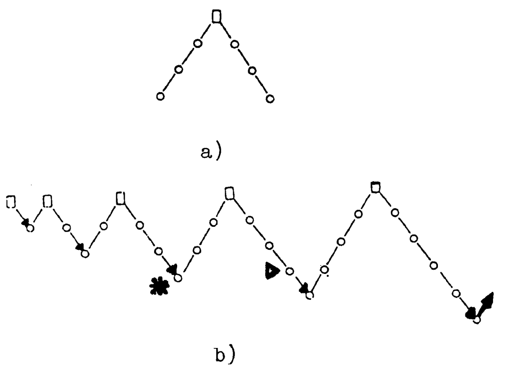


Fig. 4: Multigrid method
a) V-cycle

b) full multigrid with V-cycles

□ exact solving (E)
○ relaxation (R)
\ interpolation $I_1$
/ restriction (L)
↘ interpolation $I_2$

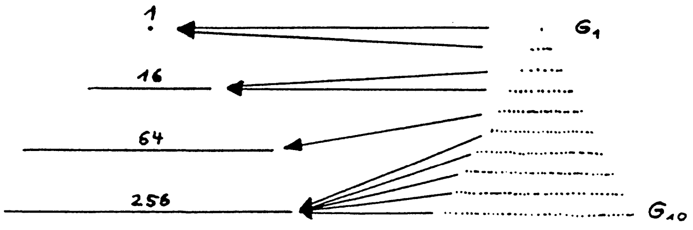


Fig. 5: Mapping the problem structure onto the EGPA system.
Left, a system with 4 layers. The total number of processors in each layer is given. On the right, are the grids $G_i$ ($2^{3(i+1)}$ points in the 3-dimensional case). The arrows show which processor manipulates which grid.

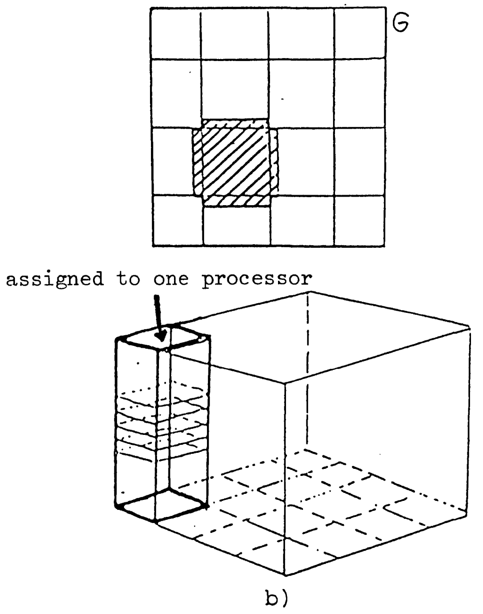


Fig. 6: Mapping one grid onto one layer
a) 2-dimensional case. For one example the area to which one processor needs access is hatched.
b) 3-dimensional case

A) Poisson Equation with Dirichlet Conditions (MGOOD, Witsch)

| Number of grid layers | 6 | 7 | 8 |
|---|---|---|---|
| Grid points in the finest grid | 4225 | 16641 | 65025 |
| Mono runtime /sec | 27.0 | 103.5 | 400.5 |
| 16 processors | 2.27 | 7.18 | 26.1 |
| Speedup | 11.9 | 14.4 | 15.3 |

B) Steady State Stokes Equation

$V_1 = V_2 = 1$ (number of relax steps before/after coarsening)

| Number of grid layers | 5 | 6 | 7 |
|---|---|---|---|
| Number of cells in the finest grids | 1024 | 4096 | 16384 |
| Mono runtime /sec | 12.35 | 48.43 | 192.82 |
| 16 processors | 1.24 | 3.58 | 12.7 |
| Speedup | 9.95 | 13.5 | 15.2 |
| $V_1 = V_2 = 2$ | | | |
| Mono runtime /sec | 20.03 | 78.72 | 313.22 |
| 16 processors | 1.97 | 5.75 | 20.5 |
| Speedup | 10.3 | 13.7 | 15.3 |

Table 1:

Measured results with 16 processors in the lowest level of the EGPA system (21 processor pyramid).

In case B we parallelized a proposal of A. Brandt [10]. This algorithm uses staggered grids. Therefore the region is partitioned into cells.

# CONSTRUCTING THE VORONOI DIAGRAM ON A MESH-CONNECTED COMPUTER

Mi Lu

Department of Electrical and Computer Engineering
Rice University

## Abstract

In this paper, we present a Mesh-Connected Computer algorithm to construct the Voronoi diagram of a set of planar points. Given a set of $n$ planar points, our algorithm constructs a Voronoi diagram on an $O(\sqrt{n}\times\sqrt{n})$ MCC with constant storage per processer in $O(\sqrt{n}\log n)$ time. Using the Voronoi diagram, the problem of determining the nearest neighbor between two sets and constructing the Euclidean minimum spanning trees can be solved with the same time complexity on the MCC. The best sequential algorithms for constructing the Voronoi diagram have an optimal $O(n\log n)$ time complexity. Previous known parallel algorithm for this problem requires $O(\log^3 n)$ time on a Parallel Random Access Machine and $O(\log^4 n)$ time on the Cube-Connected-Cycles with $O(\log n)$ storage per PE.

## I. Introduction

Voronoi diagrams (also called Thiessen diagrams) of a set $S$ of $n$ points is a well known structure which makes explicit some proximity information about $S$. In the Voronoi diagram [1], each point is surrounded by a convex polygon enclosing that territory which is closer to the surrounded point than to any other point in the set. Shamos [2] applied two-dimensional Voronoi diagrams to obtain elegant solutions in computational geometry, such as finding the nearest neighbor, construction of minimum spanning tree, etc. Generalizations of the Voronoi diagram were considered by several authors. Shamos [3] describes an $O(n\log n)$ time sequential algorithm to construct the planar Voronoi diagram for a set of planar points. The strategy used in the serial algorithm is divide-and-conquer. In the merging step the Voronoi edges are generated in a "zigzag" sequential manner which seems difficult to parallelize.

Our algorithm is based on Brown's [4] approach which transforms the problem of constructing a planar Voronoi diagram for an $n$-point set to the construction of a convex hull of $n$ points in three dimensional space. Chow [5] worked on parallelizing the algorithm to solve this problem. Using the shared memory machine (SMM) with $n$ processors, the time performance of her algorithm is $O(\log^3 n\log\log n)$. A recent paper [6] presented an $O(\log^3 n)$ time algorithm to solve this problem with $O(n\log n)$ space. An $O(\log^4 n)$ time algorithm for the construction of Voronoi diagram on Cube-Connected-Cycles with $O(\log n)$ storage per processor, was also proposed in [5] with $O(\log^4 n)$ time complexity. Our algorithm has $O(\sqrt{n}\log n)$ time complexity, and is performed on an $O(\sqrt{n}\times\sqrt{n})$ MCC with constant storage per PE which is a simpler model than the SMM. It needs to be pointed out that $\sqrt{n}\log n < \log^4 n$ for practical $n$.

In the MCC, identical processors are connected via simple and regular communication paths to form a two dimensional array, each with a constant number of registers. It is a single-instruction-stream, multiple-data-stream (SIMD) computer. (For other geometric algorithm on MCCs, see [7-9].)

Several well know MCC algorithm such as Sorting [10], and performing a Random Access Write(RAW) and a Random Access Read(RAR) [11] will be used in our paper. Section II describes the algorithm for constructing the Voronoi diagram, and section III provides some applications of the Voronoi diagram. A summary will be given in section IV.

## II. Constructing Voronoi Diagram

Given two points $p_i$ and $p_j$ in the plane, the set of points closer to $p_i$ than $p_j$ is the half-plane containing $p_i$ that is bounded by the perpendicular bisector of $p_i$ and $p_j$. Denote this half-plane by $H(p_i,p_j)$. Given a set $S$ of $n$ points in the plane, the locus of points in the plane that are closer to $p_i$ than to any other point is a convex polygon (maybe unbounded). It is known as the Voronoi polygon and denoted by $V_i$.

$$V_i = \bigcap_{j\neq i} H(p_i,p_j).$$

The Voronoi diagram is then composed of the $n$ polygons $V(i)$ (see Figure 1). The vertices of the diagram are called Voronoi points and the line segments are Voronoi edges.

To construct a 2-dimensional Voronoi diagram, Brown [4] used a technique called inversion to transform the planar points to 3-dimensions. Given an inversion center $P_0$ and an inversion radius $r$, we can transform point $Q$ to point $Q'$ by inversion, where $\overrightarrow{P_0Q'}$ is in the same direction as $\overrightarrow{P_0Q}$, and $|\overrightarrow{P_0Q'}| = r^2/|\overrightarrow{P_0Q}|$. The inversion has the following properties:

1. An inversion transforms a plane which does not pass through the inversion center to a sphere which passes through the inversion center, and vice versa. Denote the former type of transformation as a *plane→sphere* transform and the latter as a *sphere→plane* transform.
2. The interior of the sphere corresponds to one of the half spaces bounded by the plane and the exterior of the sphere corresponds to the other half space.
3. The inversion is involutory, i.e. application of inversion twice yields the original point.

Given a set of $n$ points on the $xy$ plane, we examine the inversion of three of them with respect to an inversion center $P_0$, not in the $xy$ plane. We can consider the inversion as a *plane→sphere* transform such that the three points are on the $xy$ plane originally and their images are on a sphere passing through inversion center $P_0$. On the other hand, we can consider the inversion as a *sphere→plane* transform in which the three points are originally on a sphere passing through their circumcircle on $xy$ plane and $P_0$ out of $xy$ plane, and their images are on a plane. In this case, if the other $n-3$ points are in the exterior of the circumcircle, then due to property 2, their images are in one half space bounded by the plane decided by the images of those three points (see Figure 2). So, the problem of determining for each set of three points whether their circumcircle excludes all other points becomes that of determining for each face if it is a convex face. Note that the number of the faces $\leq 2n-4$ for a hull with $n$ vertices.

The method used to construct a Voronoi diagram will be presented in a top-down manner below.

This research was partially supported by an IBM Faculty Development Award.

*Algorithm Voronoi*:

*input*: A set of $n$ points represented by their Cartesian coordinates $(x_i,y_i)$. They are distributed on a $\sqrt{n}\times\sqrt{n}$ Mesh-Connected Computer, one point per PE.

*output*: A set of Voronoi edges contained in the mesh.

1. Perform the inversion for each point. For simplicity, choose point (0,0,1) as the inversion center and choose r=1. See Figure 3. Let $(x'_i,y'_i,z'_i)$ be the coordinates of the inversed image of a point, i.e. $x' = x/(x^2+y^2+1)$; $y' = y/(x^2+y^2+1)$; $z' = (x^2+y^2)/(x^2+y^2+1)$.
2. Construct the 3-dimension convex hull $CH$ for the set of image points. This is done by algorithm *Convex Hull* which will be described latter in this section. At the end of this step, each convex face $F_i$ represented by three points through which it passes, is stored in a PE with the flag of this face *extn*=1. The addresses (i.e. PE indices) and the PEs in which its adjacent faces are present are also known by the PE. If we exclude degeneracies, each convex face is a triangle and has three adjacent faces.
3. Each PE performs the "reinversion" for the three points on each convex face it contains. The image $(x'',y'',z'')$ of a point $(x',y',z')$ on the face $F_i$ is given by $x''=x'/(1-z')$; $y''=y'/(1-z')$. Find the center of the circumcircle of the three points corresponding to each face $F_i$. Let $C_i$ denote the center of this circle. If $P_0$ and the $CH$ are at the same side of $F_i$, set $v_i$ to 1. $C_i$ is a Voronoi point.
4. Each PE containing $C_i$ with $v_i$=1 does RAR of $v_j$ from the PE containing $F_j$ such that $F_j$ is an adjacent face of $F_i$ in $CH$.
   If $v_j$=1, $(C_i,C_j)$ is a Voronoi edge.
   Else $C_i$ starts a ray in the $\overrightarrow{C_iC_j}$ direction.

**End** of algorithm *Voronoi*

**Theorem:** Algorithm *Voronoi* finds the Voronoi diagram for a set of $n$ planar points in $O(\sqrt{n}\log n)$ time on an $O(\sqrt{n}\times\sqrt{n})$ Mesh-Connected Computer.

**Proof:** We give this proof with the assumption that the algorithm *Convex Hull* is correct and takes $O(\sqrt{n}\log n)$ time. (This will be justified once we describe algorithm *Convex Hull* later in this section.) Steps 1 and 3 can be done in constant time. Step 4 involves at most three RARs and needs $O(\sqrt{n})$ time. Thus algorithm *Voronoi* has $O(\sqrt{n}\log n)$ time performance, subject to the assumption about algorithm *Convex Hull* to be presented. □

The rest of this section deals with the description of algorithm *Convex Hull*. The strategy we used is divide-and-conquer based on [4]. Given two set of faces each belonging to a convex hull, we need to first decide for each face if it is an external face. We then remove the faces which are not external. Finally we add new faces along the "circuit" which is shared by pairs of faces consisting of an external face and a non-external face.

To judge if a face is external, we need to test if the two convex hulls to be merged, say convex hull $A$ and $B$, are in the same half space bounded by the face. Let the face to be judged be $F_i^A$ in convex hull $A$. Denote all the faces in convex hull $B$ as $F_j^{B\prime}$s. Instead of testing all the $F_j^B$ with $F_i^{A\prime}$s, we only test two representatives $F_r^B$ and $F_{r'}^B$ such that if they are in the same half space as convex hull $A$, every face in $B$ is. When we refer to a face, we indicate it by three points and define each point by its coordinates $(x_i,y_i,z_i)$. The equation of the plane can be expressed as follows:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & x_2 & x_3 & 1 \\ y_1 & y_2 & y_3 & 1 \\ z_1 & z_2 & z_3 & 1 \end{vmatrix} = 0.$$

The general form of the above equation is $\alpha_i x + \beta_i y + \gamma_i z + \delta_i = 0$. The details of algorithm *Convex Hull* are as follows:

*Algorithm Convex Hull*:

*input*: A set of points represented by their coordinates $(x_i,y_i,z_i)$. They are distributed on a $\sqrt{n}\times\sqrt{n}$ Mesh-Connected Computer, one point per PE.

*output*: A set of convex faces where each face is represented by three points; a flag indicating whether the face corresponds to a Voronoi point and the index of the PEs where the information about the adjacent faces are present.

2.1 Sort the $n$ points by $x$-coordinates into non-decreasing order in shuffled row major.

2.2 Recursively solve the convex hull problem in parallel on each block of the mesh where a block consists of $2^k$ PEs in the $k$-th iteration.

2.3 Combine two convex hulls each generated in a submesh. (Fig. 4(a) is a reference.)

2.3.1 Decide for each face $F_i$ if it is an external face, and indicate if so by setting the flag *extn* to 1, as described below.

Each PE which contains a face of convex hull $A$, $F_i^A$, performs the following actions:

Find the two representative faces of convex hull $B$, $F_r^B$ and $F_{r'}^B$; (This is done by invoking the algorithm *Represent* which will be presented later.)

Test if $F_r^B$ and $F_{r'}^B$ and convex hull $A$ are in the same half space bounded by $F_i^A$.

If it is, set flag *extn* to 1.

PEs containing faces of convex hull $B$ performs similar actions independently and simultaneously, with $A$ and $B$ reversed.

2.3.2 Find the circuit edges.

The PE which contains $F_i^A$ and has set $extn_i$=1 does a RAR from the PE where $F_k^A$, an adjacent face of $F_i^A$, lies. If $extn_k = 0$, mark the points shared by $F_i^A$ and $F_k^A$. (They form a circuit edge.) All above is repeated three times, once for an adjacent face of $F_i^A$.

(Do the same thing for the PEs containing $F_j^{B\prime}$s in parallel.)

To order the edges in the circuit, independently and concurrently sort all the marked points $p'$s in $A$ and in $B$ by $cos^{-1}\ y/\sqrt{y^2+z^2}$, which is the angle between $op'$ and the $y$ axis where $p'$ is the projection of $p$ on $yz$ plane and $o$ is the origin.

2.3.3 Find the new convex faces between the circuit in $A$ and the circuit in $B$.

*comment*: Each face is decided by an edge $E_i^A$ in circuit $A$ (respectively $E_j^B$ in circuit $B$) and a vertex in circuit $B$ (respectively $A$). Using the method in [5], we need to find for each $E_i^A$ ($E_j^B$ respectively) a vertex from all the vertices in circuit $B$ ($A$ respectively) to form a convex face. Once a vertex, say $p_j$ is chosen, it divides the vertices in circuit $B$ into two subsets. Those edges with index smaller(respectively grater) than $E_i^A$ need only search among the vertices with index smaller (greater respectively) than $p_j$. Here is the implementation. (See Fig. 4(b).)

Originally all the edges in circuit $A$ forms a set and all the PEs containing points in circuit $B$ (marked points) are the submesh associated with the set. Do the following recursively:

(**) Choose an edge with the middle index in each set and broadcast its record to its associated submesh.

For each associated submesh concurrently, do the comparison in each row of the submesh in parallel and in the right most column of the submesh afterward.

Thus the vertex to form the convex face can be found by the criterion in [5]. Denote the index of the PE containing it as $\text{PE}_m$.

Let the edges with index smaller (greater respectively) than or equal to the middle be a set, the PEs with index smaller (greater respectively) than or equal to $\text{PE}_m$ be the associated mesh of the set. (For simplicity, assume the submesh is a rectangular block, since the points in a incomplete row may be distributed to the PEs in the regular block with at most constant increases in the number of points handled per PE. See Fig. 5.) **go to** the line marked "**".

For each new face, set *extn* to 1.

**End** of algorithm *Convex Hull*

**Theorem**: Algorithm *Convex Hull* finds for a set of $n$ points its convex hull in $O(\sqrt{n}\log n)$ time on an $O(\sqrt{n}\times\sqrt{n})$ Mesh-Connected Computer.

**Proof**: Sorting in step 2.1 takes $O(\sqrt{n})$ time. Denote the running time of algorithm *Convex Hull* as $T(n)$. $T(n)$ can be expressed by the recurrence equation $T(n) = T(n/2) + M(n)$. To find $M(n)$, assume algorithm *Represent* in step 2.3.1 is correct and has $O(\sqrt{n}\log n)$ time complexity. (This will be shown after the description of algorithm *Represent*.) Step 2.3.2 involves a constant number of the RARs and sorting steps; therefore $O(\sqrt{n})$ time is sufficient. In step 2.3.3, broadcast, and comparison in each row or column takes $O(\sqrt{n})$ time respectively. Step 2.3.3 needs to be repeated for $\log n$ iterations, thus takes $O(\sqrt{n}\log n)$ time in total. Since $M(n) = O(\sqrt{n}\log n)$, we have $T(n) = O(\sqrt{n}\log n)$. A convex hull with $n$ vertices has at most $3n$–6 edges and $2n$–4 faces. We can distribute the elements on the Mesh-Connected Computer of size $\sqrt{n}\times\sqrt{n}$, with a constant number of points, edges or faces per PE. The proof is complete subject to the correctness of the assumption about algorithm *Represent* which will be shown as follows. □

The representative face of $F_i^A$ is the face $F_r^B$ in $CH(B)$ such that angle$(F_i^A, F_r^B)$ is the smallest. Let the normal vectors of faces $F_i^A$ and face $F_j^B$ be $<a_i,b_i,c_i>$ and $<a_j,b_j,c_j>$ respectively.

$$\cos angle(F_i^A, F_j^B) = <a_i,b_i,c_i>\cdot<a_j,b_j,c_j> = (a_ia_j + b_ib_j + c_ic_j)$$

If angle$(F_i^A, F_j^B)$ is the smallest, $a_ia_j+b_ib_j+c_ic_j$ is the greatest when $0 \le angle \le \pi$. Investigate the distance between point $p_i$ with coordinate$(a_i,b_i,c_i)$ and point $p_j$ with coordinate$(a_j,b_j,c_j)$.

$$distance\ (p_i,p_j) = \sqrt{2(1-(a_ia_j+b_ib_j+c_ic_j))}$$

We can see if $a_ia_j+b_ib_j+c_ic_j$ is the greatest, $distance\,(p_i,p_j)$ is the smallest. To find for $F_i^A$ with normal vector $<a_i,b_i,c_i>$ the face $F_r^B$, we need only to find for point $p_i$ with coordinate $(a_i,b_i,c_i)$ its nearest neighbor among all the points $p_j$, with coordinates $(a_j,b_j,c_j)$ where $<a_j,b_j,c_j>$ is the normal vector of $F_j^B$.

All the points $p_j$ corresponding to faces of $CH(B)$ are on the unit sphere and we can construct a spherical Voronoi diagram for them by projecting onto the sphere the edges of the circumscribing polyhedron which is formed by the planes tangent to the sphere at $p_j$'s [4]. Performing point location for query point $p_i$, we can find its nearest neighbor. Observe the Voronoi diagram in Figure 1, we can find that it is composed by a set of broken line generated at different levels of the merging step, when the Voronoi diagram is being constructed by a divide-and-conquer approach. In Figure 6(a), the bold broken line was generated in the final ($\log n$–1 level) merge step. The fine broken line was generated one step earlier and the dashed broken line was generated two steps earlier. If we sort all the points, on whose Voronoi diagram we are locating the query point, by their $x$-coordinates in non-decreasing order so that their binary indices are $b_{\log n}\cdots b_1b_0$, each segment in the $i$-th level broken line is a perpendicular bisector of two points such that the most significant bit different between their indices are the $i$-th bit. To locate the query point $p_i$, we first want to decide whether it falls in the left half or right half of the Voronoi polygons. Then find which quarter it falls in, and so on. For the example of Fig. 6(a), we first want to decide which side of the bold broken line the query point $p_i$ lies on. Then find which side of the fine broken line it lies on and so on. To decide which side of the broken line a point lies on, first we divide the plane into horizontal slabs each containing one segment (see Fig. 6(b)). Once we find the slab the point falls in, it becomes very easy to examine which side of the segment, and thus of the broken line, point $p_i$ lies. The point location on the spherical Voronoi diagram is similar. The MCC algorithm is as follows:

*Algorithm Represent*

*input*: Two sets of faces belonging to the convex hull of $A$ and $B$ respectively, where face $F_i$ is represented by the equation $\alpha_ix + \beta_iy + \gamma_iz + \delta_i = 0$. The faces are distributed one per PE.

*output*: For each convex face in $A$ two representative convex faces in $B$, (the PE which contains a face in $A$ will have at the end of the algorithm the indices of the PEs which contain the later), and for each convex face in $B$ two representative convex faces in $A$.

2.3.1.1 A PE which contains a face $F_i^A$ ($F_j^B$ respectively) finds its normal vectors $<a_i,b_i,c_i>$ and $<-a_i,-b_i,-c_i>$ ($<a_j,b_j,c_j>$ and $<-a_j,-b_j,-c_j>$ respectively) where

$$a_i = \frac{\alpha_i}{\sqrt{\alpha_i^2+\beta_i^2+\gamma_i^2}}$$

$$b_i = \frac{\beta_i}{\sqrt{\alpha_i^2+\beta_i^2+\gamma_i^2}}$$

$$c_i = \frac{\gamma_i}{\sqrt{\alpha_i^2+\beta_i^2+\gamma_i^2}}$$

The following steps find for each point $p_i(a_i,b_i,c_i)$ and $p_{i'}(-a_i,-b_i,-c_i)$ in $A$ ($B$ respectively), its nearest neighbor among $p_j(a_j,b_j,c_j)$'s in $B$ ($A$ respectively). (They are all on the unit sphere.)

2.3.1.2 Construct the spherical Voronoi diagram for the points in $A$ and $B$ respectively. We describe the steps for the points in $A$. Steps for the points in $B$ are similar.

(a) Find for all the points $p_i$ $(a_i,b_i,c_i)$ corresponding to face $F_i^A$ the planes tangent to the unit sphere at that point. These planes will form a circumscribing polyhedron of the unit sphere. Each PE which contains a face $F_i^A$ generates the equation of the plane $a_ix + b_iy + c_iz = 1$.

(b) To find the edge in the circumscribing polyhedron, each PE containing $F_i^A$ does a RAR from the PE which contains an adjacent face $F_j^A$. The intersection of these planes

$$\begin{cases} a_ix + b_iy + c_iz = 1 \\ a_jx + b_jy + c_jz = 1 \end{cases}$$

decides an edge of the circumscribing polyhedron.

(c) By examining in convex hull $A$ all the faces adjacent to $F_i^A$ of $CH(A)$, we can determine the vertex of the circumscribing polyhedron $t_i(u_i,v_i,w_i)$'s.

(d) Find the Voronoi point $\lambda_i(\xi_i,\eta_i,\zeta_i)$ and Voronoi arc $e_i$ of the spherical Voronoi diagram. Each PE computes

$$\xi_i \leftarrow \frac{u_i}{\sqrt{u_i^2+v_i^2+w_i^2}}$$

$$\eta_i \leftarrow \frac{v_i}{\sqrt{u_i^2+v_i^2+w_i^2}}$$

$$\zeta_i \leftarrow \frac{w_i}{\sqrt{u_i^2+v_i^2+w_i^2}}$$

$$e_i \leftarrow \begin{cases} x^2+y^2+z^2=1 \\ \begin{vmatrix} \eta_1\zeta_1 \\ \eta_2\zeta_2 \end{vmatrix} \mathrm{x} - \begin{vmatrix} \xi_1\zeta_1 \\ \xi_2\zeta_2 \end{vmatrix} \mathrm{y} + \begin{vmatrix} \xi_1\eta_1 \\ \xi_2\eta_2 \end{vmatrix} \mathrm{z} = 0 \end{cases}$$

Each Voronoi arc "remembers" the two points with which it is associated.

2.3.1.3 Mark the levels for the Voronoi arcs in the spherical Voronoi diagram, as described below.

Sort all the points $p_i$'s (the points surrounded by the Voronoi arcs) by $\theta_i = x_i/\sqrt{x_i^2+y_i^2}$. Each point has its binary index.

Each PE which contains a Voronoi arc $e_i$ does a RAR to get the index of the pair of points it is associated with.

Find the "bit exclusive or", say $\Psi$, of the binary index of each pair of points associated with $e_i$. The level of $e_i$, $l_i$, can be found by $l_i = \lfloor \log\Psi \rfloor$. $[(2's\ complement(2^{l_i}) - 2^{l_i}) \wedge (index\ of\ e_i's\ associated\ point)]/2^{l_i+1}$ is the index of the chain to which $e_i$ belongs. For an example, if $e_i$ is associated with point 0010 and 0101, "bit exclusive or" of 0010 and 0101 is 0111, $\lfloor \log 0111 \rfloor = (2)_{10}$, so $e_i$ is of level 2. Furthermore, $2's\ complement(2^2) - 2^2 = 2's\ complement(0100) - 0100 = 1100 - 0100 = 1000$. $(1000 \wedge 0010)$ (or $1000 \wedge 0101$) $= 0$; so $e_i$ is indexed as 0 in the chains of level 2.

2.3.1.4 **From** level $i = \log n - 1$, $i = i - 1$, **until** $i = 0$,

each PE containing an unlocated query point in subset $j$ checks which side of the Voronoi arc chain $j$ of level $i$ it lies at.

(a) Sort all the points in the chain using the chain index as the major key and their $y$-coordinates as the minor key together with the query points using the index of the subset they belong as the major key and their $y$-coordinates as the minor key. Each query points records the index of the PE which it is in as *global_rank*.

(b) Sort the query points only, using the same key described above. Each query point records the index of the PE which it is in as *local_rank* and calculates

$$dest = global\_rank - local\_rank - 1.$$

Then sort all the points in the Voronoi arc chain only, using the same key described above.

(c) Each PE containing the query point does a RAR from the PE with index *dest* to get the equation of the Voronoi arc and the two points associated with it. Decide whether the Voronoi arc lies to the left/right side of the query point and record it as $bound_L/bound_R$.

(d) If a query point has $bound_L$ and $bound_R$ bounding the same region, i.e. has the same associated point, the query point has been located. Record the index of the PE in which the "owner" point of the region lies.

Let all the unlocated query points on the left/right sides of the Voronoi chain of level i be a subset whose index is $b_{\log n}\cdots b_i\cdots b_1 b_0$ with $b_i$ equal to 0/1.

*comment*: In a sorted list, the rank of an element is the number of the elements before it in the list. Given two sets $P\{p_i\}$ and $Q\{q_j\}$, if they are sorted together, *global_rank* of an element indicates the sum of the number of $p_i$'s and $q_j$'s previous to it. Its *local_rank* indicates the number of $q_i$'s previous to it only. So the *global_rank* – *local_rank* indicates the number of $p_i$'s previous to it, and the one closest to it has its own *local_rank* equal to *global_rank* – *local_rank* – 1. For the details, see [8].

**End** of algorithm *Represent*

**Theorem**: Algorithm *Represent* finds for each face in $A$ the representative faces in $B$ in $O(\sqrt{n}\log n)$ time on an $O(\sqrt{n}\times\sqrt{n})$ mesh.

**Proof**: Step 2.3.1.1 needs only constant time. In step 2.3.1.2, step (a), (c) and (d) each takes constant time and (b) needs $O(\sqrt{n})$ time for RAR. Thus step 2.3.1.2 uses $O(\sqrt{n})$ time to finish the construction of the spherical Voronoi diagram. Step 2.3.1.3 involves a sort and a RAR each requiring $O(\sqrt{n})$ time. For step 2.3.1.4, sorting in (a) and (b) each takes $O(\sqrt{n})$ time; RAR in (c) takes $O(\sqrt{n})$ time and (d) needs constant time only. Step 2.3.1.4 will be executed for $\log n$ iterations, thereby requiring $O(\sqrt{n}\log n)$ time. Thus the time complexity of algorithm *Represent* is $O(\sqrt{n}\log n)$. The number of the edges in a Voronoi diagram constructed for $n$ points is $O(n)$. If there are $k$ Voronoi arc chains of level $i$ each with $O(n)$ arcs, there will be $2k$ of Voronoi arc chains of level $i-1$ each with $O(n/2)$ arcs. So we can always distribute the $O(n)$ vertices or arcs in the Voronoi chain together with $O(n)$ query points on an $O(\sqrt{n}\times\sqrt{n})$ mesh, such that each PE has only a constant number of arcs (vertices).

We have presented an algorithm to construct the Voronoi diagram for a set of $n$ planar points on a $\sqrt{n}\times\sqrt{n}$ MCC with constant storage per PE, and proved that its time performance is $O(\sqrt{n}\log n)$.

## III. Applications of the Voronoi Algorithm

The Voronoi diagram can be used to solve many geometrical problems such as finding the nearest neighbors between two sets and constructing the minimum spanning trees.

Given two sets $P$ and $Q$ with $p$ and $q$ points respectively, the problem of all nearest neighbors between two sets is to find for each point in $Q$ its nearest neighbor in $P$. To solve this problem on MCC, we first construct the Voronoi diagram for points in $P$, which takes $O(\sqrt{p}\log p)$ time. Then we perform point location for each point in $Q$ using the same method described in algorithm *Represent*. The nearest neighbor of each point in $Q$ can be found in $O(\sqrt{p+q}\log p)$ time.

Given a set of points in the plane, the minimum spanning tree of it is a subgraph of the Voronoi dual [2]. Based on a Voronoi diagram, the Voronoi dual can be constructed in constant time on the $(\sqrt{n}\times\sqrt{n})$ mesh by joining each pair of points which are associated with a Voronoi edge. The dual is a planar graph. Using, for instance, the mesh algorithm proposed in [12], the minimum spanning tree for a planar graph can be constructed in $O(\sqrt{n}\log n)$ time. Thus the time complexity of the algorithm solving the Euclidean minimum spanning tree problem on a $\sqrt{n}\times\sqrt{n}$ mesh is $O(\sqrt{n}\log n)$.

## IV. Summary

We have presented the algorithm for constructing the Voronoi diagram of a set of planar points on a Mesh-Connected Computer having constant number storage per PE. Given a set of $n$ planar points distributed on an $O(\sqrt{n}\times\sqrt{n})$ Mesh-Connected Computer, our algorithm can find the Voronoi diagram for the set in $O(\sqrt{n}\log n)$ time.

We have also discussed some geometrical problems which can be solved on the Mesh-Connected Computer once the Voronoi diagram has been determined. The problem of determining the nearest neighbors between two sets and constructing Euclidean minimum spanning trees can also be solved with the same time complexity.

## Reference

[1] G. Voronoi, "Nouvelles applications des parametres continues á la théorie des formes quadratiques," J. reine angew. Math., Vol. 134, (1908), pp. 198-287.

[2] M. I. Shamos, "Computational geometry," Ph.D. Dissertation , Dept. of Comput. Sci., Yale Univ., New Haven, CT, (1978).

[3] M. I. Shamos and D. Hoey, "Closest point problems," Proc. of the 16th IEEE Symposium of Foundations of Computing , (1975), pp. 151-162.

[4] K. Q. Brown, "Voronoi diagrams from convex hulls," Inform. Processing Lett., Vol. 9, (1979), pp. 223-228.

[5] A. L. Chow, "Parallel algorithms for geometric problems," Ph.D. Dissertation , Dept. of Comput. Sci., University of Illinois, Urbana, Illinois, (1980).

[6] A. Aggarwal, B. Chazelle, L Guibas, C. Ó'Dúnlaing and C. Yap, "Parallel computational geometry," Proc. of Symposium on Foundation of Computer Science , (1985), pp. 468-477.

[7] R. Miller and Q. F. Stout, "Computational geometry on a mesh-connected computer," Proc. of 1984 Int. Conf. on Parallel Processing , (1984), pp. 66-73.

[8] M. Lu and P. Varman, "Solving geometric proximity problems on mesh-connected computers," Proc. of 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management , (Nov. 1985), pp. 248-255.

[9] M. Lu and P. Varman, "Mesh-connected computer algorithms for rectangle-intersection problems," Proc. of 1986 Int. Conf. on Parallel Processing , (1986).

[10] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," Communications of ACM , Vol. 20(4), (Apr. 1977), pp. 263-271.

[11] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," IEEE Trans. Computers , Vol. C-30(2) (Feb. 1981), pp. 101-106.

[12] P. S. Gopalakrishnan, I. V. Ramakrishnan, and L. N. Kanal, "An efficient connected components algorithm on a mesh-connected computer," Proc. of 1985 Int. Conf. on Parallel Processing , (1985), pp.711-714.

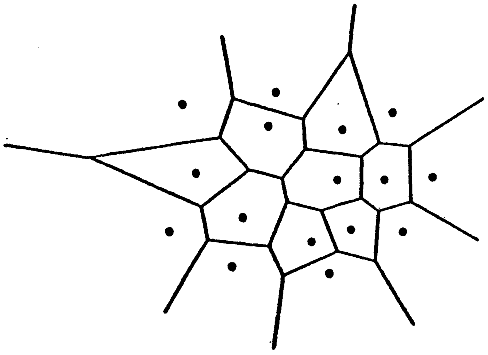


Figure 1

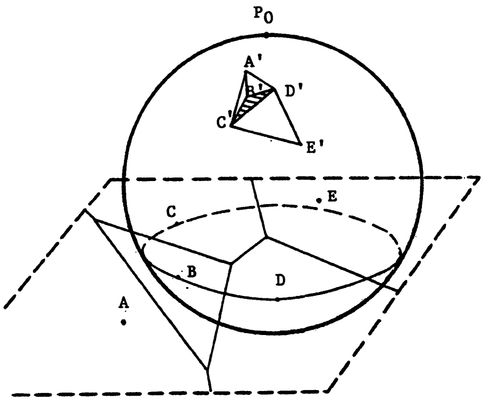


Figure 2

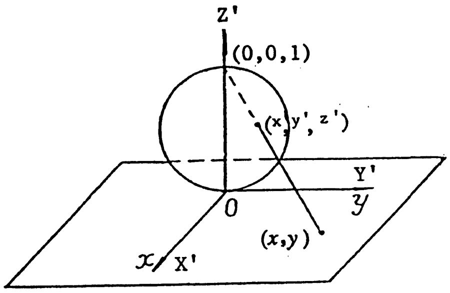
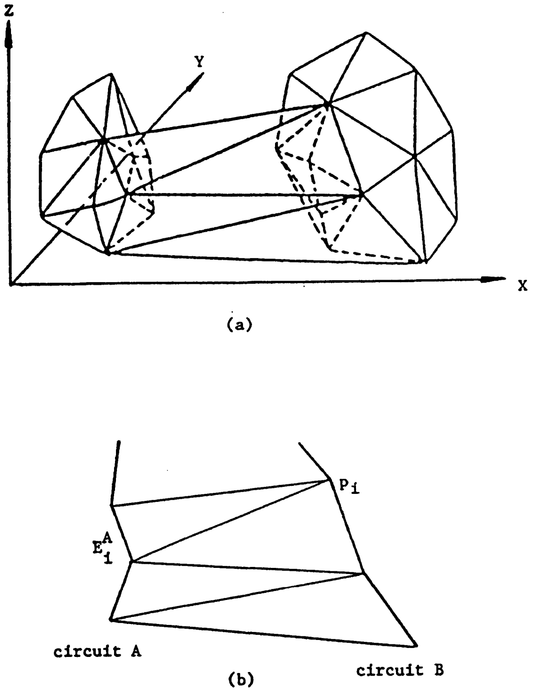


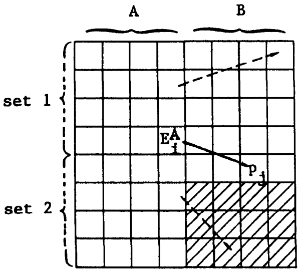
Figure 3



(a)

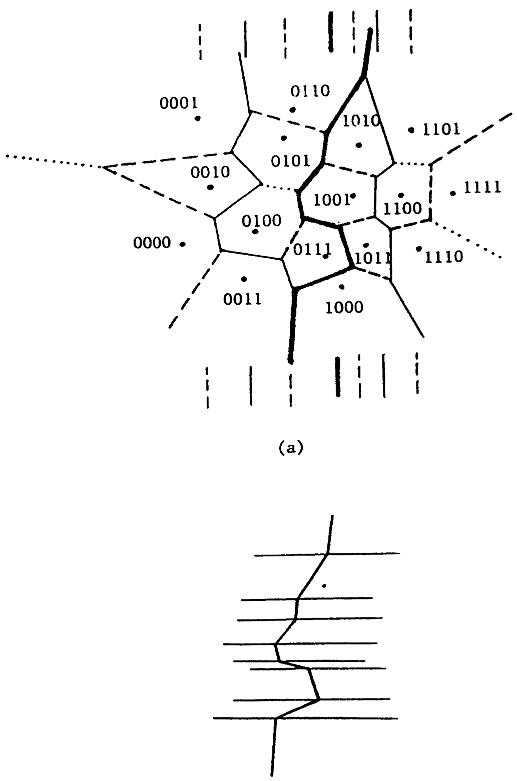
(b)

Figure 4



Originally $B$ is a submesh associated with the set of edges stored in $A$. $E_i^A$, the edge with the middle index, finds $p_j$ which divides mesh $B$ into two submeshes, the white one and the shaded one. Thus, the white submesh is associated with *set* 1 and the shaded one with *set* 2. The dashed arrows indicate the search in the next iteration.

Figure 5



(a)

(b)

Figure 6

# OPTIMAL ALGORITHMS FOR MESH-CONNECTED PARALLEL PROCESSORS WITH SERIAL MEMORIES*

by

Joseph Ja'Ja' and Robert Michael Owens†

Department of Electrical Engineering and Institute For Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Abstract -- We examine the problems of computing the discrete Fourier transform and performing matrix multiplication on an array of processors, where each processor consists of an ALU, a constant($\leq 3$) number of storage registers, and has a single serial memory attached to it. We show that the discrete Fourier transform can be computed optimally on one or two-dimensional arrays with $p$ processors in time $\Theta(n + \frac{n^2}{p^2})$ and $\Theta(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2})$ respectively. For the problem of multiplying two $n \times n$ matrices, we show the optimal bound of $\Theta(n^2 + \frac{n^4}{p^2})$ for a linear array of $p$ processors and a lower bound of $\Omega(\frac{n^2}{\sqrt{p}} + \frac{n^4}{p^2})$ and an upper bound of $O(\frac{n^3}{p} + \frac{n^4}{p^2})$ for a $\sqrt{p} \times \sqrt{p}$ array of processors. We also settle the case of multiplying a matrix by a vector in time $\Theta(\frac{p}{n} + \frac{n^2}{p})$. Optimal bounds are also established for the corresponding bit model. Our lower bound techniques use a combination of crossing sequence and communication complexity arguments.

## 1. Introduction

In an attempt to achieve the maximum possible speedups, researchers have studied various architectures and examined their suitability to solve different classes of problems. Many schemes for specific problems have been developed and have been shown to achieve optimal speeds on these architectures. However almost all of these designs require that the number of processing elements must grow linearly with the size of the problem, a condition that makes the hardware cost too high.

We examine, in this paper, a simple array architecture (one and two-dimensional) with a fixed number of processors such that a serial memory is attached to each processor. Serial memories appear in magnetic disk, magnetic bubble, charge coupled devices, and even VLSI shift registers. Because of their high information density and low cost, serial memories seem to be quite attractive for use in applications requiring the processing of a large amount of data. However, their overall performance seems to be severely limited due to the fact that the elements can be only accessed in one particular order. A similar model has been studied by several authors ([BL], [BW], [CLW], [CW], [MTW], [OJ], [TW], [WC], [W]). In particular, it was shown in ([OJ]) that sorting $n$ numbers on a linear array of $p$ processors (comparators) can be done in time $\Theta(n + \frac{n^2}{p^2})$, and in time $\Theta(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2})$ on a $\sqrt{p} \times \sqrt{p}$ array of processors. These results show that the performance of serial memories is comparable to that of random access memories whenever $p$ is large enough. In this paper, we consider numerical problems that have high computational requirements. Assuming that each processor is an ALU, we show that computing the discrete Fourier transform on $n$ points requires $\Theta(n + \frac{n^2}{p^2})$ steps in the word model, and $\Theta(n\,q + \frac{n^2q^2}{p^2})$ in the bit model, where each number consists of $q$ bits. For the case of a $\sqrt{p} \times \sqrt{p}$ array of processors, we show the optimal bounds of $\Theta(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2})$ and $\Theta(\frac{nq}{\sqrt{p}} + \frac{n^2q^2}{p^2})$ for the word and bit models respectively. The lower bound techniques use a combination of crossing sequence and communication complexity arguments. We also consider the problem of multiplying two $n \times n$ matrices and exhibit matching upper and lower bounds of $\Theta(n^2 + \frac{n^4}{p^2})$ for a linear array of $p$ processors, and a lower bound of $\Omega(\frac{n^2}{\sqrt{p}} + \frac{n^4}{p^2})$ and an upper bound of $O(\frac{n^3}{p} + \frac{n^4}{p^2})$ for a $\sqrt{p} \times \sqrt{p}$ array of processors. The problem of multiplying a matrix by a vector is shown to require $\Theta(\frac{p}{n} + \frac{n^2}{p})$ steps.

The rest of the paper is organized as follows. The formal model is introduced in the next section, while the upper and lower bounds for the discrete Fourier transform are presented in section 3. The matrix multiplication problem is considered in section 4 and the last section is devoted to the problem of multiplying a matrix by a vector.

## 2. The Model

We will consider a network of $p$ interconnected processors, where each processor $\mathbf{P}_i$, $0 \leq i < p$, consists of one ($\mathbf{R}_i$ in the case of the discrete Fourier transform) or more ($\mathbf{A}_i$, $\mathbf{B}_i$, and $\mathbf{C}_i$ in the case of matrix multiplication) storage registers and an ALU capable of adding, subtracting, and multiplying numbers. A single $q$ bit number can be stored in each

-----------------

* Supported by the U.S. Army Research Office, Contracts No. GAAG29-82-K-0110 and DAAG29-83-K-0126.

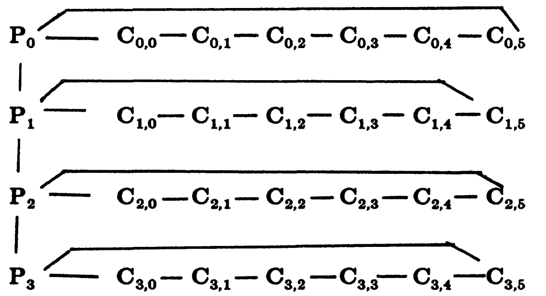
† Current address: Department of Computer Science, Penn State University, University Park, PA 16802.

storage register. Attached to each processor $\mathbf{P}_\mathbf{i}$ is a single serial memory $\mathbf{S}_\mathbf{i}$. Each serial memory $\mathbf{S}_\mathbf{i}$ consists of a linearly connected array of storage cells $\mathbf{C}_{\mathbf{i},\mathbf{j}}$, $0 \leq j < l$. Either a single $q$ bit number (the *word* model) or a bit (the *bit* model) can be stored in each cell $\mathbf{C}_{\mathbf{i},\mathbf{j}}$. The length of each serial memory $\mathbf{S}_\mathbf{i}$ is $l$.

When a serial register $\mathbf{S}_\mathbf{i}$ is clocked, the content of cell $\mathbf{C}_{\mathbf{i},\mathbf{j}}$, $1 \leq j < l$, is transferred to cell $\mathbf{C}_{\mathbf{i},\mathbf{j}-1}$. Furthermore, the input to serial memory $\mathbf{S}_\mathbf{i}$ is transferred to cell $\mathbf{C}_{\mathbf{i},\mathbf{l}-1}$ and, hence, to the serial memory itself. The output of the serial memory is the content of cell $\mathbf{C}_{\mathbf{i},0}$.

Suppose processor $\mathbf{P}_\mathbf{i}$ is connected to processors $\mathbf{P}_{\mathbf{i}_j}$, $0 \leq j < n_i$. Then, at the beginning of each step, processor $\mathbf{P}_\mathbf{i}$ has available to it the contents of the storage register(s) of processors $\mathbf{P}_{\mathbf{i}_j}$ $0 \leq j < n_i$, the contents of its own storage register(s), and the output of its serial memory. At the end of each step, each processor is expected to supply a number to its serial memory and possibly change the contents of its storage registers.

In this paper we will consider two particular interconnections. In the first case, the processors are interconnected to form a linear array of length $p$ ( *linear* model). Figure 2.1 illustrates this configuration for $p = 4$ and $n = 24$.



$P_0$
$C_{0,0}$ — $C_{0,1}$ — $C_{0,2}$ — $C_{0,3}$ — $C_{0,4}$ — $C_{0,5}$
$P_1$
$C_{1,0}$ — $C_{1,1}$ — $C_{1,2}$ — $C_{1,3}$ — $C_{1,4}$ — $C_{1,5}$
$P_2$
$C_{2,0}$ — $C_{2,1}$ — $C_{2,2}$ — $C_{2,3}$ — $C_{2,4}$ — $C_{2,5}$
$P_3$
$C_{3,0}$ — $C_{3,1}$ — $C_{3,2}$ — $C_{3,3}$ — $C_{3,4}$ — $C_{3,5}$

Figure 2.1 Linearly Connected Array.

In the second, the processors are interconnected to form a $\sqrt{p} \times \sqrt{p}$ array ( *two-dimensional* model).

Our model is consistent with most of the architectures which have used serial memories as the principal storage memory. In particular the model is consistent with most architectures which use rotating magnetic disks, charge coupled devices, bubble memories, and equal length VLSI shift registers.

## 3. The Discrete Fourier Transform

The Discrete Fourier Transform, DFT, of $n$ points $x_0$, $x_1$, $\cdots$, $x_{n-1}$ is given by:

$$y_i = \sum_{j=0}^{n-1} w^{ij} x_j, \quad 0 \leq i < n, \qquad <DFT(n)>,$$

where $w$ is a primitive $n$'th root of unity. In matrix form,

$$\mathbf{y} = \mathbf{W}\mathbf{x},$$

where $\mathbf{x} = \left[x_i, 0 \leq i < n\right]^T$, $\mathbf{y} = \left[y_i, 0 \leq i < n\right]^T$, and $\mathbf{W} = \left[w^{ij}, 0 \leq i, j < n\right]$.

In this section we will show that $DFT(n)$ can be computed in time $\Theta(n + \frac{n^2}{p^2})$ on a linear array of $p$ processors and $\Theta(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2})$ on a $\sqrt{p} \times \sqrt{p}$ array of processors. In both cases, a serial memory of length $l = \frac{n}{p}$ is attached to each processor. To establish the upper bound for the case of a linear array, we will do the following:

1) show how $DFT(n)$ can be computed on a $p \times \frac{n}{p}$ array of processors using $O(n)$ steps; and

2) show how each step of the array can be emulated using $\frac{n}{p}$ steps on our linear model.

The fast Fourier transform (decimation in frequency scheme [BB]) can be computed on a $p \times \frac{n}{p}$ array of processors in time $O(p + \frac{n}{p})$. In this case $p$ and $n$ are assumed to be both powers of two. From [JO2], we see that this algorithm consists of $\log n$ steps. Step $i$, $1 \leq i < \log p$, involves a column shuffle of length $\frac{p}{2^i}$, the evaluation of expressions of the form $a\,x + b\,y$ at each processor, and a column unshuffle of length $\frac{p}{2^i}$. Step $i$, $\log p \leq i < \log n$, involves a row shuffle of length $\frac{n}{2^i}$, the evaluation of expressions of the form $a\,x + b\,y$ at each processor, and a row unshuffle of length $\frac{n}{2^i}$. From figure 3.1, we see that a column[row] shuffle/unshuffle of two sequences of length $i$ can be performed using $i - 1$ column[row] exchange operations in a rectangular array.

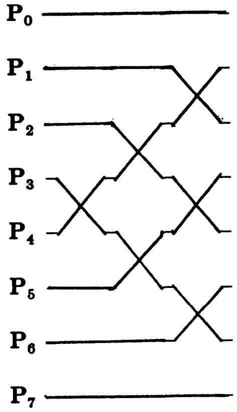


Figure 3.1 Shuffling.

Hence, $DFT(n)$ can be computed on a rectangular array using $O(p + \frac{n}{p})$ steps.

We now consider the emulation of a $p \times \frac{n}{p}$ rectangular array on our model. To emulate the array, we use a linear array of $p$ processors. A shift register of length $\frac{n}{p}$ is attached to each processor. In the emulation the contents of storage cell $\mathbf{C}_{\mathbf{i},\mathbf{j}}$, $0 \leq i < p$, $0 \leq j < \frac{n}{p}$ represent the state of processor $\mathbf{P}_{\mathbf{i},\mathbf{j}}$ of the array being emulated. From [OJ], we see that each step on the rectangular array can be emulated using $O(\frac{n}{p})$ steps on our linear model by shifting the contents of all the shift registers through the processors once. Hence, the discrete Fourier transform of $n$ points can be computed using $O((p + \frac{n}{p})\frac{n}{p}) = O(n + \frac{n^2}{p^2})$ steps on our linear model.

The weights used in the algorithm can either be serially supplied to the processors or computed. Computing the weights requires doubling the length of the serial memories and adding another storage register to each processor ([JO2]).

In the case of the two-dimensional array, it is straightforward to show a bound of $O((\sqrt{p} + \frac{n}{p})\frac{n}{p}) = O(\frac{n}{\sqrt{p}} + \frac{n^2}{p^2})$ steps using an emulation of an algorithm for a $\sqrt{p} \times \sqrt{p} \times \frac{n}{p}$ three dimensional array of processors.

Note that if we assume that a RAM is attached to each processor, we can obtain the bounds of $O(n)$ (linear array) and $O(\frac{n}{\sqrt{p}})$. Hence, if at least $\Omega(\sqrt{n})$ (linear array) or $\Omega(n^{\frac{2}{3}})$ (two-dimensional array) processors are used, a RAM based architecture is not superior to a serial memory based architecture.

To establish the lower bounds, we will elaborate on the basic model. We state precisely the conditions we need for the proof.

(1) Each serial memory $\mathbf{S}_l$ contains $\frac{n}{p}$ numbers, where $2p$ evenly divides $n$. The numbers are initially assumed to be stored in a predetermined order independent of their values.

(2) Each number consists of $q$ bits and each step operates on a number as a single entity.

(3) The serial memories are unidirectional.

(4) A general step of the algorithm at processor $\mathbf{P}_l$ consists of:

   (a) computing two quantities $\alpha_i$ and $\beta_i$ that depend only on the output of $\mathbf{S}_l$, $\mathbf{R}_l$, $\mathbf{R}_{l-1}$, and $\mathbf{R}_{l+1}$.

   (b) updating $\mathbf{R}_l$ with $\alpha_i$ and shifting $\beta_i$ into $\mathbf{S}_l$.

(5) The steps of all the serial memories are synchronous.

(6) The output numbers will reside in the serial memories in a predetermined order independent of their values.

We will call a *cycle* a set of $\frac{n}{p}$ consecutive shifts. Without loss of generality, we assume that the output is generated after precisely $c$ cycles, i.e., after $c\,\frac{n}{p}$ steps, for some positive integer $c$.

Theorem 3.1: Under assumptions (1) - (6), computing the discrete Fourier transform of $n$ points on $p$ linearly connected processors with serial memories requires $\Omega(\frac{n^2}{p^2} + n)$ steps. For a $\sqrt{p} \times \sqrt{p}$ array of processors, we need at least $\Omega(\frac{n^2}{p^2} + \frac{n}{\sqrt{p}})$ steps.

Proof: With each set of input values, we associate two sequences of $p$-tuples which we call *cycle sequences*. The first sequence can be defined as the contents $< \alpha_1^{(i)}, \alpha_2^{(i)}, \cdots, \alpha_p^{(i)} >$ of the R registers (of the processors) at the steps $\frac{n}{2p} + i\frac{n}{p}$, $0 \leq i < c$, where $c$ is the number of cycles as defined above. The second sequence is the contents $< \beta_1^{(i)}, \beta_2^{(i)}, \cdots, \beta_p^{(i)} >$ of the R registers at steps $i\frac{n}{p}$, $1 \leq i < c$. Note that the number of possible distinct cycle sequences is $2^{2qpc}$. We will next establish a lower bound on this quantity.

Recall that the $n$-point discrete Fourier transform of $\mathbf{x}$ is given by

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

Let $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$ be such that $\mathbf{x}_1$ is the length $\frac{n}{2}$ vector corresponding to the contents of the lower half of the serial memories and $\mathbf{x}_2$ is the length $\frac{n}{2}$ vector corresponding to the contents of the upper half of the serial memories. Similarly, we can define the two vectors (again each of length $\frac{n}{2}$) $\mathbf{y}_1$ and $\mathbf{y}_2$ which form the output vector. We can reorder the rows and columns of $\mathbf{W}$ to obtain a matrix $\overline{\mathbf{W}}$ such that

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \overline{\mathbf{W}} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{W}}_{1,1} & \overline{\mathbf{W}}_{1,2} \\ \overline{\mathbf{W}}_{2,1} & \overline{\mathbf{W}}_{2,2} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$$

Let $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$ and $\bar{\mathbf{x}} = \begin{bmatrix} \bar{\mathbf{x}}_1 \\ \bar{\mathbf{x}}_2 \end{bmatrix}$ be two input vectors which have the same cycle sequences. Let $\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}$ and $\bar{\mathbf{y}} = \begin{bmatrix} \bar{\mathbf{y}}_1 \\ \bar{\mathbf{y}}_2 \end{bmatrix}$ be the corresponding output vectors. Let's examine the output vector corresponding to the input vector $\begin{bmatrix} \mathbf{x}_1 \\ \bar{\mathbf{x}}_2 \end{bmatrix}$

For the first $\frac{n}{2p}$ steps, the algorithm works exactly as in the case of the input $\mathbf{x}$. At the beginning of the $(\frac{n}{2p} + 1)$'th step, the situation is identical to the beginning of the $(\frac{n}{2p} + 1)$'th step on input $\bar{\mathbf{x}}$ since $\mathbf{x}$ and $\bar{\mathbf{x}}$ have same cycle sequences. Hence, the algorithm behaves exactly as in the case of the input $\bar{\mathbf{x}}$ for the next $\frac{n}{2p}$ steps. Again, at the $(\frac{n}{p} + 1)$'th step, the situation is identical to that of input $\mathbf{x}$ at the $(\frac{n}{p} + 1)$'th step, and so on. Therefore, the output will be $\begin{bmatrix} \mathbf{y}_1 \\ \bar{\mathbf{y}}_2 \end{bmatrix}$, i.e., we have

$$\begin{bmatrix} \mathbf{y}_1 \\ \bar{\mathbf{y}}_2 \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{W}}_{1,1} & \overline{\mathbf{W}}_{1,2} \\ \overline{\mathbf{W}}_{2,1} & \overline{\mathbf{W}}_{2,2} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \bar{\mathbf{x}}_2 \end{bmatrix}$$

It follows that

$$\mathbf{y}_1 = \overline{\mathbf{W}}_{1,1}\mathbf{x}_1 + \overline{\mathbf{W}}_{1,2}\mathbf{x}_2 = \overline{\mathbf{W}}_{1,1}\mathbf{x}_1 + \overline{\mathbf{W}}_{1,2}\bar{\mathbf{x}}_2$$

$$\bar{\mathbf{y}}_1 = \overline{\mathbf{W}}_{2,1}\bar{\mathbf{x}}_1 + \overline{\mathbf{W}}_{2,2}\bar{\mathbf{x}}_2 = \overline{\mathbf{W}}_{2,1}\mathbf{x}_1 + \overline{\mathbf{W}}_{2,2}\bar{\mathbf{x}}_2$$

But this implies that

$$\overline{\mathbf{W}}_{1,2}(\mathbf{x}_2 - \overline{\mathbf{x}}_2) = 0$$

$$\overline{\mathbf{W}}_{2,1}(\mathbf{x}_1 - \overline{\mathbf{x}}_1) = 0 \qquad (*).$$

Let $rank(\overline{\mathbf{W}}_{1,2}) = \mu$ and $rank(\overline{\mathbf{W}}_{2,1}) = \lambda$. We know that ([T]) $\mu + \lambda \geq \frac{n}{2}$, regardless of how $\mathbf{W}$ is partitioned. Hence, $range\,\{\overline{\mathbf{W}}_{1,2}\,\mathbf{u}\}$ has $2^{\mu q}$ distinct points, say $\{\,\overline{\mathbf{W}}_{1,2}\,\mathbf{u}_1, \overline{\mathbf{W}}_{1,2}\,\mathbf{u}_2, \cdots, \overline{\mathbf{W}}_{1,2}\,\mathbf{u}_l, \cdots\,\}$ and $range\,\{\overline{\mathbf{W}}_{2,1}\,\mathbf{v}\}$ has $2^{\lambda q}$ distinct points, say $\{\,\overline{\mathbf{W}}_{2,1}\,\mathbf{v}_1, \overline{\mathbf{W}}_{2,1}\,\mathbf{v}_2, \cdots, \overline{\mathbf{W}}_{2,1}\,\mathbf{v}_l, \cdots\,\}$. Consider the input vectors $\begin{bmatrix}\mathbf{v}_j\\ \mathbf{u}_l\end{bmatrix}$ and $\begin{bmatrix}\hat{\mathbf{v}}_j\\ \hat{\mathbf{u}}_l\end{bmatrix}$ such that $(j, i) \neq (\hat{j}, \hat{i})$. In this case (*) does not hold. Hence any two such vectors must have two distinct cycle sequences. The number of such vectors is at least $2^{q\frac{n}{2}}$. Therefore we must have

$$2^{2pcq} \geq 2^{q\frac{n}{2}}$$

Hence, $c \geq \frac{n}{4p}$. >From this it follows that $T = \Omega(\frac{n^2}{p^2})$.

We now show that in the case of a linear array $\Omega(n)$ steps are required no matter how large $p$ is. Let's look at the link between processor $\mathbf{P}_{\frac{p}{2}}$ and processor $\mathbf{P}_{\frac{p}{2}+1}$. Half the input numbers are to the left of that link and the rest are to the right. Computing $DFT(n)$ requires $\Omega(n)$ communication between these two halves. Therefore, $\Omega(n)$ steps are required. The same argument will give $\Omega(\frac{n}{\sqrt{p}})$ for a $\sqrt{p} \times \sqrt{p}$ array of processors. □

Note that if we assume that RAM's are attached to the processors, we get the lower bounds of $\Omega(n)$ and $\Omega(\frac{n}{\sqrt{p}})$ for the linear and two-dimensional arrays respectively.

We can also establish similar lower bounds even if we assume that each number need not be treated as an atomic unit. We modify assumptions (1), (2), and (6) as follows.

(1') Each serial memory $\mathbf{S}_l$ contains $\frac{nq}{p}$ bits of the input, where $2p$ evenly divides $nq$. The bits are stored in a predetermined order independent of their values.

(2') Each step operates on single bits.

(6') The bits of the output will reside in the serial memories in a predetermined order independent of their values. The bits of an output number are stored in consecutive locations in a single serial memory.

We now define a *cycle* to be a set of $\frac{nq}{p}$ consecutive steps.

Theorem 3.2: Under the modified assumptions, computing the discrete Fourier transform of $n$ points, each consisting of $q$ bits, on $p$ linearly connected processors with serial memories requires $\Omega(\frac{n^2q^2}{p^2} + nq)$ steps. Under the same assumptions, we have the lower bound $\Omega(\frac{n^2q^2}{p^2} + \frac{nq}{\sqrt{p}})$ for a $\sqrt{p} \times \sqrt{p}$ array of processors.

Proof: The proof is similar to that of theorem 1.1 with the following modifications. Partition the elements of $\mathbf{x}$ into two subsets $\mathbf{L}$ and $\mathbf{U}$ such that $|\mathbf{U}|, |\mathbf{L}| \geq \frac{n}{3}$ and $x_i \in \mathbf{L}$ ($x_i \in \mathbf{U}$) implies that at least $\frac{1}{4}$ of the bits of $x_i$ are initially stored in the lower (upper) half of the serial memories.

Now let $\mathbf{x} = \begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix}$ be such that $\mathbf{x}_1$ and $\mathbf{x}_2$ are the vectors corresponding to the elements in $\mathbf{L}$ and $\mathbf{U}$ respectively. Let $\mathbf{y} = \begin{bmatrix}\mathbf{y}_1\\ \mathbf{y}_2\end{bmatrix}$ be defined as before. We thus have

$$\begin{bmatrix}\mathbf{y}_1\\ \mathbf{y}_2\end{bmatrix} = \begin{bmatrix}\overline{\mathbf{W}}_{1,1} & \overline{\mathbf{W}}_{1,2}\\ \overline{\mathbf{W}}_{2,1} & \overline{\mathbf{W}}_{2,2}\end{bmatrix} \begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix}$$

after reordering the rows and columns of the matrix $\mathbf{W}$. Notice that the bits of $\mathbf{x}_1$ are not necessarily stored initially in the lower half of the serial memories. However, we know that at least $\frac{q}{4}$ bits of each element of $\mathbf{x}_1$ are initially stored in the lower half of the serial memories. Consider all the possible input values for which all the bits of any element of $\mathbf{x}_1$, not in the lower half of the serial memories, are set to zero, and all the bits of any element of $\mathbf{x}_2$, not in the upper half, are set to zero. The number of such input values is at least $2^{\frac{nq}{4}}$. We now follow the same argument as in theorem 3.1 in identifying those values which yield identical cycle sequences.

Notice that if $rank(\overline{\mathbf{W}}_{2,1}) = \mu$, then there exists nonsingular matrices $\mathbf{P}$ and $\mathbf{Q}$ such that

$$\mathbf{P}\,\overline{\mathbf{W}}_{2,1}\,\mathbf{Q} = diag\,\{1, 1, \cdots, 1, 0, 0, \cdots, 0\}\ ,$$

where the number of 1's is equal to $\mu$. Hence, if $\mathbf{u}$ and $\mathbf{v}$ have different first $\mu$ entries, then

$$\mathbf{P}\,\overline{\mathbf{W}}_{2,1}\,\mathbf{Q}\,\mathbf{u} \neq \mathbf{P}\,\overline{\mathbf{W}}_{2,1}\,\mathbf{Q}\,\mathbf{v}$$

But this implies that $\overline{\mathbf{W}}_{2,1}(\mathbf{Q}\,\mathbf{u}) \neq \overline{\mathbf{W}}_{2,1}(\mathbf{Q}\,\mathbf{v})$ and thus $range\,(\overline{\mathbf{W}}_{2,1}\mathbf{u})$ contains at least $2^{\frac{\mu q}{4}}$ points in this case.

## 4. Matrix Matrix Multiplication

The product of two $n \times n$ matrices $\mathbf{A} = \left[a_{i,j},\, 0 \leq i, j < n\right]$ and $\mathbf{B} = \left[b_{i,j},\, 0 \leq i, j < n\right]$ is given by:

$$\mathbf{C} = \left[c_{i,j} = \sum_{k=0}^{n} a_{i,k}\, b_{k,j},\, 0 \leq i, j < n\right]$$

In this section we will show that this product can be computed in time $\Theta(n^2 + \frac{n^4}{p^2})$ on a linear array of $p$ processors and in time $O(\frac{n^3}{p} + \frac{n^4}{p^2})$ on a $\sqrt{p} \times \sqrt{p}$ array of processors. In both cases, a serial memory of length $l = \frac{n^2}{p}$ is attached to each processor. To establish the upper bound for the case of a linear array, one can do the following:

1) present an algorithm which computes the product of two $n \times n$ matrices on an $n \times n$ cyclic square array (an array with cyclic end around connections) using $O(n)$ steps;

2) show how each step on the square array can be emulated using $O(\frac{p}{n} + \frac{n}{p})$ steps on a $p \times l$ cyclic rectangular array;

3) show how each step on the rectangular cyclic array can be emulated using $O(1)$ steps on a rectangular noncyclic array; and

4) show how each step on a rectangular noncyclic array can be emulated using $O(\frac{n^2}{p})$ steps on our model.

We leave the details of steps 1-4 to the reader.

We now establish the lower bounds.

Theorem 4.1: Under the assumptions (1)-(6) of section 3, multiplying two $n \times n$ matrices on $p$ linearly connected processors with serial memories requires $\Omega(n^2 + \frac{n^4}{p^2})$ steps. For a $\sqrt{p} \times \sqrt{p}$ array of processors, we have the lower bound of $\Omega(\frac{n^2}{\sqrt{p}} + \frac{n^4}{p^2})$.

Proof: We actually prove a slightly stronger result by establishing the lower bounds for the problem of matrix squaring. The main strategy is similar to that of theorem 4.1. We show that the number of cycle sequences must be large.

Let $\mathbf{X}$ be an $n \times n$ matrix such that each entry consists of $q$ bits and let $\mathbf{Y} = \mathbf{X}^2$. Let $\mathbf{X} = \mathbf{X}_l + \mathbf{X}_u$, where $\mathbf{X}_l$ and $\mathbf{X}_u$ correspond to the entries initially stored in the lower and upper halves respectively of the serial memories. Suppose that $\overline{\mathbf{X}} = \overline{\mathbf{X}}_l + \overline{\mathbf{X}}_u$ generates the same set of cycle sequences as $\mathbf{X}$. If $\mathbf{Y}_l$ and $\mathbf{Y}_u$ denote the output arrays corresponding respectively to the lower and upper halves of the serial memories, then it is easy to see that we have the following:

$$\mathbf{Y}_l + \mathbf{Y}_u = \left(\mathbf{X}_l + \mathbf{X}_u\right)^2 = \mathbf{X}_l^2 + \mathbf{X}_u^2 + \mathbf{X}_l\mathbf{X}_u + \mathbf{X}_u\mathbf{X}_l$$

$$\overline{\mathbf{Y}}_l + \overline{\mathbf{Y}}_u = \left(\overline{\mathbf{X}}_l + \overline{\mathbf{X}}_u\right)^2 = \overline{\mathbf{X}}_l^2 + \overline{\mathbf{X}}_u^2 + \overline{\mathbf{X}}_l\overline{\mathbf{X}}_u + \overline{\mathbf{X}}_u\overline{\mathbf{X}}_l$$

$$\mathbf{Y}_l + \overline{\mathbf{Y}}_u = \left(\mathbf{X}_l + \overline{\mathbf{X}}_u\right)^2 = \mathbf{X}_l^2 + \overline{\mathbf{X}}_u^2 + \mathbf{X}_l\overline{\mathbf{X}}_u + \overline{\mathbf{X}}_u\mathbf{X}_l$$

$$\overline{\mathbf{Y}}_l + \mathbf{Y}_u = \left(\overline{\mathbf{X}}_l + \mathbf{X}_u\right)^2 = \overline{\mathbf{X}}_l^2 + \mathbf{X}_u^2 + \overline{\mathbf{X}}_l\mathbf{X}_u + \mathbf{X}_u\overline{\mathbf{X}}_l.$$

It is easy to deduce from the above equations the following identity:

$$\left(\mathbf{X}_l - \overline{\mathbf{X}}_l\right)\left(\mathbf{X}_u - \overline{\mathbf{X}}_u\right) + \left(\mathbf{X}_u - \overline{\mathbf{X}}_u\right)\left(\mathbf{X}_l - \overline{\mathbf{X}}_l\right) = 0 \; (*)$$

But in [JK] it was shown that $\Omega(2^{\alpha n^2 q})$ matrices exist such no pair of which satisfy (*), where $\alpha > 0$ is some constant. Therefore the number of cycle sequences must be at least as large, i.e.,

$$2^{2cpq} \geq 2^{\alpha n^2 q}$$

and hence $T = \Omega(\frac{n^4}{p^2})$. The rest of the proof follows as before.

Note that as in the case of theorem 3.2, we can establish similar results for the bit model. Moreover, it is worth mentioning that, for the case of a $\sqrt{p} \times \sqrt{p}$ array of processors, the bounds are optimal if $p \leq n$ or $p = \Theta(n^2)$. On the other hand, if $1 \leq p \leq n^2$ and only the semiring operations of addition and multiplication are allowed, then some processor will have to perform $\Omega(n^3/p)$ operations and hence we have the lower bound of $\Omega(\frac{n^3}{p} + \frac{n^4}{p^2})$ which matches the upper bound.

## 5. Matrix Vector Multiplication

The product of a $n \times n$ matrix $\mathbf{A} = \left[a_{i,j}, 0 \leq i, j < n\right]$ by a $n$ vector $\mathbf{b} = \left[b_i, 0 \leq i < n\right]^T$ is given by:

$$\mathbf{c} = \left[c_i = \sum_{j=0}^{n} a_{i,j}\, b_j, 0 \leq i < n\right]^T$$

In this section we will show that this product can be computed in time $\Theta(\frac{p}{n} + \frac{n^2}{p})$ on a linear array of $p$ processors and $\Theta(\sqrt{\frac{p}{n}} + \frac{n^2}{p})$ on a $\sqrt{p} \times \sqrt{p}$ array of processors. In both cases, a serial memory of length $l = \frac{n^2}{p}$ is attached to each processor. To establish the upper bound, we will do the following:

1) show how this product can be computed on a $p \times l$ array of processors using $O(n)$ steps; and

2) show how this product can be computed on our linear model using $O(\frac{p}{n} + \frac{n^2}{p})$ steps.

We will first consider the case where $p = n$. The algorithm for a $n \times n$ array begins with the storage registers $\mathbf{A}_{i,j}$, $\mathbf{B}_{i,j}$, and $\mathbf{C}_{i,j}$ of processor $\mathbf{P}_{i,}$, $0 \leq i, j < n$ containing $a_{i,j}$, $b_j$, and 0 respectively. Note that the elements of $\mathbf{b}$ are replicated. Figure 5.1 illustrates this initial configuration.

**A** storage registers

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $\cdots$ | $a_{0,n-1}$ |
| $a_{1,0}$ | $a_{1,1}$ | $\cdots$ | $a_{1,n-1}$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $a_{n-1,0}$ | $a_{n-1,1}$ | $\cdots$ | $a_{n-1,n-1}$ |

**B** storage registers

| | | | |
|---|---|---|---|
| $b_0$ | $b_1$ | $\cdots$ | $b_{n-1}$ |
| $b_0$ | $b_1$ | $\cdots$ | $b_{n-1}$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $b_0$ | $b_1$ | $\cdots$ | $b_{n-1}$ |

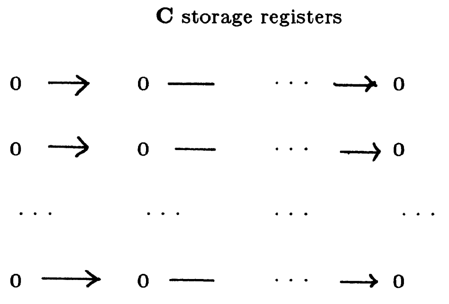


Figure 5.1 Initial Configuration for $p = n$.

The algorithm consists of $n$ steps. At the $k$'th step,

1) the contents of the $\mathbf{C}$ storage registers are shifted rightward as indicated in figure 5.1, i.e., the contents of $\mathbf{C}_{i,k-1}$, $0 \leq i < n$, $1 \leq k < n$, or 0, $0 \leq i < n$, $k = 0$ are transferred to $\mathbf{C}_{i,k}$; and
2) the content of $\mathbf{C}_{i,k}$, $0 \leq i < n$, is replaced by $\mathbf{A}_{i,k}\,\mathbf{B}_{i,k} + \mathbf{C}_{i,k}$.

It is clear that this algorithm can be emulated on our model in only $O(n)$ steps.

We now consider the case where $p < n$. In this case, we assume that $p$ evenly divides $n$. The algorithm for a $p \times n$ array begins with the storage registers $\mathbf{A}_{i,j}$, $\mathbf{B}_{i,j}$, and $\mathbf{C}_{i,j}$ of processor $\mathbf{P}_{i,j}$, $0 \leq i, j < n$ containing $a_{\hat{i},\hat{j}}$, $b_{\hat{j}}$, and 0 respectively, where $\hat{i}=\lfloor i\frac{p}{n}\rfloor$ and $\hat{j}=j+l \; mod(i,\frac{n}{p})$. The algorithm consists of $\frac{n^2}{p}$ steps. At the $k$'th step,

1) the contents of $\mathbf{C}_{i,k-1}$, $0 \leq i < n$, $0 \neq mod(k,n)$ or 0, $0 \leq i < n$, $0 \equiv mod(k,n)$ are transferred to $\mathbf{C}_{i,k}$; and
2) the content of $\mathbf{C}_{i,k}$, $0 \leq i < n$, is replaced by $\mathbf{A}_{i,k}\,\mathbf{B}_{i,k} + \mathbf{C}_{i,k}$.

It is clear that this algorithm can be emulated on our model in only $O(\frac{n^2}{p})$ steps.

We now consider the case where $p > n$. In this case, we assume that $n$ evenly divides $p$. The algorithm for a $p \times n$ array begins with the storage registers $\mathbf{A}_{i,j}$, $\mathbf{B}_{i,j}$, and $\mathbf{C}_{i,j}$ of processor $\mathbf{P}_{i,j}$, $0 \leq i, j < n$ containing $a_{\hat{i},\hat{j}}$, $b_{\hat{j}}$, and 0 respectively, where $\hat{i}=\lfloor i\frac{p}{n}\rfloor$ and $\hat{j}=j+l \; mod(i,\frac{n}{p})$. Clearly, in this case, the product can be computed in $O(\frac{n^2}{p} + \frac{p}{n})$ steps. Taking all the above facts together, we get that the product of a matrix by a vector can be computed in $O(\frac{p}{n} + \frac{n^2}{p})$ steps on our linear model.

The algorithm for a $\sqrt{p} \times \sqrt{p}$ array follows from that of the linear array. The difference being that the product can be computed in only $O(\frac{n^2}{p} + \sqrt{\frac{p}{n}})$ steps instead of $O(\frac{n^2}{p} + \frac{p}{n})$ steps, if $p > n$.

It is obvious that these bounds are optimal even if RAM's are used.

## References

[1] B.A. Bowen and W.R. Brown, *VLSI Systems Design For Digital Signal Processing,* Prentice-Hall, 1982.

[2] G. Bongiovanni and F. Luccio, "Permutation of Data Blocks in a Bubble Memory," *CACM,* vol. 22, no. 1, pp. 21-24, 1979.

[3] G. Bongiovanni and C. K. Wong, "Tree Search in Major/Minor Loop Magnetic Bubble Memories," *IEEE Transactions on Computers,* vol. C-30, no. 8, Aug. 1981, pp. 537-545.

[4] A. Chandra and C. K. Wong, "The Movement and Permutation of Columns in Magnetic Bubble Lattice Files," *IEEE Transactions on Computers,* vol. C-27, no. 1, Jan. 1979, pp. 8-15.

[5] Kin-Man Chung, F. Luccio, and C.K. Wong, "On the Complexity of Sorting in Magnetic Bubble Memory Systems," IEEE-TC C-29(7), pp. 553-562, July 1980.

[6] J. Ja'Ja' and V.K. Prasanna Kumar, "Information transfer in distributed computing with applications to VLSI," JACM 31(2), Jan. 1984, pp. 150-162.

[7] J. Ja'Ja' and R.M. Owens, "New VLSI Architectures with Reduced Hardware," Third Caltech Conference on Very Large Scale Integration, pp. 351-377, Pasadena, Ca., March 1983.

[8] J. Ja'Ja' and R. Owens," An Architecture for a VLSI FFT Processor," INTEGRATION, The VLSI Journal,1(4), Dec. 1983, pp. 305-316.

[9] G. Miranker, L. Tang and C. K. Wong, "A 'Zero-Time' VLSI Sorter," *IBM J. Res. Dev.,* vol. 27, no. 2, March 1983.

[10] R.M. Owens and J. Ja'Ja'," Sorting on Mesh-Connected Parallel Processors That have Serial Memories," *IEEE Transactions on Computers,* vol. C-34, no. 4, April 1985, pp. 379-383.

[11] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Trans on Computers, C-27(1), Jan. 1979, pp. 2-7.

[12] N. Takagi and C. K. Wong, "A Hardware Sort-Merge System," *IBM J. Res. Dev.,* vol. 29, no. 1, Jan. 1985.

[13] C. Thompson, "Area-Time complexity for VLSI," Proc. 11th annual ACM Symp. on Theory of Computing, Atlanta, Ga., 1979, pp. 81-88.

[14] C. K. Wong and D. Coppersmith, "The Generation of Permutations in Magnetic Bubble Memories," *IEEE Transactions on Comuters,* vol. C-25, 1976, pp. 245-262.

[15] C. K. Wong, "Computer Studies in Mass Storage Systems," Computer Science Press, 1983.

# AN EFFICIENT EMBEDDING OF LARGE TREES IN PROCESSOR GRIDS*

*Duane A. Bailey*
*Janice E. Cuny*

Computer and Information Sciences
University of Massachusetts,
Amherst, Massachusetts 01003

## Abstract

Programming for parallel architectures often includes the specification of a *process embedding* in which logical processes and their interconnections are mapped onto physical processor elements and their connecting links, respectively. Few parallel programming environments provide assistance in performing this embedding. We are investigating the use of grammar-based descriptions of embeddings and we report here on a specific grammar that we have developed for generating embeddings of arbitrarily large, complete binary trees in square processor arrays. The embedding has an efficient processor utilization and it is easily automated. Further, if the number of logical processes exceeds the number of physical processors, our embedding has a simple contraction which yields an optimal, near-uniform distribution of computation and communication tasks.

## Introduction

Within the next decade, parallel architectures composed of a thousand or more processing elements will be commercially available. As the number of processors grows, however, their programming will become an increasingly cumbersome task. This is especially apparent in the activity of *program embedding* in which logical processes and their interconnections are mapped onto physical processing arrays. Aspects of this mapping problem have received significant amounts of attention, yet current parallel programming environments have been able to provide only limited support. We are investigating the use of grammar-based graph descriptions in the development of a strategy for program embedding, and we report here on the characteristics of a specific embedding that we have found for mapping arbitrarily deep, complete binary trees in square grid processor arrays.

Previous embeddings of binary trees into rectangular arrays have been of two types: hand embeddings and recursive embeddings. Hand embeddings[8] have optimal processor utilization but they will not be feasible for the huge numbers of processing elements that we anticipate in future machines. Recursive embeddings[4]-[5] can be automated but they have so far failed to achieve the efficiency of hand embeddings. The Hyper-H embedding, for example, is a useful method for laying out the tree for VLSI but its processor utilization is not optimal. Our strategy, based on specialized *shape grammars*[6], automatically generates optimal embeddings for arbitrarily large trees – even in the case where the number of logical nodes exceeds the number of available processors.

In the next section, we describe the embedding of a $2n$-level complete binary tree in a $2^n \times 2^n$ processor arrays with an interconnection network of the CHiP computer[8]. In following section, we describe the mapping of such a tree onto a smaller grid. And finally, in the last section, we draw some conclusions on extending this program embedding strategy to other regular structures.

## The Tree Embedding

In this section we describe a technique for performing tree embeddings for the CHiP architecture[8]. A CHiP machine (Figure 1) consists of a $2^n \times 2^n$ array of processors with local memory. *Corridors* of switches are located horizontally and vertically between rows of processor elements. We require that each switch have degree 8 and a crossover level of 3, allowing it to host three simultaneous communication paths.

Our tree embedding technique uses a two-phase shape grammar[1] to determine the node assignments and switch settings. In the first phase, processes are embedded into the processor array. For this we use a type of parallel shape grammar, called a *template grammar*, in which derivations are constrained to rewrite the entire sentential shape at each step; that is, the left-hand sides of the set of productions applied must be a minimal set covering the sentential form. In the second phase, communication channels are routed. For this we use a type of sequential shape grammar, called a *ruler grammar*, in which scaling transformations are not allowed.

The template grammar for the first phase of our tree embedding is shown in Figure 2. The letters that appear on the shapes are not actually part of the shapes – they are simply coordinate labels; non-terminal shapes contain an arrow, while terminal shapes do not. A sample derivation is shown in Figure 3. Because the entire shape must be rewritten in every step, all successful derivations in this grammar have the same form: first odd numbered productions are applied for an arbitrary number of steps and then even numbered productions are applied for a single step. The very first production divides the processor array into quadrants. The root of the tree will be located in the quadrant labeled $P$ and it's left and right children will be located in the quadrants labeled $L$ and $R$ respectively. The single unassigned, "orphan processor" will be located in the upper right quadrant, labeled $O$. To embed the remainder of the tree, the array is recursively subdivided into quadrants which will host lower level subtrees. In the final step, processor

*Much of this work was supported by the Office of Naval Research under contract N000014-84-K-0647. Duane Bailey was supported under an American Electronics Association Fellowship from ComputerVision.

[1] A shape grammar operates on shapes in much the same way that a conventional grammar operates on strings: the portion of a sentential form that matches the left-hand side of a production is replaced by its right-hand side. Unlike string grammars, however, the productions of a shape grammar may undergo arbitrary Euclidean transformations before matching [6].

nodes and "buds" for their communication channels are laid down, all of the nonterminal shapes are removed, and we are left with a start shape for the second phase of the grammar.

A ruler grammar for the channel embedding phase is given in Figure 4. In this case, the letters are again used for labels but their positioning is significant. The grammar works by "growing" the channels from the buds left by first phase in such a way that no two channels share a common wire. Figure 5 shows several steps in a sample derivation. Derivations which erroneously create channels that are not part of our embedding will fail to rewrite all non-terminal markers[1]. Final embeddings produced by our grammar for a 63 node tree in an $8 \times 8$ grid and a 255 node tree in a $16 \times 16$ grid are shown in Figure 1. Note the symmetry of the embeddings and that the $8 \times 8$ embedding, rotated by $180^o$, is found in the upper right quadrant of the $16 \times 16$ embedding.

## Contractions Induced by Tree Embeddings

When the number of logical processes exceeds the number of physical processors, it is necessary to contract the logical structure to fit the array. The contraction must map sets of processes to processors while preserving their connectivity. Fishburn and Finkel[3] have suggested using *quotient maps* to generate equivalence classes of processes which may be multitasked on a processor. If the cardinality of each equivalence set is the same, the contraction is said to be *computationally uniform*. A quotient map also induces a set of equivalence classes on the logical communication channels and, if every physical channel emulates the same number of logical channels, it is said to be *exchange uniform*. If a contraction is exchange and computationally uniform, it is said to be *totally uniform*.

Berman and Snyder[2] have suggested the quotient map shown in Figure 6 in which the depth of the tree has been reduced by one, the left and right subtrees of the root have been identified, and the root has been grouped with its two children. Further contractions would be accomplished by iterating this procedure. Repeated coallessing of the root, however, makes this contraction unattractive: it is not computationally uniform because the equivalence class of the root must always contain nearly twice the number of processes found in any other class.

Quotient maps for grids, however, contract more uniformly. Figure 7 details such a contraction. If the grid is folded like a napkin – horizontally and vertically – groups of four nodes are identified to form a smaller grid of equivalence classes. This contraction is totally uniform. It might be expected, therefore, that embeddings of trees that take advantage of grid contractions may yield more uniform distribution of computation and communication. We have found this to be true.

Any embedding of a contracted binary tree in a square grid can off-load extra processes from the root to the unused processor, provided it is possible to route a channel between them. Our embedding assures that the orphan processor is always located horizontally opposite the root, easily accessible to such a channel. We call a complete binary tree in which an extra process has been attached to the root in this way an *augmented tree*.

Extending our grammar to augmented trees in the obvious way produces an embedding with quadrants that are identical under vertical and horizontal flips and can thus be folded into a contraction. This contraction identifies nodes and channels in much same way as Berman's quotient map except in its use of the extra processor. It off-loads nearly half the processes that would otherwise be located at the root, making it exchange uniform and as computationally uniform as possible. Our strategy of first mapping the tree onto a large *logical* processor grid, and then contracting it to an augmented tree embedding of the correct size is conceptually distinct from the contract-then-map paradigm and yields a more uniform distribution of the computation and communication.

## Conclusions

Because shape-related information is often important to the embedding process, we have investigated the use of shape grammars for describing layouts of regular process structures. The strategy we have developed first recursively assigns processes to localities and then routes their interconnections. Using it, we have been able to create efficient embeddings of arbitrarily large, complete binary trees in a square grid. We expect that this strategy will be useful in creating layouts for other regular structures and we are now concentrating on the development of tools to automate some aspects of these grammatical descriptions. In addition, we have found that our tree embedding permits contractions induced by quotient maps of square grids, resulting in a uniform distribution of tasks and communication. We expect that this embed-then-contract approach will also be useful in producing uniform maps of other recursively described structures and we are investigating the extent to which it too can be automated.

## REFERENCES

[1] Duane A. Bailey and Janice E. Cuny, "The Use of Shape Grammars in Process Embeddings," COINS Technical Report 86-23, University of Massachusetts, 1986.

[2] Francine Berman and Lawrence Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," Proceedings of the 1984 International Conference on on Parallel Processing, pp. 307-309 (1984).

[3] John P. Fishburn and Raphael A. Finkel, "Quotient Networks," *IEEE Transactions on Computers,* Volume C-31, Number 4, pp. 288-295 (1982).

[4] Haim Mizrahi and Israel Koren, "Evaluating the Cost Effectiveness of Switches in Processor Architectures," Proceedings of the 1985 International Conference on Parallel Processing, pp. 480-487.

[5] C. Mead and M. Rem, "Cost and Performance of VLSI Computing Systems," *IEEE Journal of Solid State Circuits,* Volume SC-14, Number 2 (1979).

[6] George Stiny, *Pictorial and Formal Aspects of Shape and Shape Grammars,* Birkhaüser Verlag, Basel und Studtgart, 1975, 414 Pages.

[7] Lawrence Snyder, "Parallel programming and the Poker Programming Environment," *Computer,* 17(7), pp. 27-37 (1984).

[8] Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer,* 15(1), pp. 47-57 (1982).

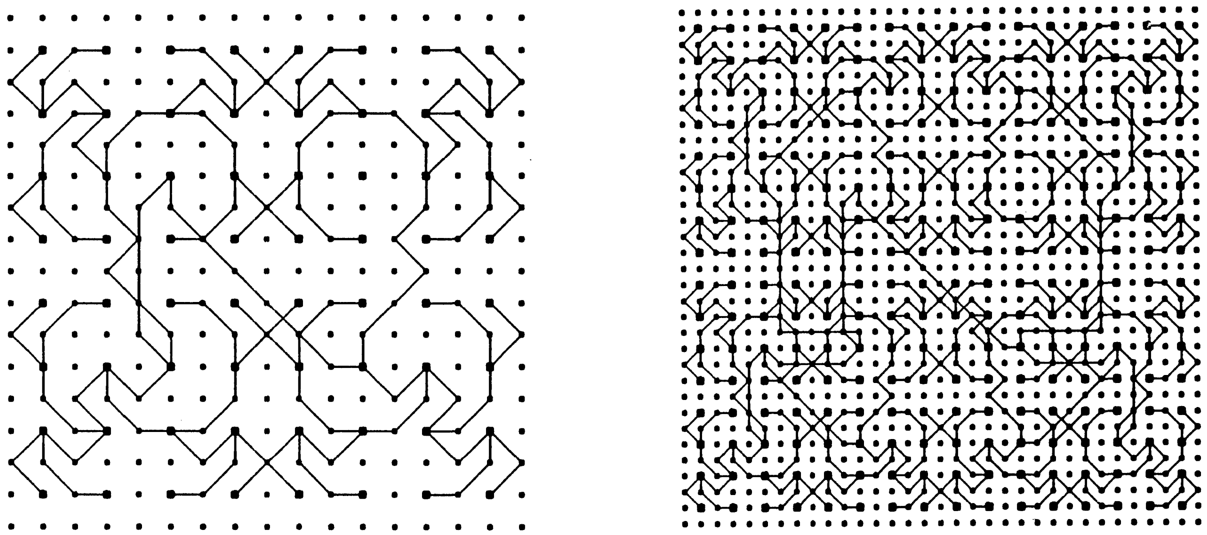


Figure 1: Tree embeddings in the CHiP processor array. Large squares are processors, and small squares are switches.

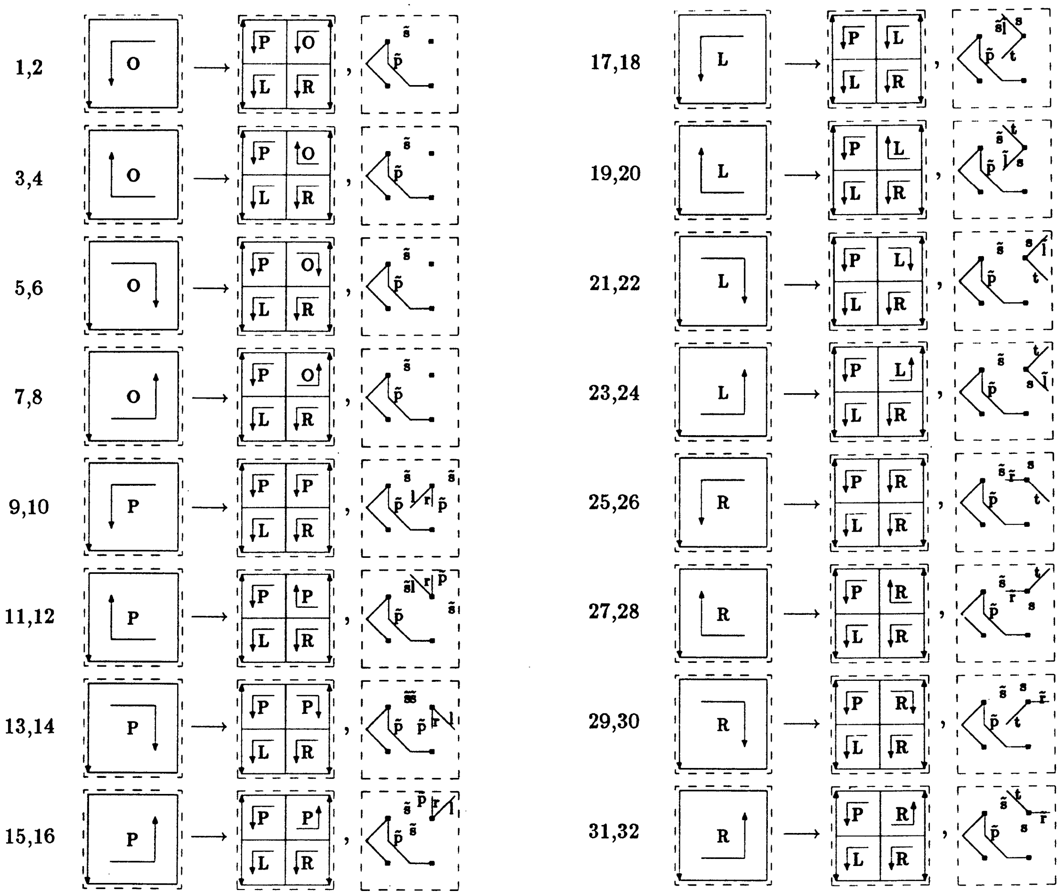


Figure 2: The process embedding grammar. Note that the start shape (left side of production 1) may be scaled before the derivation begins.

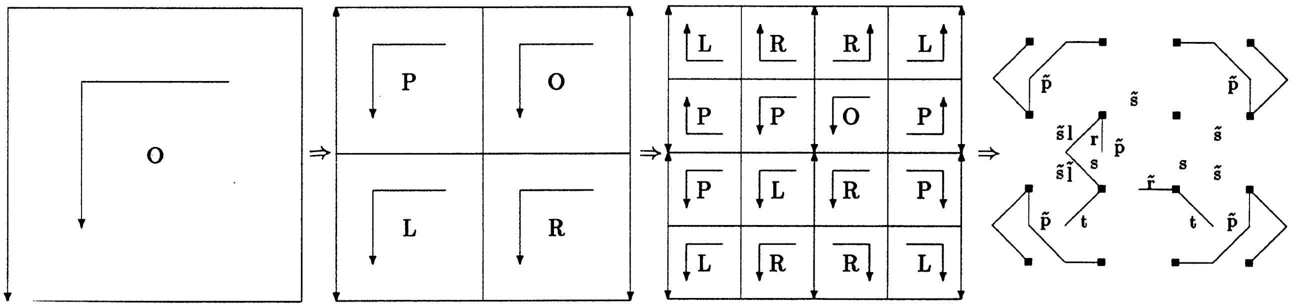


Figure 3: Derivation of a terminal shape by the process embedding phase. The result is a start shape for the channel embedding phase.

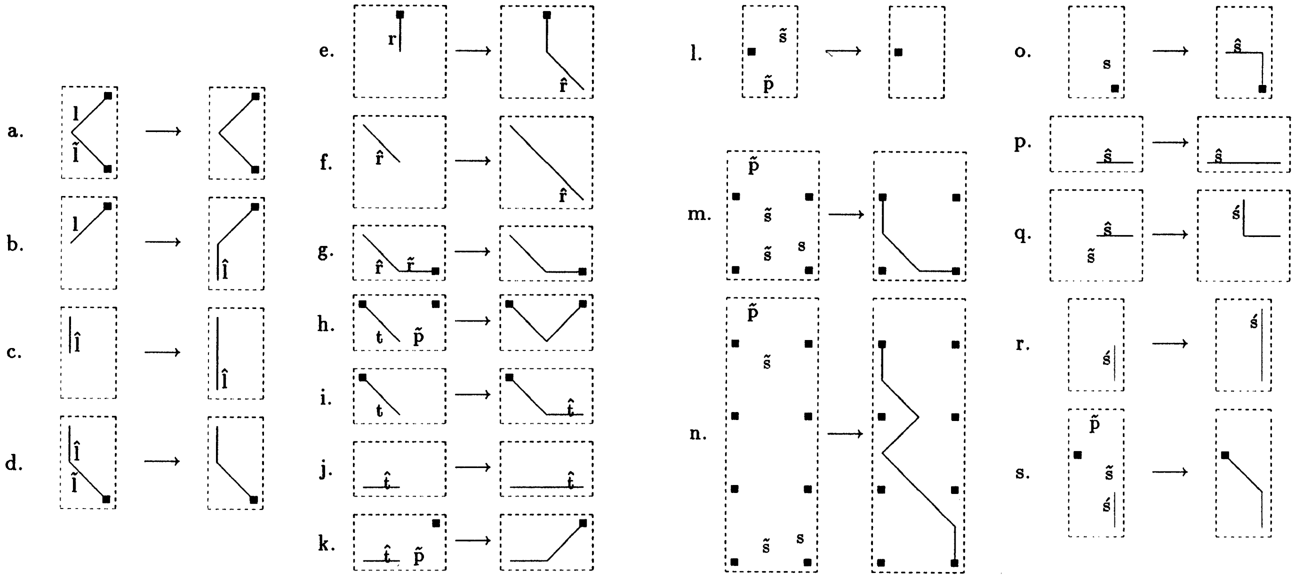


Figure 4: The channel routing grammar. Any shape accepted by the first grammar is a valid start shape for this sequential shape grammar. Productions in the leftmost column, for example, draw connections from the root to its left child.

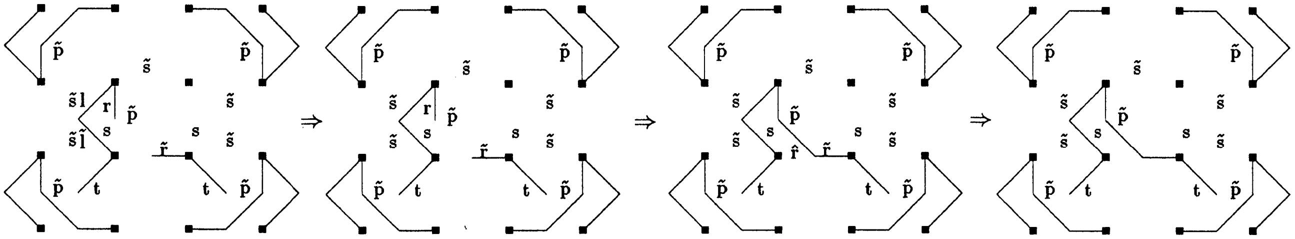


Figure 5: Routing of channels from a left and right son to their parent. The start shape is the final shape of Figure 3.

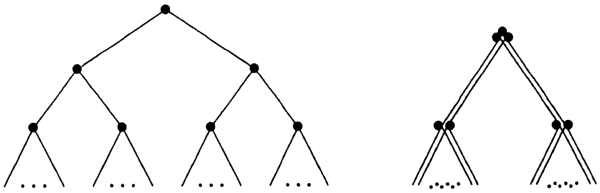


Figure 6: A tree contraction. The left and right subtrees are folded together to form a shorter tree.

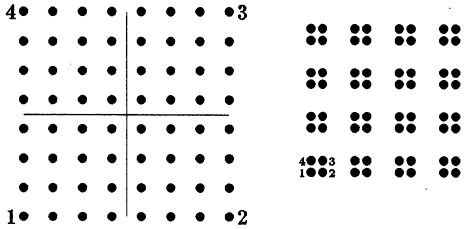


Figure 7: Quotient map for square grids. The array is folded like a napkin; all corners are mapped to the same node.

# OPTIMAL COMMUNICATION NEIGHBORHOODS

(Preliminary Version)

**L.V. Kalé**

Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Ave, Urbana, IL–61801

**Abstract** — An important consideration while designing a large-scale parallel processing system is whether it provides adequate bandwidth to sustain its communication requirements. We show that for a major class of topologies, if global communication is allowed, the communication requirements increase more rapidly than the bandwidth with increasing number of PEs. We explore the strategy of restricting the communication neighborhood of PEs. We give a method for maximizing the neighborhood while satisfying the bandwidth constraints for the point-to-point topologies and bus topologies, and demonstrate the advantages of bus topologies.

## 1 Introduction

Massive parallel processing systems, with tens of thousands of processors, are feasible with current and foreseeable hardware technologies and costs. An important concern while designing such systems is their *scalability*. A system is scalable if one can add an arbitrary amount of hardware and get a corresponding increase in performance.

In this paper, we first prove that any multi-processor interconnection scheme that allows communication between arbitrary pairs of PEs cannot be scalable. A possible solution is to restrict the maximum communication distance of each PE. For many reasons, including the desirability of uniform load distribution, we wish to maximize the number of PEs reachable in that distance from an individual PE. The problem is that of maximizing this *neighborhood* without sacrificing scalability.

For point–to–point topologies, this problem can be solved by choosing the largest communication distance that is consistent with the bandwidth requirements. Bus–based topologies are more interesting in this context because they provide an additional degree of freedom for the designer: namely, the number of PEs connected by each bus.

In the next section, we show why global–communication schemes are not scalable. Section 3 enunciates the strategy of restricted communication neighborhoods. In Section 4, we apply this strategy to point–to–point topologies. Section 5 develops a method for determining the optimal communication neigborhoods for bus topologies, and applies it to some specific topologies. Section 6 summarizes the results.

## 2 Global Communication is Not Scalable

**Theorem:** *Any topology that connects a number of PEs with uniformly distributed communication requirements without using additional switching elements is not arbitrarily scalable.*

'Uniformly distributed communication requirements' means that both the source and destination of the messages are found with equal probability from any of the PEs of the system. The intuitive idea behind the proof is simple: as the number of PEs in a network increase, so does the number of communication channels, thus creating a balance between message data generated per unit time and the total communication bandwidth. But each message now has to traverse more channels than before, and so the communication requirement eventually catches up with the available bandwidth.

Let $B_a$ and $B_r$ be the available and the required bandwidth of a network as whole, respectively. The available bandwidth can be written as: $B_a = channels(p) B$ (1)
where $B$ is the bandwidth of an individual communication channel and $channels(p)$ is the number of communication channels available in a topology with $p$ PEs. Each PE must have connection to a bounded number of channels to be scalable. Let that bound be $c$. Assuming each channel connects $w$ PEs ($w \geq 2$), $channels(p) = pc/w$ (2)
Then, $B_a = Bpc/w$ (3)

The required bandwidth depends on two factors, the message data generated by each PE, say $m$ bytes/sec, and the average number of channels that each message has to travel, say hops(p). $B_r = m\ p\ hops(p)$ (4)

Given that the number of connections per PE is bounded, the average number of hops must increase with p. To ensure that the system always provides the required bandwidth, we must have $B_a \geq B_r$. However:
$B_a \geq B_r \ \rightarrow\ Bpc/w \geq mp\ hops(p) \ \rightarrow\ Bc/(mw) \geq hops(p)$
The left hand side is constant, while the right hand side increases with $p$, and so there exists a $p$ beyond which this requirement can not be satisfied. □

The preceding theorem has to be understood in proper perspective. It does not preclude global communication altogether: The constant factors may require an unreasonably large number of PEs before the communication stagnation occurs. Communication strategies may bias the probability distribution of the PEs involved in a message towards shorter paths so that even though the length of the longest path increases with $p$, the average path–length does not. Decomposition strategies may increase the grain–size along with the number of PEs, thus reducing the overall communication need.

The result applies to a variety of topologies including the shuffle–exchange, the cube–connected cycles (CCC), and the grid of PEs. For example, in CCC, each PE is connected to 3 channels, and so the number of channels is $(3/2)p$. So if we have designed a CCC system with 384 PEs ($2^6.6$) with well–matched processor speed and communication bandwidth, then the theorem guarantees that for the next possible CCC, which has 896 PEs, communication stagnation is bound to occur. The boolean hypercube escapes this fate because the number of channels in it increases as $p \log p$ while the $hops(p)$ is at most $\log p$. However, it achieves this by increasing the number of connections per PE: there are $\log p$ connections/PE. This is inconsistent with our scalability requirement.

The result does not apply to systems with switching networks because the channels within the switching network provide the required bandwidth. For example, the omega network has $p \log p$ communication channels. In this paper we confine ourselves to topologies without additional switching elements.

0190-3918/86/0000/0823 $01.00 © 1986 IEEE

## 3 Restricted Communication Neighborhoods

The theorem of last section points to a fundamental limitation of global communication. A system designer has two approaches for dealing with its consequences.

1. Allow global communication and accept the concommitant limit on the maximum size of the system. This is the only possible approach when the application critically depends on global communication. That is a reason why Ullman [1] suggests this approach for fast sorting on multi-processors.

2. In many application domains, global communication is not *necessary*. So, it is feasible to limit the communication neighborhood of each PE and still retain the scalability of the system. Such domains include parallel execution of functional and logic programs. Here, the sub-computations can be contracted out to *any* PE, and in particular to a PE within the neighborhood. Considerations such as uniform load-distribution dictate that the neighborhoods be as large as possible. Finding the largest neighborhood while satisfying the bandwidth constraints is the topic of the rest of the paper.

## 4 The point-to-point Topologies

Let us consider the point-to-point topologies to begin our analysis. For these, $w=2$, i.e. each channel connects only 2 PEs. Let there be $p$ processors, each generating $m$ bytes/sec. Let $d$ be the (average) distance each message is allowed to go. Then, the required bandwidth of the whole system, $B_r = pmd$. By eqn. (3), the total available bandwidth is: $B_a = (pc/2)B$, where B is the bandwidth of an individual channel. For maximum communication distance $d_{\max}$, we equate the required bandwidth with the available bandwidth, to get: $pmd_{\max}=(pc/2)B$. So, $d_{\max}=(cB/2m)$.

Notice that $d_{\max}$ is the bound on the *average* communication distance. It is possible to design communication strategies that allow different communication distances, including those larger than $d_{\max}$, with varying probability so that the average is $d_{\max}$. Here we consider a simpler, more conservative strategy: restrict each message to travel *at most* $d_{\max}$ hops.

Let the number of PEs reachable in $d$ hops be $nbrhood_r(d)$ for a given topology $r$. Then the maximum communication neighborhood for $r$ is $nbrhood_r(d_{\max})$. As an example, consider the 2-dimensional grid of PEs. Here, $nbrhood_r(d)=2d^2+2d+1$. Let the bandwidth of each channel be 100K bytes/sec. and each PE generate a communication load of 10K bytes/sec. Each PE is connected to 4 channels ($c=4$), so $d_{\max} = (4\times100)/(2\times10) = 20$ and the maximum communication neighborhood includes $2\times20^2+2\times20+1 = 841$ PEs.

## 5 Bus Topologies

A *bus* is a communication channel that connects 2 or more PEs. A bus topology is an interconnection scheme that uses a number of buses. A few bus-topologies have been reported in the literature [2-5]. The number of PEs on each bus is called the *width* of the bus, and denoted by $w$. We assume that all the buses in any specific instance of a scheme have the same width. An interconnection scheme may organize a given number of PEs using a variety of bus-widths. Thus $w$ is available as a design parameter that can be chosen with some flexibility. This flexibility makes the problem of finding the optimal communication neighborhoods interesting.

We use the notation defined in Section 4. In addition, let $b$ be the total number of buses in the system. Equation (2) can then be re-written as:

$$pc = bw \qquad (5)$$

Intuitively, the total number of 'wires' coming out of the PEs must equal the total number of connections available on the buses. As before: The required bandwidth, $B_r=pmd$

The available bandwidth, $B_a=bB=(pc/w)B$ (by eqn. 5).

Equating $B_r$ and $B_a$, $pmd = (pc/w)B$.

Therefore:

$$wd = cB/m=w_0 \qquad (6)$$

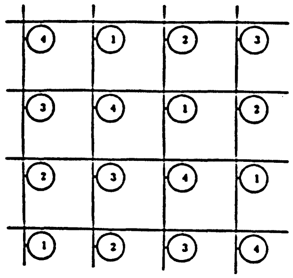
For a given application domain, $m$ is fixed, and for a given communication technology, $B$ is fixed. $c$ is constant for any interconnection scheme. So the equation says that the product $wd$ must remain constant. We denote this constant by $w_0$.

Let $nbrhood_r(w,d)$ be the number of PEs reachable from some PE in $d$ hops in an instance of the topology $r$ with the bus-width of $w$. Obviously, $nbrhood_r(w,d)$ increases with either of $w$ or $d$. Equation 6 tells us that we cannot hope to increase both simultaneously: their product must be kept constant ($=w_0$). This sets up an interesting trade-off between the two. The optimization problem thus reduces to maximization of $nbrhood_r(w,d)$ subject to the constraint that $wd = w_0$. Alternatively, it can be expressed as the maximization of $nbrhood_r(w,w_0/w)$, or of $nbrhood_r(w_0/d,d)$.

The approach for finding the optimal design parameters for a given topology is therefore to first find its *nbrhood* function, and then to maximize the special instance of it expressed above. If we cannot find the a closed-form expression for $nbrhood_r(w,d)$, we may calculate and plot $nbrhood(w_0/d,d)$ empirically, and look for the maxima.

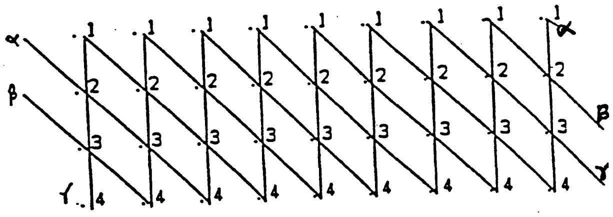
### 5.1 A simple topology

We first consider a simple topology to illustrate the method. An extension of the square grid of buses ( Figure 1) leads to the topology of Figure 2. If $w$ is the width of buses, then in 1 step we reach $2w-1$ PEs. For $d>1$ it can be seen that $nbrhood(w,d) = (d-1)w^2 + w^2/8$, assuming the source is



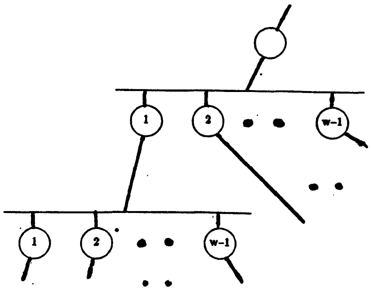
Square Grid of Buses

Figure 1



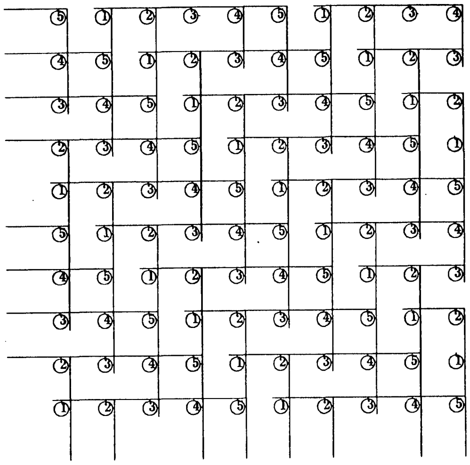
A linear extension of the Square Grid of Buses

Figure 2



Tree of Buses

Figure 3



A Lattice–mesh

Figure 4

at the center of a bus. The optimal w occurs at the maxima of $nbrhood(w,w_0/w) = (w_0/w-1)w^2+w^2/8$. By simple analysis, the optimal $w$ is $4w_0/7$. This corresponds to $d=w_0/w=7/4$. Approximating to the nearest integer, we conclude that the best way to arrange this topology is to let each bus span $w_0/2$ PEs and allow communication to a distance of 2.

### 5.2 Tree of buses

An instance of the tree–of–buses topology is shown in Figure 3. Notice that any node can be made to look like the root of two identical trees (ignoring the discontinuity introduced by the root and the leaves of the physical tree). So, the *nbrhood* function can be written as: $nbrhood(w,d) = 2(w_1^{(d+1)}-1)/(w_1-1)-1$ , where $w_1 = w-1$. For pedagogical value, we approximate this to: $2w_1^d - 1$. The optimization problem then reduces to maximizing: $w_1^{w_0/w}$ . The maxima occurs when $w_1+1 = w_1.ln(w_1)$. This leads to $w_1=3.59$, i.e. $w=4.59$ So we conclude that the width of buses should be 5. The communication distance can then be calculated as $w_0/5$.

### 5.3 The lattice–mesh

The *lattice-mesh* [4] can be thought of as a two–dimensional extension of the square grid of buses. The PEs are laid out on the lattice points in a rectangular $X_{max} \times Y_{max}$ matrix, where $X_{max}$ and $Y_{max}$ are both multiples of w. The layout of buses can be best explained by associating a label with PE. The label of a PE at (x,y) is (1 + (x+y) mod w). All the buses parallel to the X-axis start at a PE with label equal to one, and all the buses parallel to the Y-axis start at a PE with a label $Y_s$, where $Y_s$ is any label other than 1. Figure 4. shows a lattice–mesh of 100 PEs with a bus-width of 5. Buses extending beyond one end are connected consistently at the other end, forming a torroidal structure.

In $d$ hops, the maximum distance one can traverse along either axis is $wd/2$, because any path is a sequence of alternating X and Y axis movements. Moreover, it can be shown that the coordinates of the reachable PEs, assuming (0,0) to be the coordinates of the starting PE, are bounded by: $|X|+|Y| \leq wd/2+w-2$. (Intuitively, every pair of hops can traverse only a distance of $w$, except at the boundaries). Assuming $d \gg 1$, we ignore the smaller terms to get $|X|+|Y| \leq wd/2$. So the reachable PEs are confined within a diamond with distance from the center to a corner equal to $wd/2=w_0/2$. Therefore, nbrhood(w,d) is bounded from above by $2(w_0/2)^2 = w_0^2/2$, and so is independent of w and d! (Even if $w$ and $d$ vary, their product must remain constant by equation 6). I.e. beyond a certain point, increasing $d$ does not lead to any increase in $nbrhood(w,d)$.

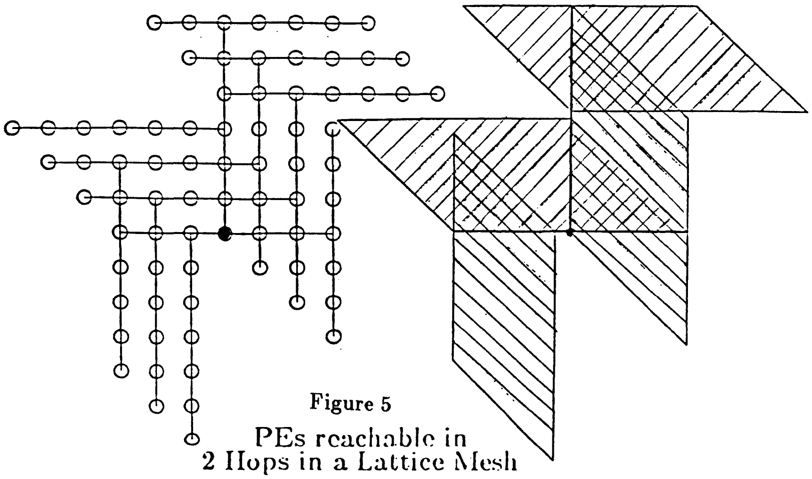


Figure 5

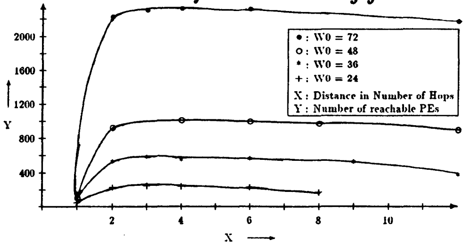
PEs reachable in 2 Hops in a Lattice Mesh

A general calculation of nbrhood(w,d) for lattice–mesh seems difficult. Instead, we derive the number of PEs reachable in a distance of 2. Figure 5–a shows the PEs reachable in 2 hops from the darkened PE, in a lattice–mesh with $w=7$. Figure 5–b depicts the reachable PEs schematically. The doubly shaded areas represent PEs that can be reached by two alternate paths. The reachable PEs are those within the shaded area. By elementary geometry,[1] $nbrhood(w,2) = 4(w^2)/2 - 3(1/2)(w/2)^2 = (13/8)w^2$. For two hops, $w=w_0/2$ (by Equation 6), and we get $nbrhood(w,2)=(13/32)w_0^2$. I.e. in two hops alone, we can reach about 80% of the upperbound on the number of reachable PEs!

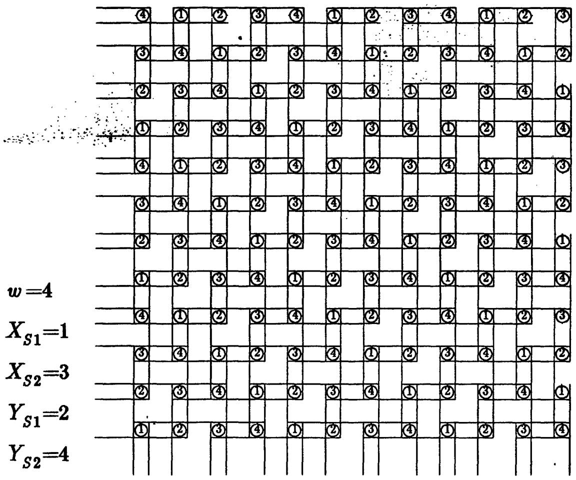
We therefore did some computations to find the maxima of $nbrhood(w_0/d,d)$ for specific values of $w_0$. The results are depicted in Figure 6. In all the instances examined, the maximum occurs at d = 3, and the difference between nbrhood($w_0$/2,2) and nbrhood($w_0$/3,3) was very small.

Notice that when the communication distance is increased from 1 to 2, for $w_0 = 24$, the number of reachable PEs increased from 47 to 223, even though the width of the bus dropped from 24 to 12. Thus at a very little (two-fold) increase in the delay we get a large increase in the neighborhood, without increasing the total communication requirements. Because of the low value of d, very few (1 if d=2) PEs incur the overhead of relaying the message. Therefore, the conclusion seems clear: ***limit communication to a distance of two or three and choose the width of buses accordingly.***

Figure 6



Maximum Reachable PEs in Lattice–mesh



A Double Lattice Mesh

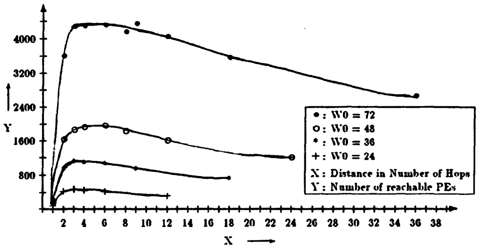
Figure 7

### 5.4 A Bus–topology better than the grid

Is there a bus–topology that is 'better' than the grid–of–processors, in the sense of having a larger neighborhood with similar hardware constraint (i.e. cost) and communication technology? To answer this question, we construct a bus topology that is a generalization of the grid, i.e. one of which the grid is an instance with $w$=2. Such a topology is shown in Figure 7. It has 4 connections per PE like the grid. It is called a double lattice–mesh (DLM) because it can be thought as two lattice–meshes overlapped on each other. A PE at ($x$,$y$) has the label 1+($x$+$y$) *mod* $w$. Analogous to the lattice–mesh, the DLM is paramterized by 4 numbers $X_{s1}$,$X_{s2}$,$Y_{s1}$,$Y_{s2}$ in addition to $w$. There are 2 sets of buses in each row (and column). Each set is arranged just like in the lattice–mesh, except that buses in one set start from PEs with label $X_{s1}$, while buses in the other set start from PEs with label $X_{s2}$. The buses parallel to the Y–axis are similarly defined. It can be easily seen that with w=2, this topology reduces to the grid of PEs.

Again, for the lack of a simple *nbrhood* function, we performed some computations to plot *nbrhood*($w_0$/$d$,$d$). Figure 8 shows the plots for different values of $w_0$. The rightmost point on each plot corresponds to the point–to–point grid. It can be verified from the plots that the bus topologies (w > 2) always do better than the grid. Moreover, $d$=3 is either the optimal distance or is very close to it.

[1] More accurate calculations for nbrhood(w,2) show that when w is odd, *nbrhood*($w$,2)=(13.$w^2$–4.$w$–1)/8, and when w is even, *nbrhood*($w$,$d$)=(13.$w^2$–6.$w$)/8.



Maximum Reachable PEs in Double Lattice–mesh

Figure 8

### 6 Conclusion

We demonstrated unscalability of global communication for a major class of inter–connection schemes, namely those with no additional switching elements. For many application domains, this limitation can be overcome by limiting the communication horizon of each processor. For the point–to–point topologies this strategy led to a simple limit on the communication radius. For the bus topologies, the problem of finding optimal communication neighborhoods is more interesting because of an additional degree of freedom for design: the width of the bus. We gave a general method of finding such neighborhoods, and applied it to several bus–topologies. Such calculations for a topology that generalizes the grid of processors were used to show that bus topologies can perform better in this respect than the point–to–point topologies.

We are investigating bus topologies in the context of a system for parallel processing of combinatorially explosive symbolic computations expressed as Logic Programs. In absense of global communication, the question of distribution of work becomes more important. Contribution of different topologies towards this needs to be investigated. Also, strategies such as (1) probabilistic distribution of communication distance instead of the absolute bound as suggested in this paper and (2) dynamic variation of this distance in response to variations in communication load need to be investigated. Many simulation studies are planned to that end.

### 7 References

1. J. D. Ullman, "Flux, Sorting and Supercomputer Organization for AI Applications", *Journal of Parallel and Distributed Computing*, **1**, (1984), 133–151.

2. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers", *IEEE Transactions on Computers*, **C–30**, 4 (April 1981), 264–273.

3. M. D. Mickunas, "Using Projective Geometry to Design Bus Connection Networks", *Proc. of Workshop on Interconnection Networks for Parallel and distributed Processing*, Apr. 1980, 47–55.

4. L. V. Kale, "Lattice–Mesh: a Multi–Bus Topology", Proc. of ICPP, St. Charles, Illinois, August 1985.

5. L. V. Kale, "Parallel Architectures for Problem Solving", Doctoral Thesis, Dept. of Computer Science, SUNY, Stony Brook, NY–11794..

# The KAP/205:
# An Advanced Source-to-Source Vectorizer
# for the
# Cyber 205 Supercomputer

Christopher Huson, Thomas Macke, James Davies
Michael Wolfe, Bruce Leasure

Kuck and Associates, Inc.
1808 Woodfield Drive
Savoy, IL 61874
217/356-2288

**Abstract.** The KAP/205 is a Fortran source-to-source vectorizer which translates serial Fortran DO loops into explicit vector syntax for the Cyber 205, a supercomputer with memory-to-memory vector instructions. The translated program may be compiled using the Fortran-200 compiler and executed on the Cyber 205. This paper explains the optimizations of the KAP/205 which are peculiar to the Cyber 205 architecture and gives some speedups attained relative to the Fortran 200 compiler alone.

## 1. Introduction

The Control Data Cyber 205 supercomputer [1] has memory-to-memory vector instructions. Because of the penalty for starting a vector instruction and the requirement that vector operands be gathered when they are not contiguous, the Cyber 205 performs best when the program uses many long contiguous vector operands. The Cyber 205 is programmed predominantly in Fortran, using the Fortran-200 compiler [2]. Fortran-200, a Fortran-77 [3] based language, includes extensions such as explicit vector syntax and in-line machine code. These extensions allow users to get good performance from their programs, but require much recoding and make the programs entirely non-portable. Users of the Cyber 205 have long desired a better automatic vectorizer that would allow them to access the power of the Cyber 205 without recoding their programs. The KAP/205 fills this need.

The KAP/205 is a Fortran precompiler which accepts and regenerates Fortran-200 programs. The KAP/205 examines the DO loops in the source program and generates vector code where possible. The output of the KAP/205 is an equivalent Fortran-200 program with the vectorized DO loops replaced by explicit vector syntax and in-line machine code. The advantage of using the KAP/205 is that programs remain portable, and the users need not concern themselves with the peculiarities of the machine.

The second section of the paper describes important features of the Cyber 205 with respect to optimization. The third section describes the KAP/205 and how it maps programs onto the Cyber 205. The fourth section describes the vector optimizer of the KAP/205, which chooses the best method to execute a loop nest on the Cyber 205. The final section presents some speedup results of various programs executed on the Cyber 205.

## 2. The Cyber 205 Supercomputer

The Cyber 205 is a high-speed, pipelined, memory-to-memory vector supercomputer. The basic clock is 20 nanoseconds, and the Cyber 205 can issue one instruction per clock in the best case. The machine is divided into a scalar unit and a vector unit sharing a common main memory, as shown in Figure 1. The floating point units are pipelined (even in the scalar unit), so floating point results can be generated at a high rate. In addition to simple vector arithmetic operations such as add and multiply, the Cyber 205 has vector reduction instructions, such as sum of a vector, product of a vector, and maximum and minimum of a vector. Vectors may have up to 65,535 elements. The vector unit can have one, two or four pipelines, which means that one, two or four 64-bit floating point results can be generated in each 20-nanosecond clock period during a vector instruction. Each vector pipeline has a multiplier and an adder, so linked-triad operations like A(I) = B(I)+C(I)×D, with an add, a multiply, and two vector inputs, can be executed with a single pass through the vector pipeline. In this case, each vector pipeline executes two floating point operations per clock (add and multiply); with four pipelines, the Cyber 205 can execute eight floating point operations every 20-nanosecond clock for a peak rate of 400 million floating point operations per second (400 megaflops). In addition, the Cyber 205 has HALF PRECISION operations; for 32-bit operands, each vector pipeline splits in half and can execute two adds and two multiplies in each clock period. Thus the maximum rate of the Cyber 205 is 800 megaflops using 32 bit operands.

Each vector instruction incurs a certain amount of startup time. Scalar instructions must be executed to set up the vector operand descriptors, but scalar instructions are very fast. Each vector instruction itself needs many clocks (as many as 50 clocks, or more than a microsecond, for some instructions) before the vector unit actually starts producing results. Some of this time is related to the pipeline depth of the vector arithmetic units, but much of it is required to start the vector operands streaming from the main memory.

### 2.1 Gathers and Scatters

All the vector arithmetic instructions require that the vector operands be stored in contiguous memory. Non-contiguous operands

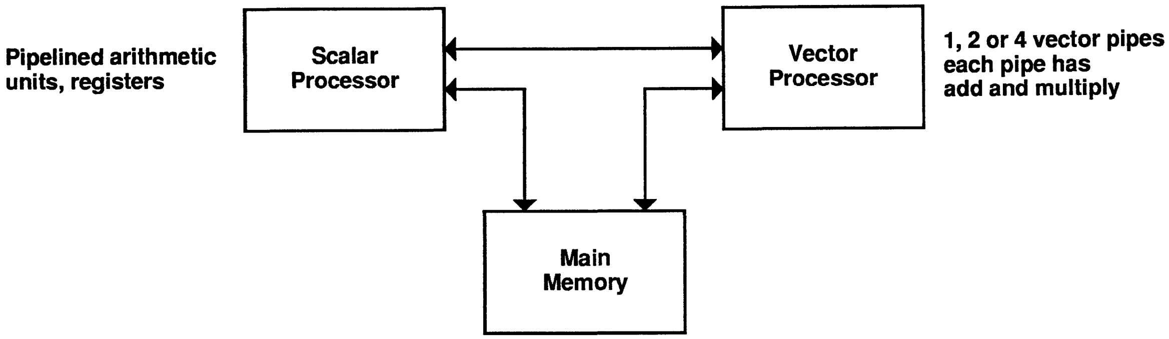

Pipelined arithmetic units, registers
Scalar Processor
Vector Processor
1, 2 or 4 vector pipes each pipe has add and multiply
Main Memory

Figure 1. Architecture of the Cyber 205 Supercomputer.

0190-3918/86/0000/0827 $01.00 © 1986 IEEE

must be gathered into temporary vectors before use, and noncontiguous results must be scattered after generation. Periodic gathers and scatters are used when the stride is fixed, and indexed gathers and scatters are used for nonlinear indexing. For example, in Figure 2(a), the array C would be gathered before being added to the array B and stored in array A. Two vector instructions would be used to execute this loop: an indexed gather (using the index vector IP) and a vector add. In Figure 2(b), the first column of the matrix D would be gathered before being added. Again, two vector instructions would be used: a periodic gather (with a period of N, the size of a row of the matrix D), and a vector add.

Gathers and scatters on the Cyber 205 are slow compared to arithmetic operations. Since the speed of a gather or scatter does not increase with the number of vector pipelines, a gather or scatter operation is even slower relative to vector arithmetic on Cybers with multiple vector pipelines.

```
      REAL A(N),B(N),C(N)
      INTEGER IP(N)

      DO 100 I = 1,N
100   A(I) = B(I) + C(IP(I))
```

(a) Indexed Gather.

```
      REAL A(N),B(N),D(N,N)

      DO 200 I = 1,N
200   A(I) = B(I) + D(1,I)
```

(b) Periodic Gather.

Figure 2. Gathering operands.

### 2.2 Vector IF hardware

A vector compare on the Cyber 205 generates a bit vector, with 1's where the test is true and 0's elsewhere. There are several different methods to use bit vectors to execute conditional vector operations [4]. The choice of the best method depends on various run-time parameters, such as the vector length, the density of the vector (percent of 1's in the bit vector), and the number of distinct vector operands in the code under control of the IF test.

The vector arithmetic operations of the Cyber 205 have an optional control vector field, so that the result is stored only where the bit vector is 1 (or only where it is 0, for the ELSE part of an IF). Where the bit vector is 0, the result vector is left unchanged. Use of control vectors is a natural method to execute simple vector IF statements.

When the density of the bit vector is small, other code sequences may be more efficient. An alternate method is to compress each operand vector into a temporary vector before performing any arithmetic, and decompress the result vectors afterwards. This shortens the vector length of the arithmetic operands. Obviously if the number of compresses and decompresses is larger than the number of arithmetic operations, then the control vector method will be faster, since compress and decompress operations run at the same speed as vector arithmetic. However, if the number of arithmetic operations is relatively large compared to the number of compress operations and the bit vector is sparse, then compressing the operands can pay off with short vectors throughout the IF block.

A third method to handle IF statements is applicable when the bit vector is very sparse, and the number of compress operations is relatively small. In this method an index vector containing the index values of the true elements of the bit vector is built. The compress (and decompress) operations above are replaced with indexed gather (and scatter) operations using this index vector. Even though the indexed gather instruction generates fewer results per clock period, this method can be faster since the vector length of the indexed gather operation is N×d, while the vector length of the compress operation is N (d is the density of the bit vector, i.e. the fraction of 1's in the bit vector; 0≤d≤1, so N×d≤N). So, while the compress instruction may produce 2 results per clock (on a 2-pipe Cyber 205), thus taking N/2 clocks (after startup), the gather instruction produces 0.8 results per clock (on average), thus taking 1.25×N×d clocks (after startup). If d is less than 0.4, this method has a chance of being faster. The additional cost of generating the index vector must be taken into account also.

An example of a DO loop containing an IF statement is given in Figure 3(a). Using control vectors for this loop using control vectors gives code similar to that in Figure 3(b); the vector code is shown here using Fortran 200 primitives. The first vector instruction generates the bit vector into a compiler temporary vector, CV. The vector comparison has a vector length of N. The next four vector instructions use the bit vector to control which elements of the result vector to store. Notice the linked-triad statement, performing a vector add and a vector multiply together. Each of these vector arithmetic instructions has a vector length of N. Assuming a 2-pipe Cyber 205, this vectorized loop would take 5×N/2 or 2.5×N clock periods, plus the time to start the four vector instructions, plus time for any scalar code.

Figure 3(c) uses compress/decompress code. The first vector instruction is the same as before, generating a bit vector into a compiler temporary vector. The next two vector instructions compress the vector operands into two more compiler temporary vectors. The vector compress instruction uses the bit vector to control the compress operation, and simultaneously computes the vector length of the result, shown as ND in the program. Each of the four vector arithmetic instructions has a vector length of N×d, where d is the density of the bit-vector. The last instruction decompresses the short result vector back into the original user array, again with a vector length of N. The three compress/decompress instructions would take 3×N/2 or 1.5×N clock periods (plus three startups). The four arithmetic operations would take 4×N×d/2 or 2×N×d clock periods (plus four startups).

```
      REAL A(N),B(N)

      DO 100 I = 1,N
       IF(A(I).GT.0) THEN
        B(I) = (A(I) * A(I) + B(I) * B(I)) * 0.5 + X
       ENDIF
  100 CONTINUE
```

(a) Original Program.

```
 vector   vector
 length   code
   N      CV(1;N) = A(1;N) .GT. 0
   N      WHERE(CV(1;N)) TA(1;N) = A(1;N) * A(1;N)
   N      WHERE(CV(1;N)) TB(1;N) = B(1;N) * B(1;N)
   N      WHERE(CV(1;N)) B(1;N) = (TA(1;N) + TB(1;N)) * 0.5
   N      WHERE(CV(1;N)) B(1;N) = B(1;N) + X

   total time = 5*N/#pipes + 5 startups
```

(b) Control Vectors.

```
 vector   vector
 length   code
   N      CV(1;N) = A(1;N) .GT. 0
   N      TA(1;ND) = Q8VCMPRS( A(1;N), CV(1;N) ; TA(1;ND))
   N      TB(1;ND) = Q8VCMPRS( B(1;N), CV(1;N) ; TB(1;ND))
   N*d    TA(1;ND) = TA(1;ND) * TA(1;ND)
   N*d    TB(1;ND) = TB(1;ND) * TB(1;ND)
   N*d    TB(1;ND) = (TA(1;ND) + TB(1;ND)) * 0.5
   N*d    TB(1;ND) = TB(1;ND) + 10.
   N      B(1;N) = Q8VDECMP( TB(1;ND), CV(1;N) ; B(1;N))

   total time = 4*N/#pipes + 4*N*d/#pipes + 8 startups
```

(c) Compress/Decompress

```
 vector   vector
 length   code
   N      CV(1;N) = A(1;N) .GT. 0
   N      IV(1;N) = Q8VINTL(1,1;IV(1;N))
   N      IV(1;ND) = Q8VCMPRS( IV(1;N), CV(1;N) ; IV(1;ND))
   N*d    TA(1;ND) = Q8VGATHR( A(1;ND), IV(1;ND) ; TA(1;ND))
   N*d    TB(1;ND) = Q8VGATHR( B(1;ND), IV(1;ND) ; TB(1;ND))
   N*d    TA(1;ND) = TA(1;ND) * TA(1;ND)
   N*d    TB(1;ND) = TB(1;ND) * TB(1;ND)
   N*d    TB(1;ND) = (TA(1;ND) + TB(1;ND)) * 0.5
   N*d    TB(1;ND) = TB(1;ND) + 10.
   N*d    B(1;ND) = Q8VSCATR( TB(1;ND), IV(1;ND) ; B(1;ND))

   total time =
      3*N/#pipes + 3*N*d/0.8 + 4*N*d/#pipes + 10 startups
```

(d) Indexed Gather/Scatter.

Figure 3. Vector IFs.

Adding in the time to generate the bit vector, this comes out to (2×d+2)×N clock periods + 8 startups. If d is less than 0.25, the compress/decompress method may be faster than the control vector method for this loop; of course, the extra startups must be factored in.

Figure 3(d) uses indexed gather/scatter code to execute the vector IF. Again, the first vector instruction generates the bit vector. The second vector instruction initializes an index vector with the index set (1,2,3,...,N). The third vector instruction compresses this index vector into itself; this creates a short index vector which contains the indices of the true elements of the bit vector. Each of these instructions has a vector length of N. The next two vector instructions gather the vector operands into compiler temporary vectors, analogous to the compress instructions in 3(c), with the difference that the vector length is N×d, not N as in the compress instruction. The four vector arithmetic instructions have vector lengths of N×d, and the last vector instruction scatters the short result vector back into the user array, also with a vector length of N×d. The first three instructions would take 3×N/2 or 1.5×N clock periods (plus three startups). The three gather/scatter instructions would take 3×N×d/0.8 clock periods, plus three startups (remember gathers operate at a rate independent of the number of pipes). The four arithmetic operations would take 2×N×d clock periods, plus four startups, just as in 3(c). This totals to (1.5+5.75×d)×N clock periods, plus ten startups. Thus, if d is less than 0.17, this has a chance of being faster than the control vector method (again, the startups must be taken into account). If d is less than 0.13, this has a chance of being faster than the compress method.

### 2.3 Fortran 200 Language

Programming the Cyber 205 is done predominantly in Fortran 200, a Fortran-77 based language with extensions to allow users to access the vector instruction set of the Cyber 205. Some of the vector extensions are shown in the example programs in Figure 3. The semicolon notation (A(1;N)) describes a starting point (A(1)) and a vector length (N), which is how the Cyber 205 hardware describes vector operands. Certain vector intrinsics (Q8VCMPRS, Q8VGATHR) use Cyber 205 vector instructions. In addition there are "special calls" to predefined subroutine names which map directly onto most Cyber 205 machine instructions.

Fortran 200 implements ROWWISE arrays, which are stored in row-major order (like Pascal) instead of column-major. In Figure 2(b), if the array D were declared ROWWISE, then no gather would be necessary in the vector code.

The Fortran 200 compiler includes a vectorizer which, compared to other automatic vectorizers, is relatively weak [5]. Because of this, users of the Cyber 205 learn and use the Fortran 200 vector extensions when they require better performance from their programs. The extensions are hard to use, and make programs non-portable and hard to maintain.

## 3. The KAP/205 Vectorizer

The KAP/205 was designed to alleviate the difficulty of getting good performance on the Cyber 205. The KAP/205 accepts Fortran 200 programs and vectorizes the DO loops, producing an equivalent program with some or all of the DO loops replaced by vector code. The KAP/205 uses Fortran 200 explicit vector notation, vector intrinsic functions and special Q8 calls (in-line assembler code) for vectorized DO loops. Messages and questions from the KAP/205 translator point out the parts of the program which might be further improved with additional help from the user.

The KAP/205 is one of a family of KAP products. The KAP is designed to be a rehostable and retargetable Fortran translation system for discovery of parallelism. Our first products are all source-to-source vectorizers.

The KAP uses the latest software technology for discovery of parallelism in programs. Much of our work is similar to that done at the Parafrase project at the University of Illinois [6,7,8]. Related techniques are also used in the PFC project at Rice University [9,10]. In Parafrase, as well as in the KAP, parallelism is discovered through the use of a precise data dependence graph which shows where data values are generated and where they are used within a DO loop or DO loop nest. The data dependence graph is processed (with simple graph traversal techniques) to find potential problem areas, which appear as cycles in the graph. Each data dependence cycle is carefully examined to see if it can be broken and the loop executed in parallel, or if part or all of the loop must be executed serially.

The KAP includes powerful algorithms for building a high quality data dependence graph. Exact data dependence tests for the most common types of array references are implemented, as well as sophisticated symbolic tests when variables appear as loop bounds or in subscripts. Several auxiliary transformations improve the data dependence graph, such as recognition of loop induction variables and promotion of scalars into arrays.

### 3.1 Retargeting KAP to the Cyber 205

A general purpose vectorizer does not solve the problem facing the users of the Cyber 205. Not all vector operations are equally efficient on this machine, as explained in Section 2. Some operations to be avoided are gathers and scatters of arrays. Short vector operations can sometimes be slower than performing the same code in scalar mode, due to the large startup time for vector instructions. (Interestingly, a vector divide with a vector length of 2 is faster than two scalar divides, because the vector divide unit is pipelined while the scalar divide unit is not; this is not true of other arithmetic operations.) Also, temporary arrays and vectors must be managed efficiently. Finally, many operations cannot be performed in vector mode on the Cyber, such as DOUBLE PRECISION arithmetic.

Retargeting the KAP to the Cyber 205 is made easier due to the modular design of the KAP. One of the first steps of the KAP is the *candidate finder*. This step examines DO loops to see that the operations within the DO loop are legal candidates for vectorization. For the KAP/205, statements with operations that cannot be executed in vector mode, such as DOUBLE PRECISION operations, are marked as non-vectorizable. READ and WRITE statements are similarly marked. DO loops consisting of all non-vectorizable statements are marked as completely serial, and are not studied further.

Much of the KAP, such as the *induction variable finder*, is machine independent, so retargeting these steps was no work at all. The two largest tasks we faced were customizing the vectorizer to produce and optimize vector code for the Cyber 205, and representing these vector constructs in efficient Fortran 200.

### 3.2 Retargeting the Vectorizer to the Cyber 205

The vectorizer of the KAP looks at different ways to execute each DO loop nest and chooses the best. For instance, the standard matrix multiply program shown in Figure 4(a) can be vectorized at least three different ways. In Figure 4(b), the inner DO loop was vectorized, as might be attempted by a simple-minded vectorizer; this results in a inner product of a column of A with a row of B. For the Cyber 205, however, a periodic gather of the column of A is necessary. In Figure 4(c), the second DO loop was interchanged to be the inner loop and was vectorized; this produces a vector operation in the inner loop. On the Cyber 205, this vector statement is actually a linked-triad (an add and a multiply, with one scalar and two vector operands), and would execute in one pass through the vector pipe. However, now two periodic gathers are required and a scatter of the result is also needed. In Figure 4(d), the outermost DO loop was interchanged and was vectorized; this produces a simple linked-triad vector operation in the inner loop with all stride-one vector operands.

The vectorizer step of the KAP actually attempts these different DO loop orderings. If DO loops are not perfectly nested, DO loop distribution may be used in order to allow more loop interchanging. A Cyber 205-specific *vector optimizer* decides which of the loop orderings should be chosen. Given the loop nest in Figure 4(a), the KAP/205 chooses the ordering in 4(d).

The KAP/205 also attempts to collapse loops. The maximum hardware vector length on the Cyber 205 is 65,535; nested loops often have a combined limit of less than this, and can sometimes be executed as a single long vector. *Loop collapsing* combines two or more loops. Other simple and well-known transformations are *statement reordering* to allow vectorization, and *cycle breaking* to eliminate simple data dependence cycles. Figure 5(a) shows a simple DO loop that can be vectorized only if the two statements are reordered; Figure 5(b) shows the equivalent vector Fortran 200 code. Figure 6(a) shows a DO loop with a cycle in its data dependence graph. Semantic analysis shows that this loop simply broadcasts an invariant value to the whole vector A, and thus can be easily vectorized.

```
          REAL C(L,M), A(M,N), B(L,N)

          DO 100 I = 1,L
          DO 100 J = 1,M
          DO 100 K = 1,N
      100  C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

(a) Standard Matrix Multiply Program.

```
          DO 100 I = 1,L
          DO 100 J = 1,M
           TA(1;N) = Q8VGATHP(A(I,1;N), M, N ; TA(1;N))
      100  C(I,J) = C(I,J) + Q8SDOT( TA(1;N), B(1,J;N))
```

(b)Original inner loop (K loop) vectorized.

```
          DO 100 I = 1,L
          DO 100 K = 1,N
           TC(1;M) = Q8VGATHP(C(I,1;M), L, M;TC(1;M))
           TB(1;M) = Q8VGATHP(B(K,1;M), L, M;TB(1;M))
           TC(1;M) = TC(1;M) + A(I,K)*TB(1;M)
      100  C(I,1;M) = Q8VSCATP(TC(1;M), L, M;C(I,1;M))
```

(c) J loop vectorized.

```
          DO 100 J = 1,M
          DO 100 K = 1,N
      100  C(1,J;L) = C(1,J;L) + A(1,K;L)*B(K,J)
```

(d) I loop vectorized.

Figure 4. Loop Interchanging.

```
          REAL A(N),B(N),C(N)

          DO 100 I = 2,N-1
           A(I) = A(I) + B(I-1)
      100  B(I) = B(I) + 1.
```

(a) Unvectorizable statement order.

```
          B(2;N-2) = B(2;N-2) + 1.
          A(2;N-2) = A(2;N-2) + B(1;N-2)
```

(b) Equivalent vectorizable statement order.

Figure 5. Statement Reordering.

```
          REAL A(N)

          DO 100 I = 1,N
      100  A(I) = A(5)
```

(a)

```
          A(1;N) = A(5)
```

(b)

Figure 6.

### 3.3 Producing Efficient Fortran 200

The hardest task of retargeting the KAP to the Cyber 205 was the production of efficient Fortran 200 code. Since the vector constructs of Fortran 200 are so close to the hardware of the machine (even going so far as to define a vector DESCRIPTOR which maps onto the hardware vector operand descriptor), this code generation was very similar to code generation in a compiler. Our job was easier because we did not have to produce our own run-time system and did not have to compile the whole Fortran language.

However, translating the vector constructs into Fortran 200 restricted the operations we could perform. For instance, Fortran 200 does not allow LOGICAL vectors. In order to vectorize IF statements that test LOGICAL array elements, such as the one in Figure 7(a), an artificial method is needed in order to point a descriptor at the proper LOGICAL array. The KAP/205 generates a call to subroutine KQQASS (Figure 7(b)) which returns a descriptor that is used in the controlling expression. The code for KQQASS is a simple descriptor assignment, with the parameter types declared as INTEGER, since the type parameters are ignored by Fortran.

Another problem was that while most of the Fortran intrinsic functions had a vector form, many of them did not allow for conditional evaluation. To use AMOD under control of a conditional expression, for instance, the arguments need to be compressed and the result decompressed.

The only dynamic memory feature in Fortran 200 is dynamic allocation of descriptors. For this reason, the KAP/205 sometimes needs to generate index arithmetic to modify the pointer and length fields of the temporary descriptors. This makes the output of the KAP/205 for vectorized loops look very much like machine code.

```
          LOGICAL L(N)
          REAL A(N),B(I)

          DO 100 I = 1,N
           IF(L(N))A(I) = B(I)
      100 CONTINUE
```

(a) Original program.

```
          DESCRIPTOR LD
          INTEGER LD

          CALL KQQASS( LD, L(1), N )
          WHERE(LD .EQ. 1) A(1;N) = B(1;N)

          SUBROUTINE KQQASS( D, A, N )
          DESCRIPTOR D
          INTEGER D, A(*), N
          ASSIGN D,A(1;N)
          END
```

(b) Vectorized version.

Figure 7. Logical Vectors.

## 4. Vector Optimization in the KAP/205

As mentioned in section 3.2, the vectorizer of the KAP/205 actually discovers many different possible methods to execute loop nests. The "best" loop ordering is the one that will execute in the shortest time in all cases. Clearly this can be data dependent; take for example the matrix multiply case shown in Figure 4. To a casual observer, the vector code in Figure 4(d) is clearly better than the vector code in either of Figures 4(b) or 4(c); 4(d) has no gathers or scatters, and the vector operation is a linked-triad. It seems that this could hardly be improved upon. This assumes that the loop bounds are relatively close, which is certainly true for square matrices. However, cases do arise where the vector length of 4(d), L, is much smaller than either of the other two loop bounds. We can make a rough estimate of the amount of time to execute each vector code segment. For Figure 4(b), one gather operation (taking one startup time plus 1.25 clocks per item gathered) and one inner product (taking one startup plus N/2 clocks, assuming a two-pipe Cyber 205) are required, resulting in two startups plus 1.75×N clocks to execute the vector code. This must be multiplied by the outer loop bounds to get the real time. Note that this is a very rough estimate; in fact startup times are not equal for all instructions, and the serial loop overhead should be factored in.

A similar study of Figure 4(c) counts two gather and one scatter operation (taking three startups plus 3×1.25 clocks per item, total) and one linked-triad (which takes three startups, one for the LINK, one for the MPY and another for the ADD, plus M/2 clocks). Figure 4(d) requires just the linked-triad code (three startups plus L/2 clocks). These numbers are summarized in Figure 8.

If L, M and N are all the same order of magnitude, then the code in Figure 4(d) will be fastest. When L is relatively small, the time for

4(d) can be dominated by the startup time, and 4(b) can look more attractive. If both L and N are small, but M is large, then even 4(c) can turn out to produce the best performance. When all the loop bounds are small, scalar execution of the original loop may be the best method.

Arnold [11] showed that for the Cyber 205, performance of vector Fortran 200 code could be estimated with reasonable accuracy, but the performance of scalar code is much less predictable. The presence of IF statements in the loop makes the prediction process even less precise, but is adequate for a first approximation.

Obviously this analysis can only be applied when the loop bounds are known. In many cases, the loop bounds are parameters to a subroutine or are read in. Also, the probability that each conditional branch is taken must be known to determine the scalar speed and to choose the best method to vectorize that conditional from among the three methods shown above (qv Section 2.2). When these values are not known, some other process must be used to decide what loop ordering to generate. A simpler process is to count important characteristics, such as number of vector statements, number of gathers and scatters needed, and so on.

A third method to approach this problem is to symbolically estimate the execution time of each loop ordering, as we did in Figure 8. IF statements would be inserted to compare the symbolic time estimates using the actual run-time loop bounds in order to choose which loop ordering to execute. This requires generating code for all possible loop orderings.

The actual method used in the KAP/205 is a simple time estimator based on Arnold's results. The best one loop ordering is chosen based on the time estimates, and the code is generated from that loop ordering.

| Timing | | |
|---|---|---|
| code | startups | vector |
| 4(b) | 2×L×M | L×M×N×1.75 |
| 4(c) | 6×L×N | L×M×N×4.25 |
| 4(d) | 3×M×N | L×M×N×0.50 |

Figure 8.

### 4.1 User Interface

The problems with all these are the time required to do the transformations, to estimate the time, and so on, and the necessity to tell the user somehow what happened to his program. A simple vectorizer, that looks only at innermost loops, is much less powerful than the KAP, but at least is easy to understand. If a loop did not vectorize, it is easy to look at the loop and see why it did not vectorize. The KAP/205 has the additional problem of somehow telling the user that any of these three loops could be vectorized, but the DO I (or whatever) loop was vectorized because it was thought best; or, in some cases, a loop was vectorizable, but it was left scalar because it will be faster that way. This occasionally produces unexpected results (to the user). The user interface from the vector optimizer will have to be tuned as the KAP/205 receives more use.

## 5. Speedup Results Using the KAP/205

In this section, we will look at several speedup results using the KAP/205.

Figure 9(a) shows execution time for three different versions of a matrix multiply program for a range of matrix sizes. The line marked "FTN200" shows the execution time for the program shown in Figure 4(a) without the use of the KAP/205. The Fortran 200 compiler does not vectorize the inner loop, giving only scalar performance. The line marked "simple vectorizer" shows the execution time for the program in Figure 4(b); this is the code that a simple vectorizer which only looks at innermost loops might produce; this code is much better than scalar execution. The line marked "KAP/205" shows the execution time for the program in Figure 4(d), which is what the KAP/205 produces for this program; this shows how important loop interchanging and vector optimization are for vector machines. Figure 9(b) shows the megaflops obtained by each of these versions of the program. Figure 9(c) shows the speedup obtained by the KAP/205 over the Fortran 200 compiler and over the hypothetical simpler vectorizer.

Figure 10(a) shows execution times obtained by executing an EISPACK benchmark program with and without using the KAP/205, for a range of problem sizes. Figure 10(b) shows the speedup obtained by the KAP/205 over the Fortran 200 compiler alone. No hand code modification to the EISPACK subroutines was necessary to obtain these results.

Not all programs get the speedup exhibited by these two examples. However, over a wide range of programs, the KAP/205 obtains an average speedup of 1.7-2.0 over the Fortran 200 compiler alone. Some programs will not speed up at all, while others will get reasonable performance and still others will get fantastic speedup.

### Acknowledgements

We would like to acknowledge Mark Byler for his contributions to the work described in this paper.

### References

[1] Control Data Corporation, *CDC Cyber 200 Model 205 Computer System*, Hardware Reference Manual, Pub. No. 60256020, rev. A, March 1981.

[2] Control Data Corporation, *Fortran 200 Reference Manual*, Pub. No. 60480200, rev. D, March 1984.

[3] American National Standards Institute, *American National Standard Programming Language Fortran*, X3.9-1978, 1978.

[4] Dickinson, A., Optimizing Numerical Weather Forecasting Models for the Cray-1 and Cyber 205 Computers, presented at the International Symposium on Vector Processing Algorithms, Colorado State University, August, 1982.

[5] Arnold, C. N., "Performance Evaluation of Three Automatic Vectorizer Packages", presented at the International Conference for Parallel Processing, August, 1982.

[6] Kuck, D., R. Kuhn, B. Leasure, M. Wolfe, "The Structure of a Advanced Vectorizer for Pipelined Processors", in *Proc. of COMPSAC 80, the 4th Int'l Computer Software and Applications Conf.*, pp. 709-715, Oct., 1980.

[7] Kuck, D., R. Kuhn, D. Padua, B. Leasure, M. Wolfe, "Dependence Graphs and Compiler Optimizations", in *Proc. of 8th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 207-218, Jan., 1981.

[8] Kuck, D., A. Sameh, R. Cytron, A. Veidenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, C. Beckman, J. Davies and C. Kruskal, "The Effects of Program Restructuring, Algorithm Change and Architecture Choice on Program Performance", in *Proc. of 1984 Int'l Conf. on Parallel Processing*, IEEE Cat. No. 840H 2045-3, pp. 129-138, Aug., 1984.

[9] Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form", Rice Technical Report 476-029-4, Rice University, Houston, Oct. 1980.

[10] Allen, J., K. Kennedy, "Automatic Loop Interchange", in *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 233-246, June, 1984.

[11] Arnold, C., "Vector Optimization on the Cyber 205", presented at the International Conference for Parallel Processing, August, 1983.

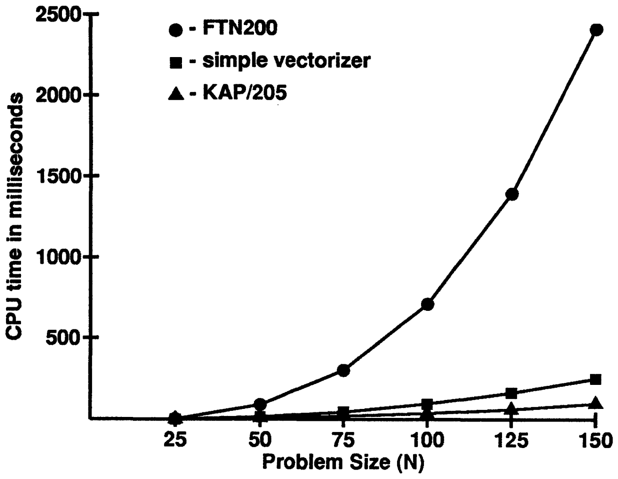


Figure 9(a) Execution times for matrix multiply.

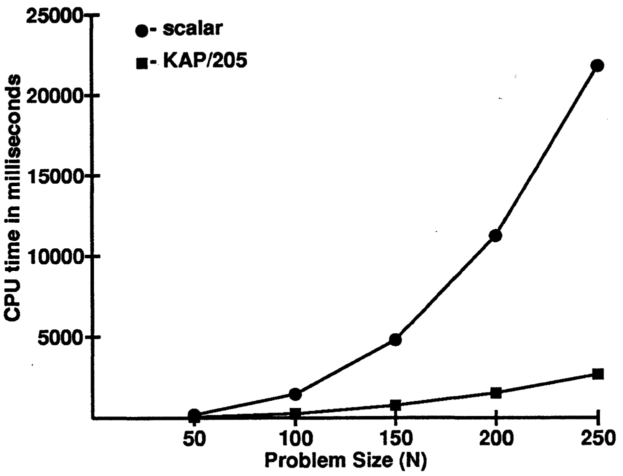


Figure 10(a) Execution times for EISPACK Benchmark.

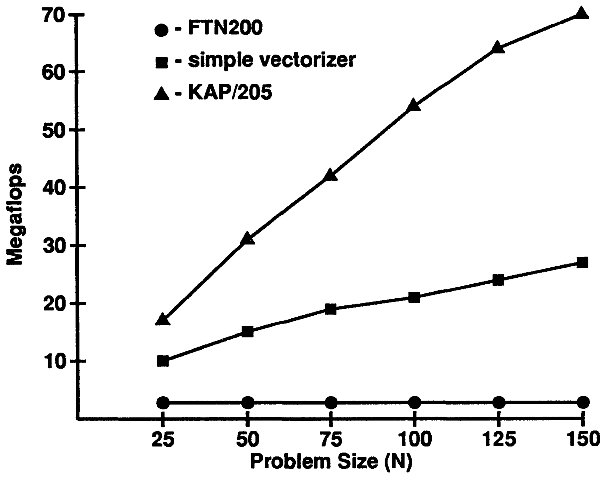


Figure 9(b) Megaflops for matrix multiply.

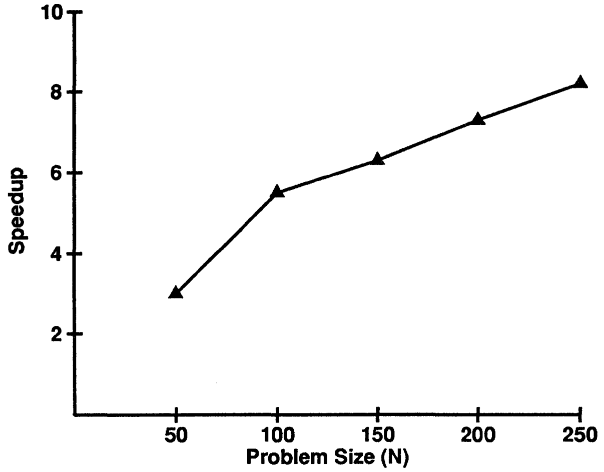


Figure 10(b) Speedup for EISPACK Benchmark.

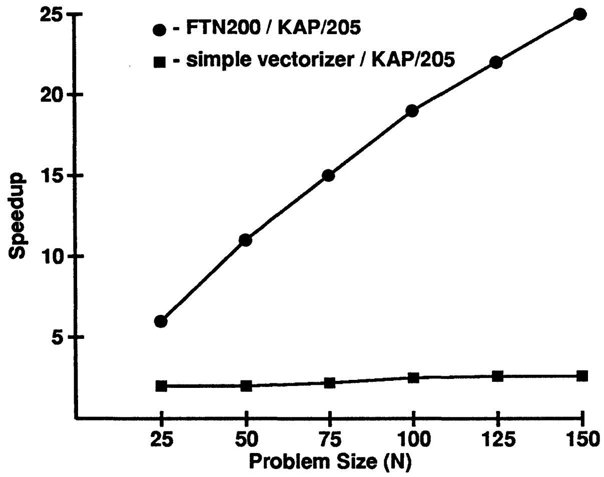


Figure 9(c) Speedups for matrix multiply.

# The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer

James Davies, Christopher Huson, Thomas Macke
Bruce Leasure, Michael Wolfe

Kuck and Associates, Inc.
1808 Woodfield Drive
Savoy, IL 61874
217/356-2288

**Abstract.** The KAP/S-1 is a Fortran source-to-source vectorizer which translates serial Fortran DO loops into explicit vector syntax optimized for execution on the S-1 uniprocessor. This paper explains the optimizations of the KAP/S-1 which are peculiar to the S-1 architecture, including transformations to improve performance of the data cache.

## 1. Introduction

The S-1 Mark IIa supercomputer [1] is a shared memory multiprocessor. Each processor has a memory-to-memory vector instruction set. In order to relieve the load on the shared memory, each processor has a private cache memory. This computer is designed to execute existing and future programs used at the Lawrence Livermore National Laboratory. Most of these programs are now written in LRLTRAN, an extended Fortran language [2].

The KAP/S-1 is a Fortran precompiler which optimizes programs for the S-1 uniprocessor. The KAP/S-1 accepts Fortran-77 [3] with selected LRLTRAN extensions, and produces a modified program using vector syntax for those statements for which the S-1 compiler should generate vector instructions. Several optimizations are also done to improve the performance of the cache memory on the S-1, both for vectorizable and non-vectorizable loops.

The next section of this paper describes the S-1 supercomputer in more detail, concentrating on the features that affect vectorization and optimization in the KAP/S-1. Following that we describe the KAP/S-1 and how it optimizes programs. The fourth section focuses on the cache memory of the S-1 and how the KAP/S-1 attempts to improve the cache hit ratio.

## 2. The S-1 Mark IIa Supercomputer

The S-1 Mark IIa computer includes up to sixteen processors, which communicate through shared memory. Each processor has a private data cache memory and a vector instruction set. In this paper we deal only with optimization for a only with single processor.

The cache memory is 16K 36-bit words organized into 4-way set-associative, 16-word cache lines. Because cache loads are done 16 words at a time, random memory reference patterns will load down the cache-to-main memory link much more on the S-1 than on other machines. This is because each cache miss will result in 16 words being loaded from the main memory, even if only one of these words will be used. However, if memory references are stride-one (meaning contiguous memory addresses are referenced), then the cache memory will work well, since the first cache miss will load the missing word and 15 more words that will be referenced soon. This fits in well with the vector instruction set.

The vector instruction set of the S-1 is relatively limited. Vector adds, multiplies, and other simple operations, as well as MAX, MIN, sum and product reductions are all available. The operands of all of these operations must be stride-one, however. The S-1 has a matrix TRANSPOSE instruction which can be used to transpose arrays which are referenced in the wrong order, or to gather a column from an array into contiguous locations. None of the vector instructions include any method to handle conditionals.

## 3. The KAP/S-1 Vectorizer

The KAP/S-1 vectorizes and optimizes programs for execution on the S-1 uniprocessor. The KAP/S-1 accepts Fortran-77 programs augmented by selected LRLTRAN extensions and produces an equivalent program with some or all of the DO loops replaced by vector code. Vectorized DO loops are represented with vector syntax similar to the syntax proposed by the ANSI X3J3 committee, known as Fortran 8x [4]. The S-1 Fortran compiler accepts this syntax and generates vector code for the S-1. The KAP/S1 vectorizer is related to the KAP/205 and KAP/ST-100 [5,6], and uses many of the same vectorization enhancement techniques.

The vectorization was significantly tuned for the S-1. Because all vector operations on the S-1 require stride-one operands, stride-one array references are preferred. Non-stride-one vector operations are handled by the S-1 compiler, which inserts TRANSPOSE instructions to gather and scatter non-stride-one operands. DO loops can often be interchanged to obtain stride-one array references. For example, the DO J loop in Figure 1(a) could be vectorized, but the resulting vector operations would not have stride-one operands. However, by interchanging the loops before vectorization, the KAP/S-1 will generate the code in Figure 1(b), which has all stride-one array references.

```
      REAL A(100,100),B(100,100),C(100,100)

      DO 100 I = 1,N
       DO 100 J = 1,N
100     A(I,J) = A(I,J) + B(I,J)
```

(a) Non-stride-one.

```
      REAL A(100,100),B(100,100),C(100,100)

      DO 100 J = 1,N
100    A(1:N,J) = A(1:N,J) + B(1:N,J)
```

(b) Interchanged for stride-one.

Figure 1.

Vectorization of the stride-one loop is not always possible; the stride-one loop in Figure 2(a), the DO I loop, cannot be vectorized. In order to obtain vector code, the arrays must be left non-stride-one, as in Figure 2(b). The KAP/S-1 tries different loop orderings and chooses the "best" one, according to its criteria. These criteria include preferring vector code over serial code, and stride-one array operands over non-stride-one array operands. The KAP/S-1 also optimizes the performance of the cache memory, as discussed in the next section.

Interestingly, much of the initial work of customizing the KAP for the S-1 involved restricting the scope of vectorization to that which the S-1 hardware and the S-1 Fortran compiler would handle. For example, the KAP/S-1 precludes vector IFs, since the S-1 has no vector IF hardware. Also, the number of reduction and recurrence operations that the KAP/S-1 would recognize was limited to those for which the S-1 had instructions. Since the vector extensions used only the triplet (colon) notation for vector assignments, diagonal references (such as A(I,I)) could not be vectorized.

```
      REAL A(100,100),B(100,100),C(100,100)

      DO 100 J = 1,N
       DO 100 I = 1,N
100     A(I+1,J) = A(I,J) + B(I,J)
```

(a) Non-stride-one.

```
      REAL A(100,100),B(100,100),C(100,100)

      DO 100 I = 1,N
100     A(I+1,1:N) = A(I,1:N) + B(I,1:N)
```

(b) Vectorized non-stride-one.

Figure 2.

## 4. Optimizing the Cache Memory

Good performance of the cache memory is critical for good performance of the S-1 supercomputer. As was already mentioned, the KAP/S-1 interchanges loops to get stride-one array operands for vector code. For scalar code, the KAP/S1 interchanges loops to get stride-one array references to improve cache performance. Two other transformations were implemented to improve the cache memory performance. These were adapted from work by Abu-Sufah and others [7,8,9] which applies program transformations to improve virtual memory performance. The applications of these transformations to cache memories is clear.

### 4.1 Name Partitioning

The first transformation is *name partitioning*. Name partitioning splits a DO loop into several loops, each of which refers to a non-intersecting subset of the arrays in the loop. The increases locality of reference by reducing the number of arrays to which each DO loop refers. In Figure 3(a), the statements in the DO loop can be split into two distinct groups: the first and third assignments refer to the set of arrays {A,B}, and the second and fourth assignments refer to the set of arrays {C,D}. Figure 3(b) shows how name partitioning would distribute the DO loop into two loops, each of which refers to a subset of the arrays. The original loop refers to four different arrays during each iteration. Each of the distributed loops refers to only two different arrays, thus improving locality of data reference.

```
      DO 100 I = 1,N
       A(I) = A(I) + B(I) + T
       C(I) = C(I) + D(I) + T
       B(I) = B(I) / 2.
100    D(I) = D(I) / 2.
```

(a) Original loop.

```
      DO 100 I = 1,N
       A(I) = A(I) + B(I) + T
100    B(I) = B(I) / 2.
      DO 101 I = 1,N
       C(I) = C(I) + D(I) + T
101    D(I) = D(I) / 2.
```

(b) Name partitioned.

```
       A(1:N) = A(1:N) + B(1:N) + T
       C(1:N) = C(1:N) + D(1:N) + T
       B(1:N) = B(1:N) / 2.
       D(1:N) = D(1:N) / 2.
```

(c) Original loop vectorized.

```
       A(1:N) = A(1:N) + B(1:N) + T
       B(1:N) = B(1:N) / 2.
       C(1:N) = C(1:N) + D(1:N) + T
       D(1:N) = D(1:N) / 2.
```

(d) Name partitioned and vectorized.

Figure 3.

If the original DO loop were vectorized, as in Figure 3(c), the first vector statement will load arrays A and B into the cache memory, while the second vector statement will load arrays C and D. If the vector length, N, is very large, the cache will fill up when C and D are loaded, thus making A and B non-resident. This means that B will have to be reloaded into the cache when the third vector statement is executed. When the distributed loops are vectorized, as shown in Figure 3(d), the second vector statement refers to the array B, which will already be cache-resident.

### 4.2 Strip Mining

*Strip mining* [10] is a more important transformation for the S-1. Strip mining is usually thought to be important for vector machines with a small maximum vector length, such as vector register machines (like the Cray [11]). Proper strip mining, however, can also improve virtual memory (and cache memory) performance.

The DO loop in Figure 4(a) may be easily vectorized, producing the code in Figure 4(b). However, if N is very large, greater than 8,000, then the first vector statement will overflow the cache memory, and will have to start reusing cache memory positions for both the A and B arrays. When the first statement is done, the cache memory contains A(N-8000:N) and B(N-8000:N). When the second statement is started, the first elements accessed are A(1) and B(1); however, these locations were flushed back to the main memory by the first statement, and thus need to be reloaded. To make matters worse, during the execution of the second statement, each new element of A and B that is loaded will replace some later element of A and B in the cache, which will then need to be reloaded later in the vector statement, until all of the arrays A and B have been from main memory to cache and back twice. For loops which use more arrays, this effect is seen at shorter vector lengths.

The solution is to strip mine the DO loop before vectorizing it, as shown in Figure 4(c). The DO IBLOCK loop steps between strips of the inner loop, and is left serial by the vectorization process. The inner loop can be vectorized, as shown in Figure 4(d). The BLOCKSIZE is chosen so that the cache memory never overflows; a rough estimate is the size of the cache divided by the number of distinct arrays referenced in the loop (assuming stride-one array references); for this loop, BLOCKSIZE=8000 would probably be appropriate. The memory reference patterns of the inner loop are very different from the original loop. Since the vector length is never so large as to overflow

```
      DO 100 I = 1,N
       A(I) = A(I) + B(I) + T
100    B(I) = B(I) / A(I)
```

(a) Original loop.

```
       A(1:N) = A(1:N) + B(1:N) + T
       B(1:N) = B(1:N) / A(1:N)
```

(b) Original loop vectorized.

```
      DO 100 IBLOCK = 1,N,BLOCKSIZE
       ISTART = IBLOCK
       IEND = MAX(N,IBLOCK + BLOCKSIZE - 1)
       DO 100 I = ISTART, IEND
        A(I) = A(I) + B(I) + T
100     B(I) = B(I) / A(I)
```

(c) Strip mined.

```
      DO 100 IBLOCK = 1,N,BLOCKSIZE
       ISTART = IBLOCK
       IEND = MAX(N,IBLOCK + BLOCKSIZE - 1)
       A(ISTART:IEND) = A(ISTART:IEND) +
     X                  B(ISTART:IEND) + T
100    B(ISTART:IEND) = B(ISTART:IEND) /
     X                  A(ISTART:IEND)
```

(d) Strip mined and vectorized.

Figure 4.

the cache memory, the second statement will always find its operands in the cache memory. When the second strip starts, it overwrites the operands from the first strip in the cache memory; since they are not needed anymore (for this loop), they can be flushed with no cost later in the execution. The total amount of traffic between the cache memory and the main memory is reduced by a factor of two for long vectors, with a small overhead cost associated with executing the strip mining code. Notice that this type of strip mining only affects memory traffic when there is more than one reference to some array, since the memory traffic that is reduced is the second and subsequent cache loads. If there is only one reference to each array in the loop, then this type of strip mining will not affect the memory traffic at all.

More dramatic performance improvements can appear in DO loop nests. Consider the doubly-nested DO loop in Figure 5(a). The DO J loop can be vectorized, resulting in the code in Figure 5(b). This code can have problems similar to those which occurred in the loop in Figure 4. If the loop bound, N, is very large, then the first iteration of the serial DO I loop will overflow the cache memory with elements of A and B. On the second iteration of the DO I loop, the initial part of the A array will have to be reloaded from the main memory. In essence, the A array will have to be loaded from the main memory N times, once for each trip around the serial loop. Simple strip mining, shown in Figure 5(c), will not alleviate this problem, since the reference pattern for the A array will not change. However, once the DO J loop is strip mined, the outer DO JBLOCK loop can be independently interchanged with the DO I loop, giving the code in Figure 5(d). Vectorizing the inner loop, as shown in Figure 5(e), will produce a different memory reference pattern. In Figure 5(e), BLOCKSIZE is chosen so that each vector statement will not overflow the cache. That way, a strip of the A array can remain cache resident for the entire execution of the serial DO I loop. Each strip of A will be used N times, and then can be flushed back to the main memory since it is not used again in this loop. The memory traffic due to the A array is thus reduced by a factor of N. Loop interchanging of strip mined loops as shown here has clear applications for vector register machines also [12].

```
    DO 100 I = 1,N
     DO 100 J = 1,N
100   A(J) = A(J) + B(J,I)
```

(a) Original loop.

```
    DO 100 I = 1,N
100  A(1:N) = A(1:N) + B(1:N,I)
```

(b) Original loop vectorized.

```
    DO 100 I = 1,N
     DO 100 JBLOCK = 1,N,BLOCKSIZE
      JSTART = JBLOCK
      JEND = MAX(N,JBLOCK + BLOCKSIZE - 1)
      DO 100 J = JSTART,JEND
100    A(J) = A(J) + B(J,I)
```

(c) Strip mined.

```
    DO 100 JBLOCK = 1,N,BLOCKSIZE
     JSTART = JBLOCK
     JEND = MAX(N,JBLOCK + BLOCKSIZE - 1)
     DO 100 I = 1,N
      DO 100 J = JSTART,JEND
100    A(J) = A(J) + B(J,I)
```

(d) Strip mined and interchanged.

```
    DO 100 JBLOCK = 1,N,BLOCKSIZE
     JSTART = JBLOCK
     JEND = MAX(N,JBLOCK + BLOCKSIZE - 1)
     DO 100 I = 1,N
100    A(JSTART:JEND) = A(JSTART:JEND) +
   X                    B(JSTART:JEND,I)
```

(e) Strip mined, interchanged, and vectorized.

Figure 5.

### 4.3 Vectorization and Cache Performance

One unfortunate side effect of vectorization is a decrease in locality of reference. This happens because a loop which formerly executed several statements for a particular index value will, after vectorization, execute each statement for all the index values before going to the next. Thus, if more than one statement in a loop referenced the same array element, as in Figure 3, in the scalar version that array element would remain in the cache through all of the references. After vectorization, the cache will be filled with all of the operands of the first statement before executing the second statement. If the vector length is long enough that all the operands do not fit in the cache, then cache misses will result when the second statement is executed. This effect is countered by proper use of name partitioning and strip mining, as shown above. In nested loops such as those in Figure 4, vectorization and cache transformations will result in fewer cache misses than the scalar code, rather than more.

## 5. Conclusion

In this paper we have shown an example of vectorization for the S-1, a vector supercomputer with a cache. By using loop interchanging, name partitioning, and strip mining, the KAP/S1 improves the cache hit ratio by a significant factor both for loops that are vectorized and for those which remain scalar.

## References

[1] Wood, L. et al, *The S-1 Project, 1979 Annual Report*, Lawrence Livermore Laboratory, UCID-18619, vols. 1-3, 1979.

[2] Derby, W., J. Engle and J. Martin, *LRLTRAN Language used with the CHAT and CIVIC Compilers*, Lawrence Livermore National Laboratory, LCSD-302, 1981.

[3] *American National Standard Programming Language FORTRAN*, American National Standards Institute, X3.9-1978, Apr., 1978.

[4] *X3J3/S8*, American National Standards Institute, version 98, Jan., 1986.

[5] Huson, C. et al, "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," *Proc. of the 1986 International Conference on Parallel Processing*, Aug., 1986.

[6] Macke, T. et al, "The KAP/ST-100: A Fortran Translator for the ST-100 Attached Processor," *Proc. of the 1986 International Conference on Parallel Processing*, Aug., 1986.

[7] Abu-Sufah, W., "Improving the Performance of Virtual Memory Computers," Rpt. No. UIUCDCS-R-78-945, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Nov., 1978.

[8] Abu-Sufah, W., D. Kuck and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," *Proc. of the 1979 National Computer Conf.*, pp. 969-974, June, 1979.

[9] Abu-Sufah, W., D. Kuck and D. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Trans. on Computers*, Vol. C-30, No. 5, pp. 341-356, May, 1981.

[10] Loveman, D., "Program Improvement by Source-to-Source Transformation," *JACM*, Vol. 24, No. 1, pp. 121-145, Jan., 1977.

[11] *Cray XM-P Series Mainframe Reference Manual*, Cray Research, Inc., HR-0032, 1982.

[12] Allen, J., and K. Kennedy, "Automatic Loop Interchange," *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, pp. 233-246, Jun., 1984.

# Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)

**Ron Cytron**

IBM T. J. Watson Research Center
Yorktown Heights, New York
(work began while the author was at
the University of Illinois at Urbana-Champaign)

Although vector constructs could be executed with speedup on a multiprocessor, the resources of a multiprocessor are wasted unless there are constructs for which it outperforms a vector processor. This paper describes a method of executing iterative loops on a multiprocessor. Loops are classified by their parallelism and a class of loops is exposed that must execute sequentially on vector machines yet can execute with speedup on a multiprocessor.

## 1.0 Motivation and Background

Program performance can be improved through the execution of vector operations, where a single operation is applied concurrently to multiple operands. With the advent of vector processors came the need for vectorization: the compile-time identification of statements within loops that yield vector operations. However, programs do not consist solely of vector operations. Some form of multiprocessing is required to achieve concurrency for constructs of greater complexity than vector loops. However, current methods for programming multiprocessors either rely heavily on user assistance, or involve automatic techniques that are confined to identifying constructs that are even simpler than vector loops. This paper presents a compile-time technique for determining the parallelism of an arbitrary iterative loop, based on the dependences among statements within that loop. In particular, loops that must execute sequentially on a vector processor can execute concurrently on a multiprocessor.

## 1.1 Architectures

Vector processors and multiprocessors both achieve concurrency through multiple processing elements, but the capability and cost of each can be characterized by the following stylized architectures: An *SEA* machine (*S*ingle *E*xecution of *A*rray instructions) drives its processing elements in a parallel or pipelined manner from a control unit that processes a single stream of instructions. In a multiprocessor, each processing element is driven by a stream of instructions issued from its own control unit. One would expect that the multiprocessor, albeit more costly, would provide increased flexibility and performance over the SEA machine.

## 1.2 Dependence

A program can be regarded as a collection of operations that are constrained in their execution by *dependences*: certain operations must complete before other operations can commence. Let a *dependence graph* be constructed for a program, such that directed edges (arcs) connect dependent operations.[1] A dependence exists between two operations, either because they must share or not share some data (data dependence), or because one operation determines whether the other operation is performed (control dependence).

Consider two statements from a program: $S_i$ and $S_j$. If $S_i$ must store any of its output before $S_j$ can fetch all of its input, then a *flow dependence* exists between statements $S_i$ and $S_j$, denoted $S_i \; \delta \; S_j$. An *anti-dependence* exists from $S_i$ to $S_j$, denoted $S_i \; \bar{\delta} \; S_j$, if $S_i$ requires data from an area in which $S_j$ stores its output. Finally, an *output dependence* exists from $S_i$ to $S_j$, denoted $S_i \; \delta^o \; S_j$, if statements $S_i$ and $S_j$ store into common storage locations. Whichever statement executes last in the sequential program must store its output last in the concurrent program.

If statement $S_i$ determines whether statement $S_j$ is executed, then a *control dependence* exists between statement $S_i$ and statement $S_j$, written $S_i \; \delta^c \; S_j$.

## 1.3 Concurrent Execution of Loops

This paper describes the effects of concurrently executing *iterative loops*. An iterative loop contains a sequence of statements (the *loop body*) that is performed for a sequence of *iteration variable* values. The iteration space for an uninterrupted *normalized loop* with upper bound $N$ is the sequence $1,2,...,N$.[2]

A compiler for an SEA architecture must search a program for *vector loops*: loops containing operations that can be executed simultaneously for all iterations. Vectorizing compilers consider the dependence issues mentioned above to determine the vector loops of a program [1, 2, 9, 15]. Statements that participate in a *dependence cycle* cannot be executed as vector loops.

Consider the execution of a vector loop on a multiprocessor, where the iterations of the loop are executed on potentially

[1] In practice, the nodes of a dependence graph are often statements, which results in smaller graphs of coarser granularity.

[2] Any loop can be transformed into a normalized loop through *do-loop normalization* [14].

0190-3918/86/0000/0836 $01.00 © 1986 IEEE

different processors. The absence of a dependence cycle guarantees that the statements of the loop can be ordered, such that all dependences point lexically forward. All dependences then have the form $S_i \delta S_j$, where $i < j$. If processors are driven synchronously, then a compiler can schedule instructions such that dependences are honored without explicit synchronization. Synchronization is necessary if processors operate asynchronously; however, by the time statement $S_j$ executes, statement $S_i$ should have finished. The processor executing statement $S_j$ would probably not have to wait for statement $S_i$ to complete. For more details on synchronization, see [11].

Loops with cycles of dependence must be executed sequentially on an SEA architecture. However, as shown in this paper, such loops can exhibit parallelism when executed on a multiprocessor.

## 2.0 The Doacross Technique

This paper presents a technique that models the execution of sequential loops, vector loops, and loops of intermediate parallelism. The analysis is first presented for a single loop and is subsequently extended for arbitrarily-nested loops. This work is then compared with other methods for executing loops on multiprocessors.

### 2.1 The Doacross Schedule

Consider a single loop $L$ of $s$ statements ($S_1$, $S_2$ , ..., $S_s$ ) and $N$ iterations. Let virtual processors[3] be assigned to loop $L$ such that virtual processor $VP_i$ executes iteration $i$. If loop $L$ is a a vector loop, all $N$ iterations can execute concurrently. Consider the execution of statement $S_1$ of the vector loop. An instance of that statement exists in each virtual processor assigned to the loop. There is effectively no delay between the execution of consecutive instances of $S_1$ , because all virtual processors can start simultaneously. This style of execution corresponds to the total partition algorithm [8] that produces the *doall* schedule [5].

Consider the execution of loop $L$ as a sequential loop. With virtual processors assigned as above, the sequential schedule requires that processor $VP_i$ waits until processor $VP_{i-1}$ finishes its iteration.

Vector and sequential loops can therefore be modeled by a *doacross schedule* with *delay d*: each iteration is assigned to a virtual processor, but each virtual processor delays executing its loop body for the time period $d$, as shown in Figure 1.

```
DO i=1,N
  delay (d*(i-1))
  $S_1$
  $S_2$
   .
   .
   .
  $S_s$
  CONTINUE
```

Figure 1. A Doacross Loop

In general, the delay $d$ between consecutive iterations (virtual processors) can range from no delay (the vector loop case) to the time of the loop body (the sequential loop case). If $T(S_i, S_j)$ is the time for executing statements $S_i$ through $S_j$, inclusively, within an iteration of the loop ($i \leq j$), then a doacross loop has $d = 0$ for a vector loop and $d = T(S_1, S_s)$ for a sequential loop. These loops, and loops of intermediate parallelism, can be characterized by:

$$\textit{parallelism} = \frac{T(S_1, S_s) - d}{T(S_1, S_s)} \times 100 \text{ percent}$$

Sequential loops have 0% parallelism and vector loops have 100% parallelism. Loops of intermediate parallelism are of the greatest interest in this paper, because they achieve no speedup on an SEA architecture.

In a doacross loop of $N$ iterations, the last iteration will accumulate $(N - 1)d$ delay time before executing the loop body. When the loop body of the last iteration has been executed, the execution of the loop is complete:

$$T(loop) = (N - 1)d + T(S_1, S_s) \qquad [1]$$

### 2.2 Generalization of the Doacross Model

In this section, the doacross model is extended to accommodate multiple loops. Figure 2 shows the syntactic model of multidimensional doacross for $L$ nested loops; loop 1 is the outermost loop and loop $L$ is the innermost loop.

```
$B_0$
  $DO_1$ $i_1$=1,$N_1$
    delay (($i_1$-1) * $d_1$)
    $B_1$
    $DO_2$ $i_2$=1,$N_2$
      delay (($i_2$-1) * $d_2$)
      $B_2$
        .
          .
            .
              $B_{L-1}$
              $DO_L$ $i_L$=1,$N_L$
                delay (($i_L$-1) * $d_L$)
                $B_L$
              $ENDDO_L$
              $A_{L-1}$
            .
          .
        .
      $A_2$
    $ENDDO_2$
    $A_1$
  $ENDDO_1$
$A_0$
```

Figure 2. Multidimensional Doacross Model

[3] Virtual processors conveniently model an infinite processor system. Limited processor results appear in [4].

The execution of a doacross loop at nest level $l$ consists of an $N_l$-way fork, where the branch corresponding to the iteration variable $i_l$ delays $(i_l - 1)d_l$ clocks before beginning the execution of block $B_l$. At the end of a doacross loop, the branches join together. For example, a doubly-nested loop structure with no delay at either nest causes all iterations of both nests to execute in parallel.

Consider the timing of a doacross loop at nest level $l$. Iteration $N_l$ delays $(N_l - 1)d_l$ clocks, and then executes block $B_l$, $DO_{l+1}$, and block $A_l$. The time to execute the statements of Figure 2 is therefore:

$$B_0 + \sum_{l=1}^{L} ((N_l - 1)d_l + T(B_l) + T(A_l)) + A_0 \qquad [2]$$

where $T(B_l)$ and $T(A_l)$ represent the time to execute a particular block of code from Figure 2.

## 2.3 Dependence Considerations

This section considers the effects of dependences on delay. Although all dependences in a loop can contribute to a loop's delay, each dependence can be considered separately,[4] with the delay for a loop chosen as the minimum delay that satisfies all dependences.

### 2.3.1 Data Dependence

Consider a loop of $s$ statements as shown in Figure 3. Consider two statements, $S_j$ and $S_i$, where $S_j$ lexically precedes $S_i$ ($j < i$). An arc in the dependence graph of Figure 3 from $S_j$ to $S_i$ means that during the sequential execution of the loop, statement $S_j$ computes data used by $S_i$. Let $I_k$ represent the iteration in which $S_j$ computes the data, and let $I_l$ represent the iteration in which $S_i$ uses the data. The semantics of sequential programs execution guarantee that $I_k \leq I_l$; the iteration in which the data is used cannot occur earlier than the iteration in which the data is computed.

```
DO I=1,N
  delay ((I-1) * ??)
  $S_1$
  .
  .
  $S_j$
  .
  .
  $S_i$
  .
  .
  $S_s$
```

Figure 3. A Loop with Unknown Delay

Consider a virtual processor assignment of the iterations of Figure 3 with no delay, as shown in Figure 4. Each column represents the statements that are executed sequentially within a virtual processor. If processors execute at approximately the same rate, virtual processor $VP_l$ will not reach statement $S_i$ of iteration $I_l$ (a column of Figure 4) before statement $S_j$ in virtual processor $VP_k$ has finished. The dependence, as shown in the schedule of Figure 4, is satisfied by the time it is needed, even in the absence of any delay. Because this dependence points forward in the program source, it is called a *lexically-forward dependence*; such dependences do not contribute to a loop's delay. As mentioned in Section 1.3, some architectures may require synchronization to honor such dependences.

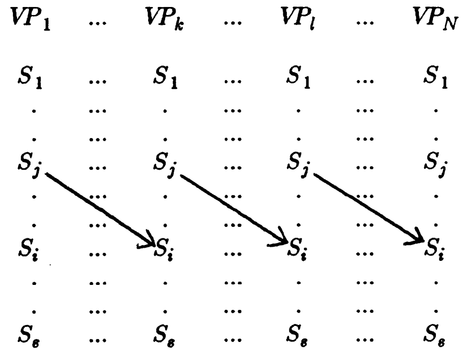
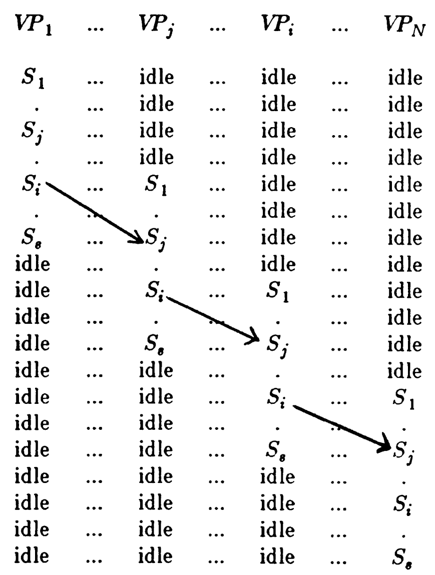

$VP_1$ ... $VP_k$ ... $VP_l$ ... $VP_N$
$S_1$ ... $S_1$ ... $S_1$ ... $S_1$
$S_j$ ... $S_j$ ... $S_j$ ... $S_j$
$S_i$ ... $S_i$ ... $S_i$ ... $S_i$
$S_s$ ... $S_s$ ... $S_s$ ... $S_s$

Figure 4. Loop with Lexically-Forward Dependences

Suppose the dependence were reversed, such that $S_i \; \delta \; S_j$. This type of dependence points backward in the program source, so it is called a *lexically-backward dependence* (abbreviated *LBD*). A lexically-backward dependence can be transformed into a lexically-forward dependence, if $S_i$ and $S_j$ do not participate in a dependence cycle, by reordering the two statements. Those lexically-backward dependences that cannot be eliminated by reordering are the only LBDs of interest in this paper: statements $S_i$ and $S_j$ participate in a dependence cycle, $S_i \; \delta \; S_j$, and $j \leq i$. The semantics of sequential programs guarantee that $I_k < I_l$, since $S_j$ executes no later than $S_i$ within the same iteration. A zero-delay loop schedule will not satisfy the dependence, since virtual processor $VP_l$ will arrive at statement $S_j$ before virtual processor $VP_k$ can satisfy the dependence by executing $S_i$. Delay must be introduced, such that $S_i$ of virtual processor $VP_k$ executes before $S_j$ of virtual processor $VP_l$, as shown in Figure 5.

[4] Multiple dependence relationships can exist between nodes of a dependence graph. The dependence graph is therefore a multigraph, but each dependence can be treated separately.



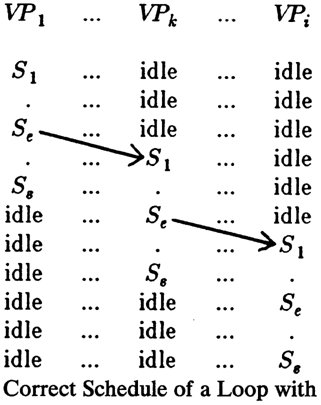
| $VP_1$ | ... | $VP_j$ | ... | $VP_i$ | ... | $VP_N$ |
|---|---|---|---|---|---|---|
| $S_1$ | ... | idle | ... | idle | ... | idle |
| . | ... | idle | ... | idle | ... | idle |
| $S_j$ | ... | idle | ... | idle | ... | idle |
| . | ... | idle | ... | idle | ... | idle |
| $S_i$ | ... | $S_1$ | ... | idle | ... | idle |
| . | ... | . | ... | idle | ... | idle |
| $S_s$ | ... | $S_j$ | ... | idle | ... | idle |
| idle | ... | . | ... | idle | ... | idle |
| idle | ... | $S_i$ | ... | $S_1$ | ... | idle |
| idle | ... | . | ... | . | ... | idle |
| idle | ... | $S_s$ | ... | $S_j$ | ... | idle |
| idle | ... | idle | ... | . | ... | idle |
| idle | ... | idle | ... | $S_i$ | ... | $S_1$ |
| idle | ... | idle | ... | . | ... | . |
| idle | ... | idle | ... | $S_s$ | ... | $S_j$ |
| idle | ... | idle | ... | idle | ... | . |
| idle | ... | idle | ... | idle | ... | $S_i$ |
| idle | ... | idle | ... | idle | ... | . |
| idle | ... | idle | ... | idle | ... | $S_s$ |

Figure 5. Correct Schedule of a Loop with LBDs

### 2.3.2 Control Dependence

Control dependences are caused by conditional and unconditional branching in programs. There are two cases of branching that must be considered: intra-loop branching and loop exits. The result of an intra-loop branch causes either the omission or repetition of a portion of the loop body, which affects the execution time of a loop. Delay is a probabilistic measure, and represents the time interval observed at the start of an iteration that is *expected* to satisfy dependences. The issue of intra-loop branching and the calculation of delay will be resolved by associating probabilities with statement execution times.

In the absence of loop exits, all iterations of a loop are executed. Consider the program with a loop exit at statement $S_e$. During its sequential execution, some iterations of the loop might be skipped. Like a loop with an LBD, delay must be introduced to accommodate the loop exit, as shown in Figure 6.



| $VP_1$ | ... | $VP_k$ | ... | $VP_i$ |
|---|---|---|---|---|
| $S_1$ | ... | idle | ... | idle |
| . | ... | idle | ... | idle |
| $S_e$ | ... | idle | ... | idle |
| . | ... | $S_1$ | ... | idle |
| $S_s$ | ... | . | ... | idle |
| idle | ... | $S_e$ | ... | idle |
| idle | ... | . | ... | $S_1$ |
| idle | ... | $S_s$ | ... | . |
| idle | ... | idle | ... | $S_e$ |
| idle | ... | idle | ... | . |
| idle | ... | idle | ... | $S_s$ |

Figure 6. Correct Schedule of a Loop with a Loop Exit

## 2.4 Calculation of Delay

There are only two constructs that contribute to a loop's delay: loop exits and lexically-backward dependences. The following sections show how to estimate delay for these two cases. After some definitions, a recursive timing formula is presented. Invoking the recursive timing formula once yields the (closed) formula for calculating delay in a single loop. The recursive formula can be invoked again to yield an equation for calculating delay in a double loop structure. The optimal solution to this equation lies in the linear programming domain, but several special cases are discussed. The general formulation for a loop nest of $L$ loops is found by invoking the recursive timing formula $L$ times.

### 2.4.1 Dynamic Statement Instances

Let a program consist of a static sequence of $s$ statements: $S_1$, $S_2$ , ..., $S_s$. Consider statement $S_i$ that is surrounded by $L$ normalized iterative loops. Let $I_1, I_2, ..., I_L$ represent the iteration variables of the $L$ loops, where $I_1$ is the iteration variable of the outermost loop, and $I_L$ is the iteration variable of the innermost loop that surrounds $S_i$. Each iteration variable can assume only integer values in its *iteration sequence*. If the upper bound of iteration variable $I_l$ is $N_l$, then the iteration sequence of that iteration variable is $<1, 2, ..., N_l>$ in the absence of loop exits.

The $L$ loops form an $L$-dimensional integer Cartesian space. The space is composed of discrete points, where each point represents a particular assignment of values to iteration variables from their corresponding iteration ranges [15]. In the absence of loop exits, statement $S_i$ will be executed for all possible values that the iteration variables can assume. The number of times statement $S_i$ executes is therefore $\prod_{l=1}^{L} N_l$.

A *dynamic statement instance*, or simply an *instance*, is the execution of a statement for a specific point in the iteration space of its surrounding loops [8]. The pair $(S_i, \hat{I})$ represents an instance of statement $S_i$ for the *iteration vector* $\hat{I}$. The components of the iteration vector correspond to values of iteration variables for the loops surrounding $S_i$ : $(i_1, i_2, ..., i_L)$. Let $\hat{I}_{|q|}$ represent the iteration vector $\hat{I}$, truncated to its first $q$ elements $(1 \le q \le L)$.

### 2.4.2 The Delta Vector

Consider two statements, $S_i$ and $S_j$, that participate in a lexically-backward data dependence: $S_i \ \delta \ S_j$; some dynamic statement instance of $S_i$ generates data or control information that is used by a dynamic statement instance of $S_j$. Let the instance of $S_i$ be $(S_i, \hat{I})$ and let the instance of $S_j$ be $(S_j, \hat{J})$. Let $CN(S_i, S_j)$ be the number of loops that surround *both* $S_i$ and $S_j$; these are called the *common loops* of $S_i$ and $S_j$. The $\Delta$-*vector* is defined as the difference in iteration index values of the two instances:[5]

$$\Delta = \hat{I}_{|CN(S_i,\, S_j)|} - \hat{J}_{|CN(S_i,\, S_j)|}$$

As developed by Wolfe [15], dependence information is available only for the common loops of $S_i$ and $S_j$. This discussion

[5] A given dependence can give rise to multiple $\Delta$-vectors, but these can be accommodated as distinct edges of the dependence multigraph, each treated separately.

considers only those Δ -vectors that are valid over the common loops of $S_i$ and $S_j$, although the results can be generalized.

As above, let $T(S_j, S_i)$ be the time for executing statements $S_j$ through $S_i$, inclusively, and let $T(S_i)$ be (an abbreviation for) the time for executing only statement $S_i$. Finally, let $\tau(S_i, \hat{I})$ be the time at which $(S_i, \hat{I})$ begins to execute.

### 2.4.3 A Recursive Timing Formula

Consider a statement instance $(S_i, \hat{I})$, where $L = |\hat{I}|$. Statement $S_i$ is surrounded by $L$ loops, with $I_L$ as the innermost loop. The time at which $S_i$ begins to execute with the iteration vector $\hat{I}$ can be defined in terms of the time at which the innermost loop starts and the time at which $S_i$ executes within the innermost loop. In terms of $\tau()$,

$$\begin{aligned}\tau(S_i, \hat{I}) = \tau(DO_L, \hat{I}_{|L-1|}) + (i_L - 1)d_L \\ + T(DO_L, S_i) - T(S_i)\end{aligned} \qquad [3]$$

$\tau(DO_L, \hat{I}_{|L-1|})$ refers to the time at which the innermost do loop $(DO_L)$ begins to execute with the iteration vector $\hat{I}_{|L-1|}$ . This term regards $DO_L$ as a statement within the scope of its outer loop $(DO_{L-1})$. The rest of the expression represents the time at which statement $S_i$ executes, for iteration $i_L$ and delay $d_L$ : $T(DO_L, S_i)$ accounts for the execution of statements from the $DO_L$ through $S_i$, inclusively; $T(S_i)$ is subtracted to make the formula reflect the time at which $S_i$ starts.

The dependence $S_i\delta S_j$ implies that for some iteration vectors, $\hat{I}$ and $\hat{J}$, statement $S_j$ must execute after statement $S_i$ has completed execution. In terms of $\tau()$,

$$\tau(S_j, \hat{J}) \geq \tau(S_i, \hat{I}) + T(S_i)$$

Since $\hat{J} = \hat{I}$ - Δ, the above formula can be rewritten:

$$\tau(S_j, \hat{I} - \Delta) \geq \tau(S_i, \hat{I}) + T(S_i) \qquad [4]$$

After adding $T(S_j, S_i) - T(S_i)$ to both sides of Formula 4:

$$\begin{aligned}\tau(S_j, \hat{I} - \Delta) + T(S_j, S_i) - T(S_i) \geq \\ \tau(S_i, \hat{I}) + T(S_j, S_i)\end{aligned} \qquad [5]$$

By observing that for some *DETLA*-vector $\hat{K}$:

$$\tau(S_i, \hat{K}) = \tau(S_j, \hat{K}) + T(S_j, S_i) - T(S_i)$$

Formula 5 becomes:

$$\tau(S_i, \hat{I} - \Delta) \geq \tau(S_i, \hat{I}) + T(S_j, S_i) \qquad [6]$$

Statement $S_i$ with iteration vector $\hat{I} - \Delta$ cannot start execution until $T(S_j, S_i)$ time has elapsed since the start of statement $S_i$ with iteration vector $\hat{I}$. Since $\hat{J} = \hat{I} - \Delta$, the instance of statement $S_i$ in the iteration in which $S_j$ sinks the dependence for $S_i\delta S_j$ cannot execute until $T(S_j, S_i)$ time after the instance of statement $S_i$ begins to execute in the iteration in which statement $S_i$ sources the dependence.

Using Formula 3, the references to $\tau()$ on the left and right of 6 become:

$$\begin{aligned}\tau(S_i, \hat{I} - \Delta) = \tau(DO_L, (\hat{I} - \Delta)_{|L-1|}) + (i_L - \Delta_L - 1)d_L \\ + T(DO_L, S_i) - T(S_i)\end{aligned}$$

and

$$\tau(S_i, \hat{I}) = \tau(DO_L, (\hat{I})_{|L-1|}) + (i_L - 1)d_L + T(DO_L, S_i) - T(S_i)$$

where $\Delta_L$ is the last (innermost) entry in the Δ-vector. Formula 6 can then be rewritten:

$$\begin{aligned}\tau(DO_L, (\hat{I} - \Delta)_{|L-1|}) - \Delta_L d_L \geq \\ \tau(DO_L, \hat{I}_{|L-1|}) + T(S_j, S_i)\end{aligned} \qquad [7]$$

### 2.4.4 The Single Loop Case

Formula 7 references the delay in the innermost loop, the innermost Δ- vector entry, and the static execution time between statements $S_j$ and $S_i$. Formula 7 also requires the time at which the next outer loop begins to execute with iteration vectors $\hat{I}$ and $\hat{I} - \Delta$ , each truncated before the innermost loop (length $L - 1$). Consider the case of a single iterative loop; because a single loop cannot have an outer loop, the recursive references to $\tau()$ vanish in Formula 7, leaving:

$$-\Delta_L d_L \geq T(S_j, S_i)$$

In terms of linear programming [6, 7], the above formula represents a *constraint* for the delay parameter $d_L$. This constraint should be set to minimize the *objective* function:

$$T(DO_L) = (N - 1)d_L + T(S_1, S_s),$$

which is the time to execute $N$ iterations of a doacross loop containing $s$ statements and delaying $d_L$ time between successive iterations. A single loop can contain a lexically backward dependence only when the iteration in which data is used occurs after the iteration in which data is computed. $\Delta_L$ was constructed from a particular $\hat{I}_L - \hat{J}_L$ for $S_i\delta S_j$ . $\hat{J}_L$ must therefore occur after $\hat{I}_L$ , assuring $\Delta_L < 0$ . The delay $d_L$ is easily solved:

$$d_L \geq \frac{T(S_j, S_i)}{-\Delta_L}.$$

The delay due to the pair of statement instances $(S_i, \hat{I})$ and $(S_j, \hat{J})$ is calculated from the time interval between $S_j$ and $S_i$ and the distance (in iterations) across which the data is passed. Since the above formula provides a lower bound for $d_L$, all delays arising from pairs of statements in a loop can be satisfied by choosing the delay of maximum value for all pairs. An algorithm for determining the delay for a loop in this manner is given in [4].

Consider the example shown in Figure 7. Each assignment statement is linked to the next assignment statement via a forward and backward arc in the data dependence graph. Figure 7 includes a table of estimated execution times for the statements. To estimate $d$ (the delay for the loop), all pairs of statements involved in lexically-backward dependences $(S_i\delta S_j)$ must be considered. For each pair $(S_j, S_i)$, the function *T()* must be evaluated. Note that in this example, the Δ-vector for each backward dependence is (-1). The delay for the loop should be 4 time units per iteration, since the pairs $(S_3, S_2)$ and $(S_4, S_3)$ both yield $T() = 4$. Figure 7 contains the timing schedule for this example.

By delaying each iteration, Figure 7 shows how the $S_3 \delta S_2$ dependence is satisfied. The second processor commences execution of $S_2$ at time step 5; $S_3$ finishes execution in the first processor during time step 4. C(1) will not be requested in $VP_2$ until $VP_1$ has calculated C(1). Note also that the third processor in Figure 7 is really unnecessary; $VP_1$ finished its loop body during time step 7, so $VP_1$ could have executed the third iteration. This is an example of limited processor scheduling, which is discussed in [4].

```
DO i=1,N
  delay((i-1)*d)
  S1:  A(i) = B(i-1) + 37
  S2:  B(i) = A(i) + C(i-1)
  S3:  C(i) = B(i) + D(i-1)
  S4:  D(i) = C(i) + E(i-1)
  S5:  E(i) = D(i) + 77
  ENDDO
```

| Statement | Execution Time |
|---|---|
| $S_1$ | 1 |
| $S_2$ | 2 |
| $S_3$ | 2 |
| $S_4$ | 2 |
| $S_5$ | 1 |

| Time | $VP_1$ | $VP_2$ | $VP_3$ |
|---|---|---|---|
| 0 | $S_1(1)$ | | |
| 1 | $S_2(1)$ | | |
| 2 | | | |
| 3 | $S_3(1)$ | | |
| 4 | | $S_1(2)$ | |
| 5 | $S_4(1)$ | $S_2(2)$ | |
| 6 | | | |
| 7 | $S_5(1)$ | $S_3(2)$ | |
| 8 | | | $S_1(3)$ |
| 9 | | $S_4(2)$ | $S_2(3)$ |
| 10 | | | |
| 11 | | $S_5(2)$ | $S_3(3)$ |
| 12 | | | |
| 13 | | | $S_4(3)$ |
| 14 | | | |
| 15 | | | $S_5(3)$ |

Figure 7. A Single Loop Example

### 2.4.5 The Double Loop Case

Because a single loop has no outer loop, the references to $\tau()$ vanished from Formula 7. A double loop contains an inner loop and an outer loop; if the references to $\tau()$ in Formula 7 are expanded using Formula 3, a new formula results that reflects the situation of two loops: one at nest $L$ and one at nest $L-1$, surrounded by a third loop at nest $L-2$:

$$\tau(DO_{L-1},(\hat{I} - \Delta)_{|L-2|}) + - \Delta_{L-1}d_{L-1} - \Delta_L d_L \geq \qquad [8]$$

$$\tau(DO_{L-1}, \hat{I}_{|L-2|}) + T(S_j, S_i)$$

Since there are only two loops, then the references to $\tau()$ vanish from Formula 8, yielding:

$$-\Delta_{L-1}d_{L-1} - \Delta_L d_L \geq T(S_j, S_i) \qquad [9]$$

Formula 9 imposes conditions on $d_L$ and $d_{L-1}$ such that dependences within two nest levels are satisfied; the optimal setting of the two delays for a single dependence in a two-loop environment cannot be determined from Formula 9. The global objective is to minimize the execution time of the two-loop construct; the execution time for the multidimensional doacross model, restricted to two loops surrounding $s$ statements, is given by:

$$T(DO_{L-1}) = (N_{L-1} - 1)d_{L-1} + (N_L - 1)d_L + T(S_1, S_s) \qquad [10]$$

The minimization of Formula 10 with respect to the constraint of Formula 9 is a linear programming problem, with Formula 10 as the objective function. The objective can be simplified by symbolically replacing constant terms:

$$T(DO_{L-1}) = \eta_1 d_{L-1} + \eta_2 d_L + \eta_3 \qquad [11]$$

where

$$\eta_1 = N_{L-1} - 1$$
$$\eta_2 = N_L - 1$$
$$\eta_3 = T(S_1, S_s)$$

Consider the case in which both $\Delta_{L-1}$ and $\Delta_L$ are negative. The boundary expression for Formula 9 can then be written:

$$d_{L-1} \geq \frac{T(S_j, S_i) + \Delta_L d_L}{-\Delta_{L-1}} \qquad [12]$$

or

$$d_{L-1} \geq \alpha_1 - \alpha_2 d_L$$

where

$$\alpha_1 = -\frac{T(S_j, S_i)}{\Delta_{L-1}}$$

$$\alpha_2 = \frac{\Delta_L}{\Delta_{L-1}}$$

Consider the case where all delay is charged to the outermost loop $d_{L-1}$; no delay is charged to the inner loop, so $d_L = 0$. The objective function becomes:

$$T(DO_{L-1}) = \alpha_1 \eta_1 + \eta_3 \qquad [13]$$

Similarly, all delay can be charged to the inner loop $d_L$; no delay is charged to the outer loop, so $d_{L-1} = 0$. The objective function becomes:

$$T(DO_{L-1}) = \frac{\alpha_1}{\alpha_2}\eta_2 + \eta_3 \qquad [14]$$

A comparison of Formula 13 with Formula 14 decides whether it is best to charge all delay to the outer loop or to the inner loop. If $\eta_1 < \frac{\eta_2}{\alpha_2}$, then all delay should be charged to the outer loop. Consider the assignment of delay when the loop bounds are the same ($\eta_1 = \eta_2$); if $\alpha_2 < 1$, then all delay should be charged to the outer loop. Recall that $\alpha_2$ was calculated from the ratio of $\Delta_L$ to

$\Delta_{L-1}$. The decision as to which loop should be assigned all delay is reduced to a comparison of the $\Delta$-vector elements for the loops involved.

When viewed as a linear programming problem, the two cases given above form the *extreme points* of the convex set delineated by Formula 9. The general solution must be one of the extreme points [7], so the two cases provide the general solution for a double loop where $\Delta_{L-1}$ and $\Delta_L$ are negative:

| Choose | | When |
|---|---|---|
| $d_L = 0$ | charge outer loop | $\frac{(N_{L-1}-1)}{(N_L-1)} < \frac{\Delta_{L-1}}{\Delta_L}$, |
| $d_{L-1} = 0$ | charge inner loop | otherwise |

Similar inequalities can be derived for $\Delta$-vectors of other signs. An interesting case occurs where $\Delta_L$ is positive[6] and $d_{L-1} = 0$. For Formula 9 to be satisfied, $d_L$ must become negative: iteration $i$ is started $d$ time units *after* iteration $i + 1$. A doacross schedule for a negative delay $d$ corresponds to running the loop in reverse with a delay of $- d$. The accommodation of negative delays requires straightforward modifications of the timing formulae (such as Formula 10) to account for the magnitude, and not the sign, of delays. Of greater concern is the conflict that could arise when two dependences place unsatisfiable constraints on delay: one dependence could require a positive delay while another dependence requires a negative delay. In the following section, an efficient algorithm is presented that sacrifices optimality but avoids such a conflict.

### 2.4.6 A General Solution

The single loop case was derived by eliminating the references to $\tau()$ from Formula 8 in the absence of any outer loops. Expanding $\tau()$ once by substituting Formula 3 into Formula 8 exposed the solution to the double loop case. Further expansion of $\tau()$ leads to the general delay formula for a dependence nested in $L$ loops between statements $S_j$ and $S_i$:

$$- \sum_{l=1}^{L} \Delta_l d_l \geq T(S_j, S_i) \qquad [15]$$

When posed as a linear programming problem, Formula 15 is the constraint under which the following objective function should be minimized:

$$T(DO_1) = \sum_{l=1}^{L} (N_l - 1) d_l + T(S_1, S_s)$$

The objective function is exactly the formula used to time the multidimensional doacross model in Formula 2. The *simplex algorithm* can find optimal values for the delay in all loops, and although this algorithm's possible running time is exponential, its expected running time is reasonable [6]. However, the presence of multiple dependences and $\Delta$-vectors in a loop greatly increases the complexity of choosing optimal delays. This section presents an algorithm that performs a single pass over the loops surrounding a block of statements to establish the minimum delay that should be observed in the surrounding loops due to dependences in that block of statements. Consider a nest of loops as shown in Figure 2 and the calculation of delay $d_k$ at nest $k$ to satisfy the dependence $S_i \delta S_j$, when each statement is from block $B_k$ or block $A_k$. Let $\Delta$ be the distance vector associated with $S_i \delta S_j$. Assume $d_{k+1}, d_{k+2}, \ldots, d_L$ are already known, but $d_1, d_2, \ldots, d_{k-1}$ are as yet undetermined. Since only one pass is desired over the loops, $d_k$ must be set such that Formula 15 is satisfied even if $d_1, d_2, \ldots, d_{k-1}$ are all 0. Therefore, $d_k$ is set such that for $S_i \delta S_j$ and the dependence's associated distance vector $\Delta$,

$$- \sum_{l=k}^{L} \Delta_l d_l \geq T(S_j, S_i)$$

The algorithm consists of two nested traversals of the loop nest shown in Figure 2. The outer traversal calls for the assignment of delays that satisfy the dependences of blocks $B_j$ and $A_j$. The inner traversal considers the dependences and assigns delays appropriately in the loops that surround blocks $B_j$ and $A_j$.

```
Input:
    A nested structure of $L$ loops as in Figure 2
Output:
    Delays $d_1, d_2, \ldots, d_L$ that satisfy all dependences in the $L$ loops.
for j:=L downto 1 do
    Let $\gamma$ be the dependence graph for statements in blocks $B_j$ and $A_j$.
    for k:=j downto 1 do
        Let Q = all pairs $\{((S_i, S_j), DELTA)\}$ such that $i \geq j$ and $(S_i \delta S_j) \in \gamma$ with distance vector $\Delta$.
        Let $Q = Q \cup \{((S_e, S_1), -1)\}$ if $S_e$ exits from loop $k$.
        $d_k = \max_{q \in Q} \frac{\sum_{l=k+1}^{L} \Delta_l d_l + T(S_j, S_i)}{-\Delta_k}$.
    end for
end for
```

Note that the algorithm establishes the delay at each loop as the maximum delay required by dependences associated with that loop. At loop $k$, if some dependence required a negative delay while others required a positive delay, then loop $k$ would be assigned a positive delay. Any dependence that requires a negative delay will be eventually accommodated as the algorithm proceeds to the outermost nest, since an LBD must have some $\Delta_l < 0$, $l < k$ (the dependence is from a sequential program).

## 2.5 Comparison with Other Methods

This section compares doacross with two other techniques for multiprocessor compilation of loops. The first method, *loop distribution*, is the direct application of a transformation developed for SEA compilation that separates the vector and sequential statements of a loop into multiple loops [10, 14]. The second method, *partitioning*, is targeted for multiprocessors [8]; vector and sequential statements are not separated, but statements that participate in a dependence cycle are placed in a partition and the iterations of a loop are pipelined through the partitions [5]. Neither of these two methods provides the performance of doacross.

[6] $\Delta_{L-1}$ is always negative for a flow dependence from a sequential program.

### 2.5.1 Loop Distribution

Consider a loop $L$ of $N$ iterations with $q$ statements participating in dependence cycles and $s - q$ statements that appear in vector loops after loop distribution. Let the statements be numbered such that statements $S_1$, $S_2$, ..., $S_q$ are the statements that participate in dependence cycles and statements $S_{q+1}$, $S_{q+2}$ , ..., $S_s$ are the statements that appear in vector loops after distribution. Only one processor is used for the sequential execution of the $q$ statements; the time for executing the sequential statements is therefore $N \times T(S_1, S_q)$ . $N$ processors are used for the vector execution of the $s - q$ statements; the time for executing the vector statements is therefore $T(S_1, S_s) - T(S_1, S_q)$ . The total time for executing the statements of loop $L$ is $(N - 1) \times T(S_1, S_q) + T(S_1, S_s)$ . The time for executing the distributed loops is identical to the time for executing a doacross schedule of loop $L$ when $S_q \delta S_1$ and the delay is therefore $T(S_1, S_q)$ . In this example of loop distribution, $S_q$ and $S_1$ are members of a strongly-connected component of the dependence graph of loop $L$. Since this condition does not imply $S_q \delta S_1$, the delay for the doacross schedule of loop $L$ is bounded by $T(S_1, S_q)$.

Distributing the loops causes another problem for multiprocessors even if $S_q \delta S_1$. The sequential portion of the loop uses one processor, but the vector portion utilizes $N$ processors. If $N$ processors are dedicated to the distributed loops of loop $L$, then $N - 1$ processors are idle during the execution of the distributed sequential loops. These sequential loops dominate the computation for large $N$. If processors are allocated dynamically, the processor allocation for loop $L$ grows from 1 to $N$ processors when a distributed vector loop is encountered and diminishes from $N$ to 1 when a distributed sequential loop is encountered. This fluctuation in processor usage is expensive when the overhead for processor allocation and deallocation is significant. In [4], a formula is developed for determining the maximum number of processors that a doacross loop with a given delay can use. If $S_q \delta S_1$, the delay assigned by doacross is $T(S_1, S_q)$ . The number of useful processors (as determined by [4]) is $T(S_1, S_s)/T(S_1, S_q)$ . These processors are allocated for the duration of the execution of loop $L$. In terms of program performance, doacross can use fewer processors yet provide at least the performance of loop distribution for multiprocessors. In terms of processor usage, the schedules produced by doacross utilize processors more uniformly than the schedules produced by loop distribution.

### 2.5.2 Partitioning

In [8], there are two types of partitioning: *total partitioning* and *partial partitioning*. Total partitioning is possible only when a loop contains no dependence cycles; such loops correspond exactly to zero-delay loops under doacross. These loops are also called *forall* loops in [5]. Doacross assigns delays to loops that contain lexically-backward dependences that (by definition) participate in a dependence cycle.

Consider a loop $L$ of $s$ statements: $S_1$ , $S_2$, ..., $S_s$. Partial partitioning uses the distance vectors of dependences to divide the index set of a loop $L$ into independent partitions; each partition can then be assigned to a processor. The number of partitions created by the partial partitioning algorithm is given by the greatest common divisor of all negated distances in the dependence graph of loop $L$. Consider the loop of Figure 7. There is a dependence arc between each pair of adjacent statements, and data is either passed within an iteration ($\Delta = 0$) or across one iteration ($\Delta = -1$). The greatest common divisor of the negated distances is 1; only one partition is created by the partial partitioning algorithm of [8]. Consider the pipelining algorithm of [5] that is based on partial partitioning. The statements of a loop are partitioned by the strongly-connected components of the data dependence graph associated with the loop. Let $\Pi_t$ be the most time-consuming partition of a loop. The delay associated with a pipeline schedule is the time for executing the statements of $\Pi_t$.

Doacross introduces a delay that accounts for the most time-consuming lexically-backward dependence. Consider the partition $\Pi_t$; unless there is a dependence from the last statement of $\Pi_t$ to the first statement of $\Pi_t$, the delay introduced by doacross is strictly less than the delay introduced by [8] and [5] pipelining. The example of Figure 7 shows that doacross provides improved performance over partitioning. Pipelining places $S_1$ through $S_5$ in a single processor; since the dependence $S_5 \delta S_1$ does not exist in Figure 7, doacross can schedule multiple processors for the execution of the loop.

## 3.0 Summary and Extensions

This paper characterizes loops that obtain no performance gain on an SEA machine, yet benefit from execution on a multiprocessor. Such loops are parameterized by a measure of their parallelism. This measure defines the optimization problem of rearranging statements to minimize delay (thus maximizing parallelism). In [4], this problem is formally defined and a heuristic is given for its solution. For special dependence graphs, Munshi has shown optimal solutions [12]. Doacross also assumes that statements appear in the same order within each processor. Cuny has shown that by allowing the order to differ between processors, delay can be reduced [3]. In [4], the techniques presented in this paper are applied to routines from the EISPACK package [13], with very favorable results.

For this work to be useful, a mapping of iterations to processors must be defined. [4] presents a processor allocation algorithm for doacross loops, modelled after the *predictor-corrector* algorithms for numerical analysis.

## 4.0 Acknowledgements

The author is grateful to his colleagues at the University of Illinois at Urbana-Champaign who contributed to the development of this material as a dissertation, especially David Kuck, who as thesis advisor directed and motivated this work. Thanks also to Constantine Polychronopoulos and Jih-Kwon Peir, for their helpful suggestions and reading of this work. The author also thanks members of the Advanced Fortran group at IBM Research, especially Fran Allen and Michael Burke. Michael Burke contributed a shorter and more intuitive development of the material in Sections 2.4.3 and 2.4.5 . The author also thanks a referee for his thorough reading of this paper and suggestions. Thanks also to Ken Zadeck and Marc Donner for their text processing expertise.

## Bibliography

[1] Utpal Banerjee, Data Dependence in Ordinary Programs, University of Illinois at Urbana-Champaign, November 1976. M.S. Thesis.

[2] Utpal Banerjee, *Speedup of Ordinary Programs* PhD Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign,. 1979.

[3] Jan Cuny, personal communication, 1986.

[4] Ron Cytron, *Compile-Time Scheduling and Optimization for Asynchronous Machines* PhD Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois,. 1984.

[5] James Russell Beckman Davies, Parallel Loop Constructs for Multiprocessors, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, 1981. M.S. Thesis.

[6] Walter W. Garvin, *Introduction to Linear Programming*, McGraw-Hill Book Company, Inc., New York, 1960.

[7] Saul I. Gass, *An Illustrated Guide to Linear Programming*, McGraw-Hill Book Company, Inc., New York, 1970.

[8] David Alejandro Padua Haiek, *Multiprocessors: Discussion of Some Theoretical and Practical Problems* PhD Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois,. 1979.

[9] Ken W. Kennedy, Automatic Translation of Fortran Programs to Vector Form, Rice University, Houston, Texas, October 1980.

[10] Bruce R. Leasure, Compiling Serial Languages for Parallel Machines, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, November 1976. M.S. Thesis.

[11] Samuel Pratt Midkiff, Automatic Generation of Synchronization for High Speed Multiprocessors, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, 1986. M.S. Thesis.

[12] Ashfaq Munshi, personal communication, 1986.

[13] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines - Eispack Guide*, Springer-Verlag, Heidelberg, West Germany, 1976.

[14] Michael J. Wolfe, Techniques for Improving the Inherent Parallelism in Programs, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, 1978. M.S. Thesis.

[15] Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers* PhD Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois,. 1982.

# A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS

Robert G. Babb II
Lise Storc
Dept. of Computer Science and Engineering
Oregon Graduate Center
Beaverton, OR

William C. Ragsdale
Parallel Processing Systems
Control Data Corporation
Monterey, CA

**Abstract**—*Large-Grain Data Flow* (LGDF) combines dataflow techniques with traditional sequential programming to provide a high level model for parallel processing. The user designs parallel applications using networks of program modules connected by datapaths. The resulting specification is automatically transformed for efficient parallel scheduling on a particular underlying architecture. Program execution is controlled via producer/consumer data synchronization. This paper describes the LGDF parallel processing approach, and proposes methods for implementing efficient LGDF schedulers for a specific computer that employs a ring communication architecture, the CDC CYBERPLUS.

## 1. INTRODUCTION

The difficulty in achieving reliable, efficient, maintainable, and portable parallel programs argues strongly for the use of higher level, possibly more restricted, models of parallelism than those provided by basic machine facilities. Programmers can use the higher level model to design applications, and then can rely on automatic means (such as a preprocessor) to generate a concrete implementation tailored for a particular parallel machine. Our experience has shown that programming within the restrictions of a suitable higher level parallel model can make parallel program design significantly more reliable and *even easier* than design using a sequential computation model[1].

Several other approaches to solving the problems of parallel programming have been proposed. Automatic transformation, advocated by Kuck and others[2], envisions development of software tools to automatically transform "dusty deck" FORTRAN for parallel processing. Complex control and data dependencies make this approach very difficult, and results can be poor. Another problem with the automatic transformation approach is that it does not lead to development of novel, highly efficient parallel algorithms, which user-visible parallelism tends to encourage.

At least for the near future, layered languages show promise for making up for the deficiencies of FORTRAN as a parallel processing language. Users code parallel operations using macros. The macros are expanded to produce low level synchronizing operations suitable for a particular parallel machine. This tends to lead to fewer bookkeeping errors in the coding of process synchronization, as well as providing a degree of portability of code between different parallel processors. Ideally, one can simply modify the definitions of the macros to get layered language programs running on a different parallel machine.

A particular layered language approach, Large-Grain Data Flow (LGDF) is described in Section 2. Section 3 presents a brief discussion of the architecture of the CDC CYBERPLUS emphasizing characteristics that are relevant to parallel program interaction. Possible LGDF scheduler implementation strategies for CYBERPLUS are discussed in Section 4. Conclusions and extensions are given in Section 5.

## 2. OVERVIEW OF LGDF

Large-Grain Data Flow (LGDF) combines some features of dataflow computation with subroutine-like code resembling traditional sequential programming. A comparison between LGDF and sequential coding is illustrated in Fig. 1. The main difference between the data flow and sequential versions for this small program lies in the way that the specified computation (C=A+B) is *scheduled*. Scheduling occurs in traditional sequential programming via the flow of control, represented by the successive values of the program counter. The subroutine P10 will be *called* when the process reaches appropriate control points in the program. In contrast, the LGDF program P10 is scheduled for execution asynchronously, whenever appropriate, based on the arrival of the input values A and B, and on the availability of buffer space to write out a new computed value for C.

```
SEQUENTIAL                  LARGE-GRAIN DATA FLOW
PROGRAMMING                 PROGRAMMING

SUBROUTINE ADD(A, B, C)     begin_(p10)

C = A+B                         C = A+B
                                clear_(A)
RETURN                          clear_(B)
END                             set_(C)
                                suspend_

                            end_(p10)
```

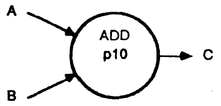


Fig. 1. Traditional and LGDF programs for C = A+B.

The program shown in Fig. 1 is more fine-grained than the typical LGDF program, which corresponds to 5 to 50 or more FORTRAN statements.

### 2.1. LGDF NETWORKS

*Datapaths* link LGDF programs to form *networks* (Fig. 2) that allow communication of data values from one program to another. Datapaths can be in one of two states: *empty* or *full.*

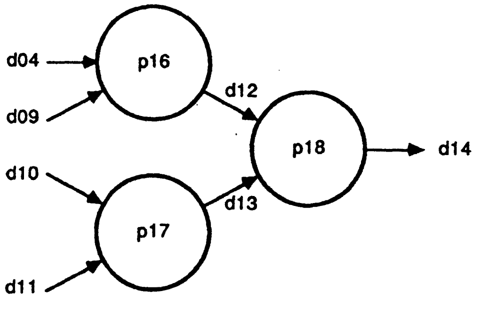


Fig. 2. LGDF network.

0190-3918/86/0000/0845 $01.00 © 1986 IEEE

An LGDF program is activated (asynchronously) depending only upon the state of its associated input and output datapaths:

> **Execution Rule**—A program may start an execution cycle if and only if all of its input datapaths are in the *full* state and all of its outputs are in the *empty* state.

Programs can change the state of an output datapath from empty to full by means of the LGDF function *set_*. A *set_* has the effect of making the data associated with a datapath available to activate downstream programs. In a similar fashion, programs can *clear_* their inputs in order to make the space available for writing into by upstream programs.

LGDF networks coordinate parallel process activation via data flow interactions. Each datapath arc in a network represents both a ***shared memory*** and an associated data flow interlock mechanism. The control mechanism provided by the program execution rule can be used to synchronize access to shared data areas with a simple producer/consumer protocol. This protocol precludes race conditions for shared data (write/read conflicts).

An LGDF program that has finished computation and data flow interactions for the current execution cycle can ***suspend_***. At that point, it is subject to the:

> **Data Flow Progress Rule**—Upon suspension of an execution cycle, a program must have cleared at least one input or set at least one output datapath. Otherwise, it is terminated.

Programs that are in the ***terminated*** state cannot be scheduled for further execution. However, unaffected parts of the same network may still execute. By designing programs that satisfy the Data Flow Progress Rule, many types of parallel program design problems can be avoided.

LGDF networks can be combined into hierarchies, since any node in a network may be defined either by an LGDF program or by another LGDF network. The meaning of this is the same as if the lower level network were substituted graphically for the higher level network abstraction.

Applications designed using LGDF are implemented with the aid of a prototype set of software tools that rely on macro-expansion techniques to generate appropriate scheduling code for various target computers. Several steps are involved in LGDF programming. First, a hierarchical set of LGDF network data flow graphs is designed, based on logical data dependencies inherent in the application. The structure of the resulting network is encoded in a ***wirelist*** file. Data declarations for each datapath in the network are packaged separately. The programmer then writes the LGDF programs, combining FORTRAN with a small set of LGDF macros. Finally, the LGDF programs are macro-expanded to produce compilable FORTRAN code for the particular machine, and this code is compiled and executed. For more details on the software tools and use of LGDF for parallel processing see [3].

The basic idea in LGDF programming is to balance the average grain size of LGDF programs in a network with scheduling overhead in order to reduce the significance of the overhead. This will in general depend on the capabilities of the given architecture, the efficiency with which the LGDF implementation mechanisms exploit those capabilities, the particular application algorithm, and the strategy for data flow parallelization of the algorithm.

## 3. CYBERPLUS ARCHITECTURE

A Control Data CYBERPLUS Parallel Processing System[4] [5] consists of an ensemble of up to 16 very fast processors (20 nanosecond cycle time) organized in a ring structure and interfaced to a host CYBER computer. There is a great deal of internal parallelism possible within one CYBERPLUS CPU. Each processor can contain up to nineteen independent functional units. A crossbar interconnection provides for the routing of results from the output of each functional unit to the input of any or all functional units on the next cycle. A CYBERPLUS processor contains two types of memory, a bulk memory (up to 512K 64-bit words) for data and a smaller (4K 240-bit words) program memory for executable code. Code overlays can be stored in the bulk memory for rapid transfer to the program memory as needed.

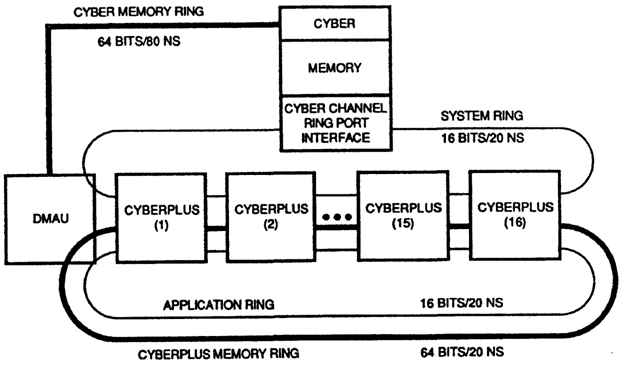


Fig. 3. Typical CYBERPLUS ring group.

At a higher level of parallelism, up to 16 CYBERPLUS processors can be interconnected in a ring group and interfaced to the host mainframe, as shown in Fig. 3. Up to four ring groups can be interfaced to the one host computer.

Two rings allow for the transfer of 16-bit data, the System Ring and the Application Ring. These rings interconnect the CYBERPLUS processors. The System Ring also provides an interface to the host CYBER computer through its standard I/O channel. The CYBER Memory Ring provides a 64-bit wide data path between the CYBERPLUS bulk memories and the memory of the host computer through a direct memory access port. The CYBERPLUS Memory Ring allows 64-bit transfer of data between the CYBERPLUS bulk memories.

### 3.1. SYSTEM AND APPLICATION RINGS

This dual ring system provides a high speed parallel channel for interconnecting the processors and the host computer of a CYBERPLUS Parallel Processing System. The rings are formed by connecting the individual ring ports of the processors in a circular structure. Each ring port has an input and output register. These channels are 29 bits wide, with 16 bits of data and 13 bits of control information making up a ring packet. Ring packets move sequentially and synchronously around the ring from one register to the next, at the rate of 20 nanoseconds per transfer. This results in a peak data transfer capability between processors of 800 megabits per second per ring. A ring port can remove a ring packet from the ring and place the same or a new ring packet on the ring on every cycle. Ring packets currently on the ring have priority over ring packets trying to enter the ring from a processor.

Each processor has two ring ports, one for the System Ring and one for the Application Ring. The System Ring ports respond to an extended instruction set that provides additional control over the CYBERPLUS processors. The host uses the System Ring to transfer programs and data to and from each processor in a ring group and to initiate the execution of programs in the processors.

The System Ring and the Application Ring can be connected to circulate data in opposite directions, allowing processors equal access to both left and right neighboring processors. Both rings can also be used by application programs for the transfer of data among the processors and for synchronization of the application processes running in the processors.

## 3.2. CYBER AND CYBERPLUS MEMORY RINGS

The CYBER Memory Ring and the CYBERPLUS Memory Ring are supported by the Direct Memory Access Unit (DMAU). Via the System Ring, a program initiates, monitors, and terminates all of the data transfers conducted by the DMAU. The DMAU also provides a queueing system to allow the stacking of data transfer requests, reducing the transition time from one data transfer request to the next. A data transfer request may specify multiple destinations, allowing the broadcast of a block of data to any or all processors in the ring group on one trip around the ring.

The CYBERPLUS Memory Ring allows block transfers between the local bulk memories of the processors to proceed at a rate of one 64-bit word each 20 nanosecond cycle. This results in a maximum transfer rate of 3200 megabits per second. Block transfers between CYBERPLUS local memories and CYBER host memory use the CYBER Memory Ring and proceed at a rate limited by the maximum access rate of the host memory system (800 megabits per second).

The DMAU provides a much higher bandwidth data path between the host computer and the CYBERPLUS local memories than the System Ring (800 vs 24 megabits per second). It also supports higher interprocessor data rates than the System and Application Rings alone (3200 vs 1600 megabits per second).

## 4. IMPLEMENTATION OF LGDF ON CYBERPLUS

Our primary goal in designing an LGDF scheduler for the CYBERPLUS is to minimize scheduling overhead. To achieve this, we need to minimize the time when one processor must lock out other processors from accessing shared scheduler tables. Efficient mechanisms must also be designed for the transfer of data among LGDF processes. Secondary considerations include providing the ability to schedule arbitrarily large LGDF networks, and scheduling processes efficiently with any number of processors. Also important is the ability to simulate LGDF parallelism on a single CYBERPLUS processor. This aids in implementing large networks, provides the ability to debug applications in a more controlled (less parallel) setting, and provides greater flexibility in assigning processes to processors.

The smaller the average grain size in a given application, the more opportunity there is for parallel computation. However, for the CYBERPLUS, it would be inefficient to schedule computations on a very low level, such as the basic arithmetic operations. The amount of sequential computation in an average grain should be large enough relative to the scheduling process to reduce overhead to an acceptable percentage, say ten percent or less.

Since the amount of code required for an LGDF scheduler is typically quite small, we have chosen to implement a resident scheduler in each processor. This avoids the overhead of passing messages between a processor and a single central scheduling process. In addition, parts of the scheduling process can accomplished in parallel. However, to maintain scheduler integrity, the distributed LGDF schedulers must obtain exclusive access to certain system information contained in shared tables for a portion of the scheduling process.

### 4.1. LGDF SCHEDULER TABLE INTERLOCKING

Even though the CYBERPLUS has no shared global memory, the effect of a shared table can be implemented by keeping a copy of it in each processor's local memory. A processor wishing to update a system table does so under the protection of an interlock scheme. The processor then updates its local copy, broadcasts the new table to all other processors via the CYBERPLUS Memory Ring, and releases the interlock[1].

To implement the required interlocking, a token passing scheme is used. Two different types of tokens are required: D tokens to protect access to (subsets of) datapath state tables, and P tokens to protect access to (subsets of) LGDF process state tables. In the following discussion, we will assume that only two tokens exist, one of each type. The processor which currently owns each token is known by the schedulers on all other processors. If a processor needs a token which it does not own (e.g., because an LGDF process which it is executing requests a *clear_* or *set_*), it sends an interrupt message over the ring to the current owner. If the current owner of the token no longer needs it, the token is sent to the requesting processor by broadcasting the processor number of the new owner to all processors via the ring. Thus, when ownership changes, each requesting processor is notified of the new owner and can determine if it now has the token. If a requesting processor did not get the token on a change of ownership, a new request can be made to the new owner for the token.

The requirement that the scheduler be able to handle arbitrarily large LGDF networks, combined with the limited program memory of CYBERPLUS, indicates that LGDF programs should be stored as overlays in the CYBERPLUS bulk memory. When the scheduler wishes to execute an LGDF process, it loads the overlay from bulk memory into program memory and then branches to it. This also allows each processor to execute any of the LGDF processes of the application.

P (process) token interlocking behaves as follows. First, the scheduler obtains the P token which allows it to examine the process queue. The process queue contains the name of all processes which have previously been identified as eligible to execute. A process is removed from the queue and the revised queue is broadcast to all other processors via the CYBERPLUS Memory Ring. The P token is then released and the appropriate LGDF process overlay is loaded into program memory and executed. When the process suspends, it returns to the scheduler, which checks for violation of the LGDF Data Flow Progress Rule, then attempts to select a new process for execution.

If the process queue was empty when the P token was obtained, the scheduler must also obtain the D (datapath) token. All eligible processes (i.e., that satisfy the LGDF Execution Rule) that are not currently executing are placed into the eligible process queue, and scheduling resumes as described above[2]. If the eligible process queue is still empty, the scheduler enters an idle loop. When any processor changes the state of a datapath (full to empty or empty to full), it sends a special "change message" to all processors via the ring. Any schedulers in the idle loop will be interrupted by this message. They then compete for the P token in order to obtain a process to execute.

To achieve independence of the scheduling code from the number of processors, all processors are treated as equals (there are no masters or slaves). All CYBERPLUS processors contain the same application programs and scheduling code, as shown in Fig. 4. The processors compete for the chance to advance the state of the computation by running any eligible process, as described above.

### 4.2. IMPLEMENTING LGDF DATAPATHS

Since each processor can execute any of the LGDF processes, all datapaths of the network must be available to each processor. This can be accomplished by keeping a copy of each datapath in the bulk memory of each processor. A process which wants to write on a datapath first updates its local copy of that datapath,

[1] It might seem more efficient to broadcast updates to system tables, rather than the entire table, but the tables required to store the data flow state for LGDF networks tend to be very small (a few bits per datapath and per process).

[2] Note that the P and D tokens are not explicitly released. The scheduler is merely willing to pass the tokens to whichever scheduler requests it next.

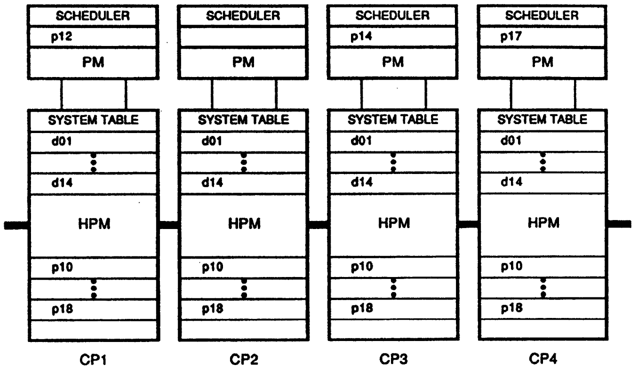


Fig. 4. LGDF scheduler on CYBERPLUS.

then broadcasts the update via the CYBERPLUS Memory Ring. The process then can then change the state of the datapath to full by executing a *set_*. (In the simplified producer/consumer model of LGDF parallel processing under consideration here, each datapath can only be written by a single LGDF process, so no write/write conflicts are possible). A drawback to this scheme is that when datapaths represent large arrays, the duplication can consume large amounts of bulk memory.

Several alternatives exist when this duplication causes a problem with resources. As shown in Fig. 5, the memory of the host processor can be used as a global memory for the processors in the ring group. In this scheme, the producer process writes the array to host memory and places a pointer to it in the datapath, which is then broadcast to all processors in the ring group. When the consuming process executes, it can load the array directly into its local memory, thus avoiding duplication in other processors.

Fig. 6 shows another alternative that avoids the transfer time to and from host memory. However, this method requires a memory manager on each processor as part of the scheduler. In this scheme, the producer process obtains a block of local memory from its scheduler to contain the large array to be passed to the consumer process. The address of this block and processor number are placed in the datapath, which is broadcast to all other processors in the ring group. The producer process can suspend and allow other processes to execute in that processor, but the block containing the array is protected by being reserved through the memory manager. When the consumer process for this array executes, the datapath will contain the processor number and the address of the required array. The array is transferred directly to the consumer processor via the Memory Ring. The scheduler then sends a message via the ring to the producer processor's memory manager, releasing the block of memory in that processor.

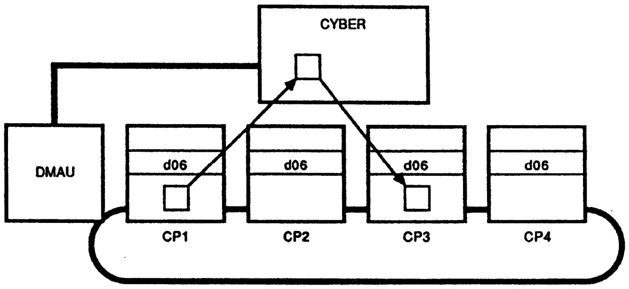


Fig. 5. Datapath via CYBER memory.

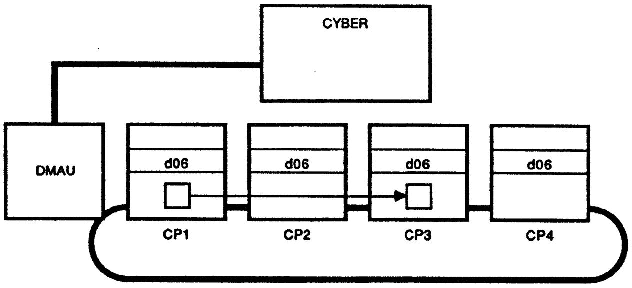


Fig. 6. Datapath via managed memory.

Another method for handling extremely large arrays is through use of disk storage on the host (or directly on the ring group). The array is written to disk, perhaps in piecemeal, by the producer processor. A descriptor pointing to it is placed in the datapath, and the datapath is broadcast via the CYBERPLUS Memory Ring to all other processors. The consuming process then references the datapath to obtain the disk descriptor and reads the data, again perhaps piecemeal, directly into its local memory.

## 5. CONCLUSIONS AND EXTENSIONS TO WORK

An alternative approach to placing schedulers and all code and data in each CYBERPLUS processor is to download code and data from the CYBER host onto a CYBERPLUS only as needed, with all scheduling done by the host. This implementation would reduce duplication of program and data storage, but the scheduling overhead would be much larger. Thus, a larger average grain size would be required to keep the significance of overhead small. Another possible strategy is to allow LGDF programs running on the host to interact transparently with LGDF programs on the CYBERPLUS processors, and migrate back and forth automatically.

The design described in this paper envisions no distinction among CYBERPLUS processors -- any processor can run any LGDF process. For very large systems, there could be specialized schedulers on different CYBERPLUS nodes. One or more CYBERPLUS processors would be dedicated to each portion of the total LGDF network. This would require the ability to subset the shared system tables. For example, in a CYBERPLUS configuration with multiple ring groups, subsets of the network could be assigned to each ring group, with a master scheduler located on the CYBER host coordinating data flow between sub-nets.

## 6. REFERENCES

[1] R. G. Babb II, "Data-driven implementation of data flow diagrams," in *Proc. 6th Int. Conf. on Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 309–318.

[2] D. D. Gajski, D. A. Padua, D. J. Kuck and R. H. Kuhn, "A second opinion on data flow machines and languages," *Computer*, vol. 15, no. 2, Feb. 1982, pp. 58–69.

[3] R. G. Babb II, "Programming the HEP with Large-Grain Data Flow Techniques," in *MIMD Computation: HEP Supercomputer and Its Applications*, (ed. by J. S. Kowalik). Cambridge, MA: The MIT Press, 1985.

[4] Control Data Corporation, "CYBERPLUS Hardware Reference Manual," Publication No. 77960981.

[5] Control Data Corporation, "CYBERPLUS System Software and Utilities Reference Manual," Publication No. 60561680.

# *Macro vs Micro Dataflow: A Programming Example*

*Maya B. Gokhale*

*University of Delaware*
*Newark, Delaware 19716*

Although there are yet to be developed formal methodologies for parallel algorithm design, several important principles (domain decomposition, substructuring, dataflow) have been established as useful in writing parallel algorithms. This paper is an exploration into the principle of data flow, examined at two levels- macro (or task level) and micro (or instruction level). Several versions of concurrent algorithms are presented for a simple linear algebra problem. The macro dataflow programs use an extended Fortran for the Denelcor HEP. The micro dataflow programs use a dataflow programming language Manchester Dataflow (MaD). Programming effort and efficiency considerations in each environment are contrasted.

## 1. Introduction

The design of efficient parallel algorithms is still in the realm of "black art." However, because of the great progress in developing parallel algorithms, there are certain principles of parallel programming which have been established. This paper investigates a specific methodology- dataflow- by means of a case study. Two parallel environments are studied. The first extends the traditional sequential program by supporting concurrent cooperating tasks which share data structures (task level parallelism). The Denelcor HEP is our example of this programming environment. The second is a tagged token dynamic dataflow system, the Manchester Dataflow Machine [1] programmed in Manchester Dataflow Language (MaD). This machine supports parallelism at all levels, from task to individual instruction.

The next section expands on the notions of task level and instruction level parallelism. Each machine is described briefly. Then, we discuss a simple problem: solving an upper triangular system of simultaneous equations using back substitution. An algorithm is outlined to solve the problem under each of the two computation models. Efficiency issues under each model are discussed. Finally we compare and contrast the programming effort required for each model.

## 2. Two levels of Parallelism

In both task level (also called macro) and instruction level (micro) parallelism, the programming model involves concurrent execution of instructions. At the macro level, the unit of concurrency is the process or task. Within a task, instructions are executed sequentially. A machine to support such a form of parallelism usually follows the sequential computer model. The concurrency is supported in hardware by duplication of processing units (PU). Each PU has its own PC, ALU, and register set. The programmer specifies the concurrency by using such primitives as "create task" (or "fork") to start a new concurrent process, and "synchronize" (or "join") to coordinate execution of concurrent processes.

Instructional level parallelism might at first glance be thought to be at the extreme end of task level parallelism. However, in a task level parallel environment, there is some measureable overhead associated with task set up to context switch among tasks. Efficiency demands that the task be sufficiently large that context switch overhead does not overwhelm the computation time. If parallelism at the instruction level is to be supported, an entirely different computation model (and therefore machine realization) is required.

In practice, data flow has been the most popular model developed for instructional level parallelism. Rather than instruction execution controlled by one (or more) program counter(s), an instruction is executed on a data flow machine when all the operands required by the instruction are available. Completion of an instruction causes a new result to be propagated to other instructions which use it as an operand.

### 2.1 The Denelcor HEP

The HEP is a shared memory multiprocessor with a tagged data memory. Each location may be either full or empty. A location may be read only if the tag indicates that it is full. A location may be written only if the tag indicates that it is empty. The HEP is programmed in an extended Fortran IV. The extensions provide for asynchronous variables used for process synchronization; process creation (a fork command); a coroutine facility; and the ability to specify local variables.

The challenge in creating efficient programs for the HEP is to set up independent concurrent instruction streams. The instruction steams related to one problem constitute a Task System. The processes can be synchronized through the scoreboarded memory [2].

### 2.2 The Manchester University Data Flow Machine

Machine language for a data flow machine takes the form of a directed graph rather than the traditional linear sequence of instructions. A node in the graph represents a unit of computation; an arc represents the flow of data into and out from the units of computation. A node may "fire" or be executed whenever data (also called a "token") is available on the input arcs, and as a result, produce data on the output arcs. Variable names in a data flow language denote values rather than locations [4]. For this reason, reassignment to variables (e. g. , `I := I + 1`) is not generally permitted.

The simple scenario outlined above becomes more complicated in the presence of multiple sets of data for an instruction. To keep data sets distinct, some implementations provide for data tagging. Each token carries with it an identification as to the block level at which it belongs. The tag can also be used to differentiate elements of an array.

Definition of array elements poses a special problem for data flow machines. The semantics of the data flow programming language must ensure that reassignment to individual elements is not permitted. However, when a generalized subscript expression is used to index an array, there is no way to guarantee at compile time that reassignment will not occur. For this reason, data flow languages often have restrictions on the definition of arrays. In the Manchester Dataflow language (MaD) [3], assignment to individual array elements is not allowed. Instead, the programmer must formulate one expression which computes the values of all elements of the array.

## 3. Algorithm Design for Parallelism: An Example

At this point, using dataflow as the guiding principle, we would like to investigate algorithm design for each form of parallelism discussed above, macro and micro. Our goal is to discover: 1) styles of programming and 2) machine characteristics which influence both algorithm performance and ease of programming.

The vehicle we use for the investigation is a very simple problem in linear algebra. The problem is to solve a set of simultaneous equations. The coefficient matrix is assumed to have been reduced to upper triangular form by some standard technique. Thus, back substitution can be performed on the coefficient matrix to solve for the unknowns.

0190-3918/86/0000/0849 $01.00 © 1986 IEEE

The problem may be formulated as follows:

Given a set of n equations in n unknowns,

$$\begin{array}{cccccc} a(1,1)x(1) & a(1,2)x(2) & \cdots & a(1,n)x(n) & = & b(1) \\ 0 & a(2,2)x(2) & \cdots & a(2,n)x(n) & = & b(2) \\ 0 & 0 & \ddots & & & \\ 0 & 0 & \cdots & a(n,n)x(n) & = & b(n) \end{array} \qquad (1)$$

find $x(i), i = 1 \cdots$ n , where x(i) is given by

$$\frac{x(i) = b(i) - \sum_{k=i+1}^{n} \quad a(i,k) * x(k)}{a(i,i)} \qquad (2)$$

### 3.1 Back Substitution on the HEP

We first look at a concurrent task-based solution to (2). Let each process P(i) be responsible for the computation of x(i).

```
P(i):
    (1) Form the sum a(i,k)*(k),   k = i+1 ... n.
    (2) Take b(i) - (1).
    (3) Let x(i) = (2) / a(i,i).
```

In addition to the code for each process, we must plan for task initiation and synchronization among the tasks. For synchronization, semaphored variables are used. We will make the x vector a semaphored variable, a "$" variable in HEP Fortran. The "$" variables are placed in the special scoreboarded memory. $x(n) may only be read by P(n-1) when the location is FULL. The algorithm is written so that only P(n) writes $x(n). Therefore, even if P(n-1) tries to read $x(n) before the value has been computed, the process will merely wait on the FULL condition. The programmer is responsible for correct use of the scoreboarded variables to synchronize tasks. The graph of Figure 1 shows the flow of shared data (e. g., the $x variables) among tasks.

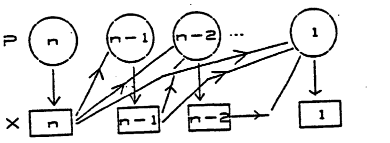


Figure 1. Flow of shared variables among tasks

Unfortunately, we cannot merely prepend an x with "$". When process P(i) reads a semaphored variable, it sets the semaphor EMPTY. Thus all the other processes are stalled: each is waiting for FULL on the variable, and this event will not occur again. Our solution to this problem involves having multiple copies of the variable, one for each process using it. Figure 2 shows the HEP version of the back substitution algorithm. x(i) is the local copy of P(i)'s result. The scoreboarded copies are written to $x(Processor-num, i), where the processor-num can vary from 1 to i-1.

```
For i=n to 1 by -1
    Create process P(i)

P(i):
    sum=0
    for k=n to i+1 by -1
        sum = sum +
            a(i,k) * $x(i,k)
    $x(i) = \frac{(b(i)-sum)}{a(i,i)}$
    for k=i-1 to 1 by -1
        $x(k,i) = x(i)
```

Figure 2. Synchronized Task-based Back Substitution

With this approach, counting only floating point operations and synchronization stores, a 4-equation problem takes 20 timesteps. Each additional equation adds a cost of 4 timesteps.

A slight modification to the order of evaluation of intermediate terms of the program in Figure 2 results in an improved execution time. We revise the body of P(i) as in Figure 3. The subtraction of the sum products on the left from b(i) on the right is done at the beginning rather than at the end. With this approach, the cost of each additional equation is 3 timesteps rather than 4 because of the overlapped subtraction.

In summary, there are two classes of program design we have employed. The optimizations represented by Figure 3 could be made by a fairly standard mechanical optimizer. They involve expression rearrangement and partial loop unfolding. However, the decision at the outset to create a process for each equation; to run the loops from n downto 1; the synchronization method: these design decision are not as readily mechanizable. This class of decision is made by the knowledgeable programmer.

```
P(n):
    for k = n-1 to 1 by -1
        $x(k,n) = $\frac{b(n)}{a(n,n)}$

P(i) (i<n):
    sum = b(i) -
        a(i,n)*$x(i,n)
    for k=n-1 to i+1 by -1
      sum = sum -
        a(i,k)*$x(i,k)
    x(i) = $\frac{sum}{a(i,i)}$
    for k=n-1 to i+1 by -1
    $x(k,i) = x(i)
```

Figure 3. Task-based Back Substitution Optimized

### 3.2 Back Substitution on the Data Flow Machine

We next implement an algorithm to solve the same problem on the data flow machine. On the HEP we manually decompose a problem into concurrent tasks. We must observe from the problem description which groups of operations can be done in parallel and then explicitly code those groups into discrete tasks. In contrast, on the data flow machine, the concurrency is managed automatically by the hardware. We can look at the algorithm as a whole rather than as a collection of cooperative tasks. The difficulty on the data flow machine is to formulate and code the algorithm correctly in the data flow language.

On the HEP a vector $x in the scoreboard memory holds the values of x(i). Elements of the vector are defined incrementally by different tasks. In MaD this incremental definition is not allowed. Instead the vector must be defined completely by one expression. This constraint results in a programming style unfamiliar to most conventional language programmers.

In MaD an array is represented by a (possibly multi-dimensional) STREAM. A stream is a sequence of objects of some type terminated by a special token called "end-of-stream". The vector of unknowns must be declared in MaD as a "stream of real". Instead of the familiar definition

$$x(i) = (b(i) - sum)/a(i,i)$$

we must use list processing operators such a CONS to put together the vector x. By using CONS, we create a new list which is the old list plus a new x(i).

The vector x is produced as follows:

The value of one element of x depends on values of other, previously computed elements. For example, the value of x(n-1) depends on the value of x(n). In MaD we handle this problem by iteratively producing new versions of x. Initially x is null. Then a new version of x is created which has one element, written [x(n)]. The next version of x has two elements, [x(n-1), x(n)], etc. Essentially, we are manipulating the vector as a list, and using the cons operator to put a new element to the front of the list.

This seems to violate the single assignment rule. However, this limited form of reassignment is permissible in MaD as long as the reassignment is done within a repetitive block, and we differentiate between "old" and "new" values of x at each iteration. The MaD code for computing x is as follows:

```
new x = cons( (b(i) - sum) / a(i,i), x(i))
```

The next problem is the computation of sum, which is the name used for the

```
sum(a (i,k)*x(k) ), k = i+1 ... n.
```

This computation illustrates another way to specify concurrency in MaD, the FOR EACH clause. Given a stream, in this case, the current list x, the clause "for each x" causes some action to be performed involving each element of the list x. For the sum block, the product a(i,k)*x(k) is formed. sum is computed in a nested block within a while loop in Figure 4, which gives the complete (but incorrect) block in which x and sum are computed. Although this code seems correct at face value, it contains a subtle timing problem. Consider the sequence of computation for x and sum:

Initially `x = [b(n)/a(n,n)]` and `sum = 0`.

The new value of sum is a(n-1,n)*x(n). The new value of x is [(b(n-1) - 0) / a(n-1,n-1), x(n)]. This is incorrect for x(n-1) because the sum used in the computation of x is the OLD sum. Because an incorrect value has been generated for x(n-1), an incorrect value is computed for sum at the next iteration, and the error is propagated to each subsequent x(i). This problem occurs even if sum is initialized to a(n-1,n)*x(n). The new x will be

```
[ (b(n-1) - a(n-1,n)*x(n)) / a(n-1,n-1) , x(n)]
```

but the new sum will be

```
a(n-2,n-1)*x(n) instead of
a(n-2,n-1)*x(n-1) + a(n-2,n)*x(n)
```

The problem is that the new sum computation needs the NEW x, not the old one. The correct code should require the keyword "new" in the for each clause:

```
for each xi in new x do ...
```

Unfortunately, the implementation of the MaD compiler used for this example does not support the use of a "new" version of a variable on the right hand side of a definition. The final (correct) code from which results on parallelism are taken is shown in Figure 5. We are forced to cycle through the loop 2n times. On the "odd" cycle, we update sum, and on the "even" cycle, we update x.

```
declare x:  stream real; i:  integer; sum:  real;
initi := n-1; sum := 0;
    x := [b(n)/a(n,n)]; [* x initially contains x(n) *]
while i > 0 do          [* iterative loop *]
    new i := i-1;
[* add the next x(i) to the front of x *]
    new x := cons((b(i)-sum)/a(i,i), x);
    new sum := declare  [* a nested block *]
        prod:  real; k:  integer; xi:  real;
        init k := i+1; prod := 0;
        for each xi in x do
            [* concurrent loop *]
            new prod := prod + a(i,k)*xi;
            new k := k+1;
        return prod;    [* as the new sum *]
return x;
```

Figure 4. (Incorrect) Data Flow Back Substitution

```
declare x:  stream real; i:  integer; sum:  real;
        oddd:  boolean;
initi := n-1; sum := 0; oddd := true;
    x := [b(n)/a(n,n)]; [* x initially contains x(n) *]
while i > 0 do          [* iterative loop *]
    new i := if oddd then i else i-1;
[* add the new x(i) to the front of x *]
    new x := if not oddd then cons((b(i)-sum)/a(i,i), x)
        else x;
    new sum := if oddd then
    declare                     [* a nested block *]
        prod:  real; k:  integer; xi:  real;
        init k := i+1; prod := 0;
        for each xi in x do
            [* concurrent loop *]
            new prod := prod + a(i,k)*xi;
            new k := k+1;
        return prod     [* as the new sum *]
        else sum;
    new oddd := not oddd;
return x;
```

Figure 5. (Correct) Data Flow Back Substitution

The results for a 4-equation problem are summarized in Figure 6. S1 is the total number of machine instructions executed. Sinf is the total number of time steps needed for the computation given as many processors as could be used. The Average Parallelism figure determines the maximum number of processors which can be utilized, in this case 9 (8.8). The number of result tokens generated can be greater than the number of instructions because some instructions generate two result tokens. The Max Matching Store Occupancy indicates how many tokens were waiting in the associative memory for another token with a matching tag to arrive.

```
s1 = 2526       Sinf = 288
Average Parallelism of 8.8
3114 Result Tokens Generated
Max Matching Store Occupancy = 115 Tokens
```

Figure 6. Data Flow Back Substitution: Unoptimized

In this example, as in the HEP case, there are optimizations which the programmer can use to increase performance. In the computation of sum, prod is created iteratively in spite of the for each construct. The old value of prod is used at each step to compute a new value. MaD (and most data flow languages) have a more parallel construct to accomplish the same function. We can create a stream of a(i,k)*x(k), k = i+1 .. n. All these multiplications can occur in parallel. Another list processing operator ALL is used to create a stream from the products. When individual data items are defined separately, the all operator can be used to gather them into a stream. This operator cannot be used to define x because a new element of x is defined in terms of previously computed elements would not be available incrementally. However, previous values of prod are not needed in this case. When the entire stream is produced, we can apply the reduction "+!" to the steam. This identical to the APL "/" operator: a "+" is inserted between adjacent stream elements, and the new expression is evaluated. If logarithmic reduction is implemented, the n-element steam can be added in log n time steps. The results in emulation from this optimization are summarized in Figure 7. Although the total number of time steps and the maximum matching store occupancy have decreased, there is an added cost in this optimisation: the total number of instructions executed has increased, and more result tokens have been generated. A maximum of 10 PE's could be utilized in this case.

In this case, as in the task-based dataflow, we have observed two categories of optimizations. The iteration replaced by the reduction operator is an optimization well within the range of current optimization techniques. The timing problem and the generation of the stream are in the realm of expert programmer.

```
S1 = 2604 Sinf = 265
Average Parallelism of 9.8
3210 Result Tokens Generated
```

```
Max Matching Store Occupancy = 101 Tokens
```

Figure 7. Data Flow Back Substitution: Optimized

## 4. Conclusions from the Experiment

We find even in this simple problem that the programmer bears the burden of management of parallelism for both task level and instruction level parallel machines. On the HEP, we can observe that

- The programmer bears complete responsibility for organizing the task system, including the decisions as to which groups of operations should go into which tasks.
- The programmer must synchronize among tasks correctly. Shared variables must be allocated in the scoreboarded memory, and a proper discipline of definition and access maintained so that race conditions and deadlock are avoided.
- The programmer must arrange the order of evaluation within a task to maximize concurrency. Shared variables should be computed as early as possible so that waiting tasks can continue.
- Actual allocation of functional units to tasks is handled automatically. Thus at the Fortran level, the programmer need not statically assign processors to tasks.

The first two tasks are not readily automated. However program analysis tools such as graphical display of data dependencies could help the programmer detect errors and/or possible optimizations. The third can be accomplished to a certain extent by mechanical optimization, and the fourth is already handled by the operating system and hardware.

With instruction level dataflow, we have seen that the programmer is not really freed from these concerns in the data flow environment. Data flow languages (of which MaD is a fair representative) do limit the expressive power of the programmer. We note that

- Rather than having access to the more problem-oriented data structures of vectors and matrices, we are forced to use lists and list processing constructs to build the vectors.
- As illustrated in the first version of back substitution in MaD, subtle timing problems can still crop up. We are required to have a detailed understanding of timing considerations as in hardware design ("old" as opposed to "new" values in an iteration) to come up with a correct program. Although we do not have to explicitly code synchronization, we still have to be aware of what is produced when.
- The programmer also bears responsibility for using perhaps less familiar forms of optimize the program. In this example, we had to know about the combination of FOR EACH, ALL, and reduction to parallelize computation of sum. If we code the algorithm in a sequential way, it is not going to be magically parallelized by virture of running on a data flow machine.

Of these points, the first could be remedied by using a language which allows partial assignment assignment to an array in conjunction with runtime hardware to detect duplicate definition (perhaps with I-structures [5]). The second is once again in the realm of expert programmer. The third could be done by a mechanical optimizer.

We see that the principle we have chosen to investigate - dataflow - although manifested differently at the macro and micro levels, is nontheless invaluable as a means of structuring an algorithm. In order to design the task system for the HEP version of the program we first observed the inherent data dependencies of the problem (Figure 1), and then used those data dependencies to generate a structured synchronization mechanism among the tasks. We voluntarily observed the discipline that each variable x(i) had one producer and many consumers, and used the scoreboarded memory resource to implement this discipline. This dataflow usage of the shared variables made the program more easily understandable and better organized. At the instruction level, we were also guided by the flow of data to operators to arrive at a correct and efficient algorithm. Thus, the notion of data flow, that data dependencies and only data dependencies should govern a computation sequencing is a valuable tool by which to structure a concurrent algorithm.

## 5. Acknowledgements

I would like to thank Dr. Swarn P. Kumar for her assistance in the HEP portion of this work; the Data Flow Group at Manchester University for providing the emulation systems; and ICASE for the use of their facilities for this project.

## 6. References

[ 1] Gurd & Watson, "Data Driven System for High Speed Parallel Computing", Computer Design, 6,7 1980.

[ 2] Lord, Kowlik and Kumar, "Solving Linear Algebraic Equations on an MIND Computer", JACM 83, v. 30 #1, pp. 103-117.

[ 3] Bowen, D.L., "Implementation of Data Structures on a Data Flow Computer", Ph.D. thesis, Manchester U., 1981.

[ 4] Ackerman, W., "Data Flow Languages", IEEE Computer, Feb. 1982.

[ 5] Arvind and Thomas, "I-Structures: An Efficient Data Type for Functional Languages", MIT TR 178, 1980.

# Perfect Graphs and Parallel Algorithms

David Helmbold and Ernst Mayr

Department of Computer Science
Stanford University
Stanford, California 94305

## 1 Introduction

This paper presents several parallel algorithms for graph problems, in particular for perfect graphs. Our main result is a deterministic $\mathcal{NC}$ algorithm for solving the two processor scheduling problem, answering an important open problem posed in [VV85]. We also present an $\mathcal{NC}$ algorithm for transitively orienting comparability graphs. By combining these two results, we obtain an $\mathcal{NC}$ algorithm for the matching problem on co-comparability graphs (the complements of comparability graphs) and nearly co-comparability graphs. In addition our transitive orientation algorithm gives us $\mathcal{NC}$ algorithms for several additional problems, such as identifying permutation graphs and finding the maximum weighted clique and optimal colorings in comparability graphs. Comparability, co-comparability, and permutation graphs are all subclasses of perfect graphs.

Scheduling problems have a long history and extensive literature. The most fundamental scheduling problems involve unit time execution tasks with precedence constraints restricting the order of execution. When the number of processors varies, the problem is $\mathcal{NP}$-complete [Ull75]. There are no published polynomial time algorithms for a fixed number of processors greater than two. The first polynomial time algorithm for the two processor case was published in 1969 [FKN69]. Faster algorithms were given by Coffman and Graham [CG72] in 1972, and a decade later, Gabow [Gab82,GT83] found an asymptotically optimal algorithm. Recently, Vazirani and Vazirani have published a randomized parallel solution [VV85]. Like Fujii et. al. they use the connection between matching and two processor scheduling, so their algorithm relies on an $\mathcal{RNC}$ matching subroutine such as [KUW85b] or [MVV].

In contrast, our scheduling algorithm [HM86b] is deterministic and does not require the aid of a matching subroutine. Therefore we are able to exploit the relationship between matching and two processor scheduling in the other direction, obtaining a deterministic parallel matching algorithm for co-comparability graphs.

The only ingredient required to convert our scheduling algorithm into a matching result is an $\mathcal{NC}$ transitive orientation subroutine. This routine takes an undirected graph and directs the edges so that the resulting digraph is transitively closed. Those graphs which can be transitively oriented are called comparability graphs. The complements of comparability graphs are co-comparability graphs. Kozen, Vazirani and Vazirani, in independent work, coupled a transitive orientation routine with our two processor scheduling algorithm to achieve an $\mathcal{NC}$ matching algorithm on co-comparability graphs [KVV85]. Our transitive orientation subroutine is also the key element in our algorithms for testing for permutation graphs and finding maximum weighted cliques and optimal colorings on comparability graphs.

## 2 Main Theorems and Applications

In this section we give some definitions, state our main results, and prove several important consequences. We use $[a, b]$ to denoted a directed arc and $(a, b)$ to denote an undirected edge between vertices $a$ and $b$.

A *perfect graph* is an undirected graph where the chromatic number and maximum clique size of every induced subgraph coincide. A *precedence graph* is an acyclic, transitively closed digraph. Thus if arcs $[a, b]$ and $[b, c]$ are in a precedence graph, then so is the arc $[a, c]$. A *comparability graph* is an undirected graph with the property that every edge can be assigned a direction such that the resulting graph is a precedence graph. The complement of a comparability graph is a *co-comparability* graph. Precedence graphs are equivalent to partial orders. Some graphs, such as a simple three-cycle, are both comparability and co-comparability graphs.

The undirected graph $G = (V, E)$ is a *permutation graph* if there exists a pair of permutations on the vertices such the edge $(v, v') \in E$ if and only if $v$ preceeds $v'$ (or $v'$ preceeds $v$) in both permutations. Permutation graphs are equivalent to the comparability graphs of partial orders with dimension two. A graph is both a comparability graph and a co-comparability graph if and only if it is a permutation graph [PLE71]. Permutation graphs, comparability graphs and co-comparability graphs are all subclasses of perfect graphs [Gol80].

This work was supported by in part by a grant from the AT&T Foundation, ONR contract N00014-85-C-0731, and NSF grant DCR-8351757.

An instance of the two processor scheduling problem is given by a precedence graph $\vec{G} = (V, \vec{E})$. Each vertex represents a task whose execution requires unit time on either of two identical processors. If there is a directed edge from task $t$ to task $t'$, then task $t$ must be completed before task $t'$ can be started. A schedule is a mapping from tasks to timesteps such that at most two tasks are mapped to each timestep and for all tasks $t$ and $t'$ if $t$ must preceed $t'$ ($t \prec t'$) then $t$ is mapped to an earlier timestep than $t'$. The length of a schedule is the number of timesteps used. An optimal schedule is one of shortest length.

The maximum matching problem on co-comparability graphs and the two processor scheduling problem are closely related. If $G$ is a co-comparability graph and $\vec{\bar{G}}$ is a transitive orientation of $G$'s complement, then the pairs of tasks mapped to the same timestep in an optimal two processor schedule of $\vec{\bar{G}}$ correspond to a maximum matching in $G$. Furthermore, there is a sequential algorithm for converting any maximum matching for $G$ into an optimal two processor schedule for $\vec{\bar{G}}$ [FKN69]. In [VV85] it was conjectured that this process is inherently sequential, but with our two processor scheduling algorithm it can be solved quickly in parallel.

**Theorem 1:** *Two processor scheduling is in $\mathcal{NC}$.*

**Proof:** We outline an $O(\log^2 n)$ time algorithm in section 3. Further details can be found in [HM86b]. □

**Theorem 2:** *There is an $\mathcal{NC}$ algorithm which detects if an undirected graph is transitively orientable, and if so finds a transitive orientation.*

**Proof:** We present such an algorithm in section 4. See also [KVV85]. □

**Corollary 2.1:** *There is an $\mathcal{NC}$ algorithm which detects whether or not a graph is a permutation graph.*

**Proof:** Graph $G$ is a permutation graph if and only if both $G$ and $\bar{G}$ are comparability graphs [PLE71]. Therefore, by running our transitive orientation algorithm on both $G$ and $\bar{G}$, we can determine if $G$ is a permutation graph. □

**Corollary 2.2:** *There is an $\mathcal{NC}$ algorithm which finds a maximum node-weighted clique in comparability graphs.*

**Proof:** Given a comparability graph $G$, we find a transitive orientation, $\vec{G}$. Examine any $k$-path in $\vec{G}$. Because $\vec{G}$ is transitively closed, the nodes on the $k$-path form a $k$-clique in $G$. Similarly, every $k$ clique in $G$ is a $k$-path in $\vec{G}$. Thus the problem of finding a maximum node-weighted clique in $G$ reduces to finding a maximum weight path in $\vec{G}$. Since $\vec{G}$ is a DAG, standard parallel techniques (i.e. transitive closure) can be used to find a heaviest path in $G$. □

**Corollary 2.3:** *There is an $\mathcal{NC}$ algorithm which finds a minimal node-coloring of comparability graphs.*

**Proof:** Given a comparability graph $G$, we find a transitive orientation, $\vec{G}$. We say that a vertex $v$ is on level $i$ in $\vec{G}$ if the longest (directed) path from $v$ to a sink contains exactly $i$ vertices. Clearly any pair of nodes on the same level are not adjacent in $G$, so they can be assigned the same color. Every node on level $i > 1$ is a predecessor of at least one node on level $i - 1$. Therefore, if $\vec{G}$ has $k$ levels then $\vec{G}$ has a path of length $k$ and $G$ has a $k$-clique. Since no coloring can use less colors than the size of the largest clique, this yields an optimal coloring. □

**Theorem 3:** *There is an $\mathcal{NC}$ algorithm for finding maximum matchings on co-comparability graphs.*

**Proof:** One such algorithm is given in section 5. □

This theorem is extended to nearly co-comparability graphs in section 5.

**Corollary 3.1:** *Maximum matchings for permutation graphs and partial orders of dimension 2 can be found in $\mathcal{NC}$.*

**Proof:** As stated above, these graphs are co-comparability graphs. □

**Corollary 3.2:** *There is an $\mathcal{NC}$ algorithm which finds maximum matchings on interval graphs.*

**Proof:** Interval graphs are a subclass of co-comparability graphs [GH64]. □

## 3 Two Processor Scheduling

The scheduling algorithm is built around a routine that, for any precedence graph, computes the length of the graph's optimal schedule(s). This length routine is applied repeatedly in order to find an actual optimal schedule for the input graph.

Let $G = (V, \prec)$ be the precedence graph we are interested in. If $t \prec t'$ then $t$ is a predecessor of $t'$ and $t'$ is a successor of $t$. For any pair of tasks, $t, t' \in V$, define $V^t_{t'}$ to be the set of tasks which are both successors of $t$ and predecessors of $t'$ and $G^t_{t'}$ to be the subgraph of $G$ induced by $V^t_{t'}$. The *schedule distance* between tasks $t$ and $t'$, $SD(t, t')$, is the length of an optimal schedule for $G^t_{t'}$. If $t \not\prec t'$ then $SD(t, t') = 0$.

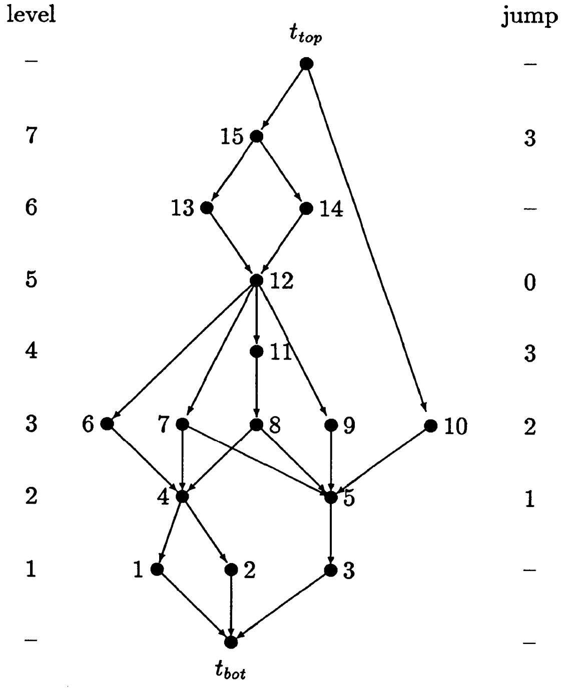


Figure 1: This is a precedence graph containing fifteen tasks (transitive arcs have been omitted). The special tasks $t_{top}$ and $t_{bot}$ are added when computing the length of optimal schedules for $G$. The levels of the original graph are on the left and the jump sequence is on the right.

**Lemma 3.1:** *Let $S$ be a set of tasks such that for all $\hat{t} \in S$:*

**i.** $t \prec \hat{t} \prec t'$;

**ii.** $\mathrm{SD}(t, \hat{t}) \geq k$; *and*

**iii.** $\mathrm{SD}(\hat{t}, t') \geq l$.

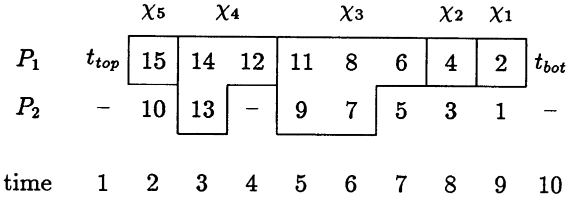
*Then* $\mathrm{SD}(t, t') \geq k + l + |S|/2$.

**Proof:** Count the number of timesteps required to schedule those tasks between $t$ and $t'$. There must be at least $k$ timesteps before the first task in $S$ is scheduled. It takes at least $|S|/2$ timesteps to complete the tasks in $S$. After the last task in $S$ has been completed, at least $l$ additional timesteps are required. Therefore $SD(t, t') \geq k + l + |S|/2$. □

The distance algorithm (see figure 2) uses a method like transitive closure to compute the schedule distance between all pairs of tasks in a precedence graph $G = (V, A)$. It initially guesses that the scheduling distance between each pair of tasks is zero. By repeatedly applying lemma 3.1 to each pair of tasks in parallel the algorithm refines its guesses. Below we prove that after $\log |V|$ iterations, the algorithm's guess for each pair of tasks has converged to schedule distance. The distance algorithm has a straightforward implementation on an $n^5$ processor P-RAM taking $O(\log^2 n)$ time.

```
$d_0(\star, \star) := 0$;
for $i := 1$ to $\lceil \log n \rceil$ do
    for all $t, t'$ with $t \prec t'$ do in parallel
        for all $0 \leq k, l < n - 1$ do in parallel
            $S_{t,t',k,l} := \{s : t \prec s \prec t',$
                        $d_{i-1}(t, s) \geq k,$
                        $d_{i-1}(s, t') \geq l\}$;
            $d_i(t, t') := \max_{S_{t,t',k,l} \neq \emptyset} \{d_{i-1}(t, t'),$
                        $k + l + \lceil |S_{t,t',k,l}|/2 \rceil\}$;
$SD(\star, \star) := d_{\lceil \log n \rceil}(\star, \star)$
```

Figure 2: The distance algorithm.



|  |  | $\chi_5$ | $\chi_4$ |  | $\chi_3$ |  |  | $\chi_2$ | $\chi_1$ |  |
|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | $t_{top}$ | 15 | 14 | 12 | 11 | 8 | 6 | 4 | 2 | $t_{bot}$ |
| $P_2$ | – | 10 | 13 | – | 9 | 7 | 5 | 3 | 1 | – |
| time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Figure 3: This is a lexicographically maximal jump schedule for the graph in figure 1. Each of the sets $\chi_i$ is boxed.

**Lemma 3.2:** *The distance algorithm always computes the schedule distance between every pair of tasks.*

**Proof:** Lemma 3.1 guarantees that the distances computed by the algorithm are never greater than the the schedule distances.

In [CG72] it is shown how to construct sets of tasks $\chi_0, \chi_1, \ldots, \chi_k$ for any precedence graph such that:

- Those tasks in any $\chi_i$ are predecessors of all tasks in $\chi_{i-1}$, and
- The length of optimal schedules for $G$ is $\sum_i \lceil |\chi_i|/2 \rceil$ (See figure 3).

Our algorithm doesn't compute these $\chi$s, we simply use their existence to prove that the distances the algorithm does compute converge to the scheduling distance.

Examine how the distance algorithm computes the schedule distance between an arbitrary pair of tasks, $t$ and $t'$. Let $\chi_1, \chi_2, \ldots, \chi_k$ be a set of $\chi$s for $G^t_{t'}$, $\chi_0 = \{t\}$, and $\chi_{k+1} = \{t'\}$. After the first iteration of the outer loop, the distance computed between any task in $\chi_i$ and one in $\chi_{i+2}$ is at least $\lceil |\chi_{i+1}|/2 \rceil$. After the second iteration, the distance computed between any task in $\chi_i$ and any task in $\chi_{i+4}$ is at least $\lceil \chi_{i+1}/2 \rceil + \lceil \chi_{i+2}/2 \rceil + \lceil \chi_{i+3}/2 \rceil$ ($S = \chi_2$, $k = \lceil \chi_{i+1}/2 \rceil$, $l = \lceil \chi_{i+1}/2 \rceil$). Each iteration we double the number of $\chi$s accounted for.

After $\log k$ iterations, the computed distance between $t$ and $t'$ is at least the optimal schedule length for $G^t_{t'}$, and thus at least $SD(t, t')$.

Since $G$ contains $n$ tasks, each $G^t_{t'}$ has at most $n - 2$ $\chi$s. Therefore, after $\lceil \log n \rceil$ iterations the algorithm computes the schedule distance for each pair of tasks. □

The distance algorithm can be used to compute the length of optimal schedules for a graph. Augment the graph with two dummy tasks, $t_{\text{top}}$ and $t_{\text{bot}}$, which are a predecessor and successor (respectively) of all other tasks in $G$. Now $SD(t_{\text{top}}, t_{\text{bot}})$ is the length of $G$'s optimal schedules, and can be found using the distance algorithm.

The method for converting the distance algorithm into one which finds an optimal schedule involves several constructions. For the sake of brevity this paper contains only an outline of our method. Interested readers should consult [HM86b] for a more detailed presentation.

The search for an optimal schedule can be restricted to the class of *Lexicographically Maximal Jump* (LMJ) schedules. Each task $t$ in the precedence graph is assigned a *level* equal to the number of tasks in the longest path from $t$ to a sink. *A level schedule* gives preference to tasks on higher levels. More precisely, suppose levels $L, \ldots, l + 1$ have already been scheduled and there are $k$ unscheduled tasks remaining on level $l$. If $k$ is even level schedules pair the $k$ tasks with each other and there is no *jump* from level $l$. If $k$ is odd, a level schedule pairs $k-1$ of the tasks with each other and the remaining task $t$ is paired with a task from a lower level $l' < l$. In this case, level $l$ *jumps* to level $l'$. We assume that there are an unlimited number of dummy tasks on level 0 which can be paired with any other tasks. The *jump sequence* of a level schedule is the levels jumped *to*, listed in decreasing order by level jumped *from* (see figure 1). The *Lexicographically Maximum Jump (LMJ)* sequence is the jump sequence (resulting from some level schedule) that is lexicographically greater than any other jump sequence resulting from a level schedule. An *LMJ schedule* is a level schedule whose jump sequence is the LMJ sequence. Note that our definition of LMJ is similar to the definitions of highest level first in [Gab82] and [VV85]. The following theorem establishes the importance of LMJ schedules.

**Theorem 4:** [Gab82] *Every LMJ schedule is optimal.* □

Our two processor algorithm uses the distance algorithm to find the LMJ sequence and which jump (if any) a pair of tasks can be used for. In general, there will be many possible pairs for each jump. A path doubling computation finds a consistent set of task pairs for the jumps. The remaining tasks are paired up within levels. Since there are never precedence constraints between two tasks on the same level, this pairing can be done arbitrarily. An LMJ schedule is obtained by sorting the resulting set of task pairs (both for jumps and within levels).

## 4 Transitive Orientation

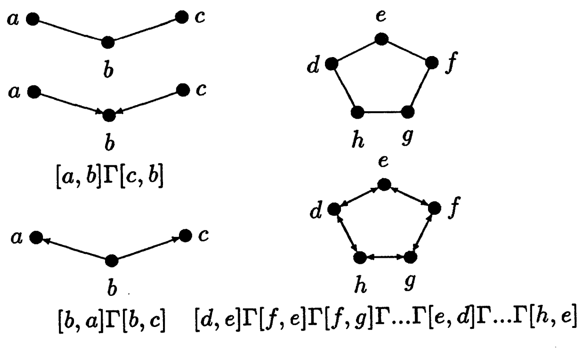
The transitive orientation problem is nontrivial because some edges cannot be oriented independently. If the edges $(a, b)$ and $(b, c)$ are in the graph to be oriented, but the edge $(a, c)$ is not, then the edges $(a, b), (b, c)$ cannot be oriented independently. If we choose the arc $[a, b]$ then we are forced to include the arc $[b, c]$ in the transitive orientation (see figure 4). The binary relation $\Gamma$ reflects this simple kind of forcing. [PLE71]. Given $G = (V, E)$, we say that $[a, b]\Gamma[a', b']$ if $(a, b) \in E$; $(a', b') \in E$ and either $a = a'$ and $(b, b') \notin E$ or $b = b'$ and $(a, a') \notin E$. Thus $[a, b]\Gamma[a, b]$ and if $[a, b]\Gamma[c, b]$ then $[b, a]\Gamma[b, c]$, however $[a, b] \not\Gamma [b, a]$.

The reflexive, transitive closure of $\Gamma$, $\Gamma^*$, is an equivalence relation on the possible orientations of edges in $E$. For obvious reasons, we call these equivalence classes *implication classes.* If $A$ is a set of arcs (e.g. an implication class) then $A^*$ denotes the set of undirected edges $\{(a, b) : [a, b] \in A \vee [b, a] \in A\}$, and $A^{-1}$ is the set of arcs $\{[b, a] : [a, b] \in A\}$. A set of arcs $A$ is *consistent* if $A \cap A^{-1} = \emptyset$, and is *inconsistent* when $A \cap A^{-1} \neq \emptyset$.

Implication classes and $\Gamma$-decompositions have been studied by M.C. Golumbic. Many of the lemmas in this section also appear in [Gol77] or [Gol80].

**Lemma 4.1:** *If $A \neq B$ are implication classes of $G$ then either $A = B^{-1}$ or $A^* \cap B^* = \emptyset$.*

**Proof:** Assume that $(a, b) \in A^* \cap B^*$. Without loss of generality, let $[a, b] \in A$. If $[a, b] \in B$ then $B = A$ since implication classes are equivalence classes. Therefore $[b, a] \in B$, and $[b, a] \notin A$. By definition, if $[a, b]\Gamma[a', b']$ then $[b, a]\Gamma[b', a']$. Thus some $[c, d]\Gamma^*[a, b]$ if and only if $[d, c]\Gamma^*[b, a]$, so $A = B^{-1}$. □



$[a, b]\Gamma[c, b]$

$[b, a]\Gamma[b, c]$ $[d, e]\Gamma[f, e]\Gamma[f, g]\Gamma \ldots \Gamma[e, d]\Gamma \ldots \Gamma[h, e]$

Figure 4: Graphs and Implication Classes

Given an undirected graph $G_1 = (V, E)$ pick any implication class $B_1$ and delete it, forming $G_2 = (V, E - B_1^*)$. Next form $G_3$ by removing some implication class $B_2$ from $G_2$. Continue the process until removing $B_k$ from $G_k$ results in a graph with no edges. The sequence of implication classes removed, $B_1, B_2, ..., B_k$, is called a $\Gamma$-*decomposition* of $G$. The following theorem points out the usefulness of $\Gamma$-decompositions.

**Theorem 5:** *(TRO Theorem [Gol80]) Let $B_1, B_2, ..., B_k$ be a $\Gamma$-decomposition of an undirected graph $G$. The following statements are equivalent:*

**i.** *$G$ is a comparability graph.*

**ii.** *Every implication class of $G$ is consistent.*

**iii.** *Each $B_i$ in the $\Gamma$-decomposition is consistent.*

*Furthermore, when these conditions hold the union of the $B_i$s is a transitive orientation of $G$.*

**Proof:** The proof of this theorem requires several technical lemmas, and thus is beyond the scope of this paper. The interested reader is referred to [Gol77,Gol80]. □

The TRO theorem suggests a sequential algorithm for transitively orienting comparability graphs. One can take any edge, orient it arbitrarily, find the associated implication class, add the implication class to the transitive orientation and remove it from the comparability graph. Repeating this procedure yields a $\Gamma$-decomposition of the comparability graph and therefore a transitive orientation. This is essentially the algorithm in [PLE71].

In order to parallelize this algorithm it is neccessary to understand how implication classes change during a $\Gamma$-decomposition. We will see below that the changes are very simple: implication classes are either merged with other implication classes or remain unchanged.

**Lemma 4.2:** *Let $A$ and $B$ ($A^* \neq B^*$) be implication classes of $G = (V, E)$. Then all of the arcs in $A$ are in the same implication class of $G' = (V, E - B^*)$.*

**Corollary 4.2.1:** *Let $B$ be an implication class of $G = (V, E)$. Every implication class of $G' = (V, E - B^*)$ is the union of implication classes of $G$.*

**Proof:** A $\Gamma$ relationship between two arcs is lost only if one of the arcs is deleted or a triangular edge added. Removing $B^*$ certainly doesn't add any triangular edges, and by Lemma 4.1 $A^* \cap B^* = \emptyset$, so all edges of $A^*$ remain in $G'$. □

Three edges, $(a, b), (a, c), (b, c)$ in an undirected graph $G$ form a *tricolored* triangle if $G$ has three implication classes, $A$, $B$, and $C$ such that $A^* \neq B^* \neq C^* \neq A^*$ and $(a, b) \in A^*, (b, c) \in B^*$, and $(a, c) \in C^*$. We say that $A$ and $B$ are *triangle related*, written $A \triangle B$, if either $A = B^{-1}$ or there is a tricolored triangle in $G$ with one edge in $A^*$ and another edge in $B^*$.

**Lemma 4.3:** *Let $G = (V, E)$ be an undirected graph with implication classes $A$ and $B$, $A^* \neq B^*$. $A$ is not an implication class of $G' = (V, E - B^*) \Longleftrightarrow A \triangle B$.*

**Proof:** ($\Leftarrow$) Since $A \triangle B$, $G$ contains a tricolored triangle $(b, c) \in A^*, (a, c) \in B^*, (a, b) \in C^*$. Without loss of generality, let $[b, c] \in A$ and $C \neq A$ be the equivalence class containing $[b, a]$. The edge $(a, c)$ is not in $E - B^*$, so $[b, c] \Gamma [b, a]$ in $G'$. Since $A$ is not closed under the $\Gamma$ relation, it can not be an implication class of $G'$.

($\Rightarrow$) By Lemma 4.2 $A$ is a proper subset of some implication class for $G'$. Therefore in $G'$ and arc $[b, c] \in A$ is $\Gamma$ related to an arc $[b, a] \notin A$ (the case where $[b, c] \Gamma [a, c]$ is analogous). Let $C$ be the implication class of $G$ containing $[b, a]$ ($A \neq C \neq B$). Since $[b, c] \Gamma [b, a]$ in $G'$ but not in $G$, the triangular edge $(b, c)$ must be in $B^*$. Thus $(b, c) \in A^*, (a, b) \in C^*, (a, c) \in B^*$ form a tricolored triangle in $G$ and $A \triangle B$. □

**Lemma 4.4:** *Let the implication classes $B_1, ..., B_k$ of $G = (V, E)$ be an independent set under the $\triangle$ relation. Then in $G' = (V, E - B_1^*)$, $\{B_2, ..., B_k\}$ is an independent set (under $\triangle$) of implication classes.*

**Corollary 4.4.1:** *If implication classes $B_1, ..., B_k$ of $G$ form an independent set under the $\triangle$ relation, then $B_1, ..., B_k$ are the first $k$ implication classes in a $\Gamma$ decomposition of $G$.*

**Proof:** By Lemma 4.3, $B_2, ..., B_k$ are all implication classes of $G'$. Assume to the contrary that $G'$ contains a tricolored triangle $(a, b) \in B_i^*, (a, c) \in B_j^*, (b, c) \in A^*$. The arc $[b, c]$ belongs to some implication class $C$ of $G$. By Lemma 4.2, $B_i^* \neq C^* \neq B_j^*$. Therefore, $(a, b), (a, c), (b, c)$ is a tricolored triangle in $G$ and $B_i \triangle B_j$ – contradiction. □

**Lemma 4.5:** *Let $B_1, B_2, ..., B_k$ be a maximal independent set under the $\triangle$ relation for some graph $G_1 = (V, E)$. Every implication class of $G_{k+1} = (V, E - B_1^* - B_2^* - ... - B_k^*)$ is the union of at least two implication classes of $G$.*

**Corollary 4.5.1:** *The number of implication classes for $G$ is at least twice the number of implication classes for $G_{k+1}$.*

**Proof:** Every other implication class of $G$ is $\triangle$ related to some $B_i$. Therefore, by Lemma 4.3 every implication class of $G$ is merged with at least one other implication class of $G$ during the formation of $G_{k+1}$.

Since implication classes are never split (Lemma 4.2), every implication class of $G_{k+1}$ must be the union of at least two implication classes of $G$. □

The input to our algorithm is an undirected graph $G_1 = (V, E)$. Our algorithm's output is either $\vec{G}$, a transitive orientation of $G_1$, or an indication that $G_1$ has no transitive orientation. The algorithm proceeds in several iterations. Graph $\vec{G_1}$ is initialized to $(V, \emptyset)$. During iteration $i$ the algorithm determines a maximal independent set of implication classes for $G_i$. These implication classes are deleted from $G_i$ forming $G_{i+1}$, and added to $\vec{G_i}$ forming $\vec{G}_{i+1}$ From Lemma 4.5 the number of implication classes in $G$ is halved each iteration. Therefore, after $\log n$ iterations $G$ will contain no edges and $\vec{G}$ will be a transitive orientation of the original graph.

Each iteration consists of the following four steps:

1. Determine the implication classes of $G_i$. This can be done using standard parallel techniques such as solving two-SAT formulae or finding connected components [SV82].
2. Determine the $\triangle$ relation on implication classes.
3. Use a maximal independent set subroutine, such as [KW84,Lub85], to obtain a maximal independent set, $M$ of implication classes.
4. In parallel, for each of the $A_j$s in $M$, delete $A_j^*$ from $G_i$, and add $A_j$ to $\vec{G_i}$.

Step 3 is the most expensive of these steps, requiring $O(\log^2 n)$ time and $n^4$ processors. The $\log n$ iterations can therefore be done in $O(\log^3 n)$ time on $n^4$ processors.

## 5 Maximum Matching

The two processor scheduling and transitive orientation algorithms can be used to find maximum matchings on co-comparability graphs. To find a maximum matching on the co-comparability graph $G = (V, E)$, first create the comparability graph $\bar{G} = (V, \{(a, b) : (a, b) \notin E\})$. Applying the transitive orientation routine converts $\bar{G}$ into a precedence graph. An optimal two processor scheduled can be found for the precedence graph using our scheduling algorithm. We will see below that the pairs of tasks scheduled together form a maximum matching of $G$.

Let $S$ be any optimal two processor schedule for $\overset{\rightrightarrows}{G}$. A *task-pair* of $S$ is a pair of tasks mapped to the same timestep by $S$. Since there are no precedence relationships between tasks in a task-pair and each task is mapped to a single timestep, the set of task-pairs of $S$ form a matching in $G$. Because $S$ is an optimal schedule, no schedule has more task-pairs.

A task is *available* at some point in a schedule if it can be executed without violating the precedence constraints.

**Lemma 5.1:** *If co-comparability graph $G$ has a perfect matching then $\overset{\rightrightarrows}{G}$ has a schedule where every task is in a task-pair.*

**Proof:** We say a pair of tasks is *mated* if the pair is in the perfect matching. Construct the schedule (and modify the "mated" relationship) iteratively as follows:

> If two mated tasks are both available, schedule one such mated pair. Otherwise find two mated pairs, $(t, t')$ and $(s, s')$, such that $t$ and $s$ are available and there is no precedence relationship between $t'$ and $s'$. Schedule $t$ with $s$ and mate $t'$ with $s'$.

Note that there are never precedence constraints between a pair of mated tasks. This method clearly takes two tasks each timestep and does not violate the precedence constraints. What we must show is that it always constructs a schedule for $\overset{\rightrightarrows}{G}$.

Assume to the contrary that at some point it does not find a pair of tasks to schedule. Let $U$ be the set of available tasks and $U'$ be the set of tasks which are mated to tasks in $U$. Since the method fails, $U \cap U' = \emptyset$ and there is a precedence relationship between every pair of tasks in $U'$ (i.e. $U'$ is totally ordered). Let $t'$ be the task in $U'$ which preceeds all other tasks in $U'$. Since $t' \notin U$, there must be some $t \in U$ such that $t \prec t'$. However, by the transitivity of precedence, $t$ also preceeds its mate – contradiction. □

**Lemma 5.2:** *Let $\overset{\rightrightarrows}{G} = (V, \prec)$ be a precedence graph and $S$ a two processor schedule for $\overset{\rightrightarrows}{G}' = (V - \{t\}, \prec)$. A single timestep containing $t$ can be inserted into $S$ yielding a schedule for $\overset{\rightrightarrows}{G}$.*

**Proof:** Let $t'$ be the last predecessor of $t$ in $S$. Insert task $t$ immediately after the timestep containing $t'$. Obviously there are no precedence conflicts between $t$ and its predecessors. Since $S$ is valid schedule, there are no precedence conflicts between tasks in $V - \{t\}$. Therefore any precedence conflict which is violated is of the form $t \prec \hat{t}$. By transitivity $t'$ also preceeds $\hat{t}$, so $\hat{t}$ comes strictly after $t'$ in $S$. Since $t$ is inserted immediately after $t'$, task $t$ appears before $\hat{t}$ in the modified schedule. □

Let $M$ be the tasks in a maximum matching on $G$. The above Lemmas suggest a way to obtain a schedule, $S$, for $\overset{\rightrightarrows}{G} = (V, \prec)$ where the paired tasks of $S$ are precisely the tasks in $M$. Start by finding an optimal schedule, $S'$ for the subgraph of $\overset{\rightrightarrows}{G}$ induced by $M$ and add the tasks in $V - M$ one at a time. One $\mathcal{NC}$ implementation of this algorithm involves bucket sorting the tasks in $V - M$ based on which task-pair of $S'$ they follow. By topologically sorting the tasks within each bucket we can quickly determine where each task should be inserted.

**Theorem 6:** *The task-pairs of any optimal schedule for $\vec{\vec{G}}$ form a maximum matching on $G$.*

**Proof:** Let $M$ be the tasks in some maximum matching of $G$. Let $S$ be an optimal schedule for the subgraph of $\vec{\vec{G}}$ induced by $M$. By Lemma 5.1, the task-pairs of $S$ form a maximum matching on $G$. By Lemma 5.2 we can insert the other tasks of $\vec{G}$ one at a time without disturbing the task-pairs. Therefore, the task-pairs of the resulting schedule for $\vec{\vec{G}}$ form a maximum matching on $G$. Since every optimal schedule has the same number of task-pairs and the task-pairs of every schedule form a matching, the task-pairs of any optimal schedule for $\vec{\vec{G}}$ forms a maximum matching on $G$. □

If $\bar{G}$ is not transitively orientable it may still be possible to find a maximum matching in $G = (V, E)$. Assume we are given a set $U$, consisting of $O(\log n)$ edges, such that $\bar{G} \cup U$ is transitively orientable. The following method finds a maximum matching in $G$.

For each $S' \in 2^S$ such that $S'$ is a matching find (in parallel) a maximum matching in $G' = (V - \{v : (v, v') \in S'\}, E - S$. A maximum matching for $G$ occurs whenever the cardinality of the maximum matching for $G'$ plus $|S'|$ is maximal.

A graph $G$ is a $k$-nearly comparability graph when:

— $G$ has at most $k \log n$ inconsistent implication classes and

— each inconsistent implication class of $G$ is split into consistent implication classes by the addition of at most $k$ edges.

A $k$-nearly co-comparability graph is the compliment of a $k$-nearly comparability graph.

Let $G$ be a $k$-nearly co-comparability graph (for some constant $k$). The following is an outline of an $\mathcal{NC}$ algorithm for finding a maximum matching in $G$. In parallel examine each set, $T$, of $k$ edges not in $\bar{G}$. Determine which inconsistent implication classes are split when $T$ is added to $\bar{G}$. For each inconsistent implication class $A$, pick any set of $k$ edges which splits $A$ into consistent implication classes. At most $k^2 logn$ edges are picked, so the above method can be used to find a maximum matching for $G$.

## 6 Conclusions

Although the algebraic approach was used to obtain the first parallel matching algorithms [KUW85b,MVV], these are randomized algorithms. It is interesting that we can obtain deterministic matching algorithms for wide classes of graphs using a purely combinatorial approach. Perhaps the combinatorial approach will yield deterministic algorithms for matching on other classes of graphs as well.

It was surprising how much more difficult computing the actual schedule was than simply computing its length. In higher complexity classes such as $\mathcal{P}$ and $\mathcal{NP}$ it is often easy to go from the decision problem to computing an actual solution, because of self reducibility. However this does not necessarily seem to be the case for parallel complexity classes. To support this observation we note that the random $\mathcal{NC}$ algorithm for finding the cardinality of a maximum matching is much simpler than the random $\mathcal{NC}$ algorithm for determining an actual maximum matching [KUW85a].

There are several open problems related to scheduling. We are attempting to extend our two processor result to the case when the tasks have nonuniform start times and/or deadlines. When the precedence constraints are restricted to in-trees or out-trees there are parallel algorithms for generating schedules on an arbitrary number of processor [DUW84,HM86a]. It is an open problem whether interval-ordered tasks [PY79] can be scheduled in parallel.

One variant of the two processor problem that we know to be $\mathcal{NP}$-complete (by reduction from the clique problem) allows incompatibility edges as well well as precedence constraints. When there is an incompatibility constraint between two tasks they can be executed in either order, but not concurrently. Incompatibility constraints arise naturally when two or more tasks need the same resource, such as special purpose hardware or a database file.

## References

[CG72] E.G. Jr. Coffman and R.L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[DUW84] D. Dolev, E. Upfal, and M. Warmuth. Scheduling trees in parallel. In *Proc. International Workshop on Parallel Computing and VLSI*, pages 1–30, Amalfi, 1984.

[FKN69] M. Fujii, T. Kasami, and K. Ninamiya. Optimal sequencing of two equivalent processors. *SIAM J. Appl. Math.*, 17(4):784–789, 1969.

[Gab82] H.N. Gabow. An almost-linear algorithm for two-processor scheduling. *JACM*, 29(3):766–780, 1982.

[GH64] P.C. Gilmore and A.J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canad. J. Math*, 16, 1964.

[Gol77] M.C. Golumbic. Comparability graphs and a new matroid. *J. Combinatorial Theory (B)*, 22(1):68–90, 1977.

[Gol80] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[GT83] H.N. Gabow and R.E. Tarjan. A linear time algorithm for special case of disjoint set union. In *Proc. 15th STOC*, 1983.

[HM86a] David Helmbold and Ernst Mayr. Fast scheduling algorithms on parallel computers. *Advances in Computing Research*, 1986. to appear.

[HM86b] David Helmbold and Ernst Mayr. Two processor scheduling is in $\mathcal{NC}$. In *Proceedings of the 1986 Agean Workshop on Computing: VLSI Algorithms and Architectures*, July 1986.

[KUW85a] R.M. Karp, E. Upfal, and A. Wigderson. Are search and decision problems computationally equivalent? In *Proc. 17th STOC*, 1985.

[KUW85b] R.M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. In *Proc. 17th STOC*, 1985.

[KVV85] D. Kozen, U.V. Vazirani, and V.V. Vazirani. Nc algorithms for comparability graphs, interval graphs, and testing for unique perfect matching. In *Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Dehli, 1985.

[KW84] R.M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proc. 16th STOC*, 1984.

[Lub85] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proc. 17th STOC*, 1985.

[MVV] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani. Parallel algorithms for rank and matching. private communication, Nov. 22, 1985.

[PLE71] A. Pnueli, A. Lempel, and S. Even. Transitive orientation of graphs and identification of permutation graphs. *Can. J. Math.*, 23(1):160–175, 1971.

[PY79] C.H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Computing*, 8(3), 1979.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–63, 1982.

[Ull75] J.D. Ullman. $\mathcal{NP}$-complete scheduling problems. *J. Comput. System Sci.*, 10(3):384–393, 1975.

[VV85] U.V. Vazirani and V.V. Vazirani. The two-processor scheduling problem is in $\mathcal{RNC}$. In *Proc. 17th STOC*, 1985.

# Communication-Efficient Parallel Graph Algorithms

Charles E. Leiserson
Bruce M. Maggs

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

*Abstract*—Communication bandwidth is a resource ignored by most parallel random-access machine (PRAM) models. This paper shows that many graph problems can be solved in parallel, not only with polylogarithmic performance, but with efficient communication at each step of the computation. We measure the communication requirements of an algorithm in a model called the *distributed random-access machine* (DRAM), in which communication cost is measured in terms of the congestion of memory accesses across cuts of an underlying network. The algorithms are based on a communication-efficient variant of the tree contraction technique due to Miller and Reif.

## 1. Introduction

Underlying any realization of a parallel random-access machine (PRAM) is a communication network that conveys information between processors and memory banks. Yet in most PRAM models, communication issues are largely ignored. The basic assumption in these models is that in unit time each processor can simultaneously access one memory location. For truly large parallel computers, however, computer engineers will be hard pressed to implement networks with the communication bandwidth demanded by this assumption. The difficulty of building such networks threatens the validity of the PRAM as a predictor of algorithmic performance. This paper introduces a more restricted PRAM model, which we call a *distributed random-access machine* (DRAM), to reflect an assumption of limited communication bandwidth in the underlying network.

We measure the cost of communication in a network in terms of the number of messages that must cross a cut of the network, as in [9] and [12]. Specifically, a *cut* $S = (A, \overline{A})$ of a network[1] is a partition of the network into two sets of processors $A$ and $\overline{A}$. The *capacity* $\text{cap}(S)$ is the number of wires connecting processors in $A$ with processors in $\overline{A}$, i.e., the bandwidth of communication between $A$ and $\overline{A}$. For a set $M$ of messages we define the *load* of $M$ on a cut $S = (A, \overline{A})$ to be the number of messages in $M$ between a processor in $A$ and a processor in $\overline{A}$. The *load factor* of $M$ on $S$ is

$$\lambda(M, S) = \frac{\text{load}(M, S)}{\text{cap}(S)},$$

and the *load factor* of $M$ on the entire network is

$$\lambda(M) = \max_S \lambda(M, S).$$

The load factor provides a simple lower bound on the time required to deliver a set of messages. For instance, if there are 10 messages to be sent across a cut of capacity 3, the time required to deliver all 10 messages is at least the load factor 10/3.

Networks in which the load-factor lower bound can be met to within a polylogarithmic factor as an upper bound include volume and area-universal networks, such as fat-trees [9,12] and meshes of trees [13]. The load factor can be also met to within a polylogarithmic factor as an upper bound by the standard universal routing networks, such as the Boolean hypercube, the butterfly (a.k.a. FFT, Omega), and the cube-connected cycles, but the lower bound is weak because every cut of these networks is large relative to the number of processors in the smaller side of the cut. Networks for which the load factor lower bound cannot in general be approached to within a polylogarithmic factor as an upper bound include linear arrays, meshes, and high-diameter networks in general.

Whereas communication is essentially free in PRAM models, the cost of communication in a DRAM depends on the locality of memory accesses as measured by the load factor of an underlying network. The DRAM is an attempt to abstract the essential communication characteristics of volume and area-universal networks without relying in detail on any particular network. Much as the PRAM can be viewed as an abstraction of a hypercube, in that algorithms for a PRAM can be implemented on a hypercube with only polylogarithmic performance degradation, the DRAM can be viewed as an abstraction of a volume or area-universal network. Fast, communication-efficient algorithms on a DRAM translate directly to fast, communication-efficient algorithms on, for example, a fat-tree.

This paper shows that many graph problems for a graph $G = (V, E)$ can be efficiently solved with $O(|E|)$ processors in the DRAM model. The algorithms we give apply to all of the popular PRAM models because a PRAM can be viewed as a DRAM in which communication costs are ignored. In fact, the algorithms we give can all be performed on an exclusive-read, exclusive-write DRAM, and when run on a PRAM, they are nearly as efficient in the PRAM model as corresponding concurrent-read, exclusive-write PRAM algorithms in the literature.

The remainder of this paper is organized as follows. Section 2 contains a specification of the DRAM model and the implementation of data structures in the model. The section demonstrates why the "recursive doubling" technique frequently used in parallel algorithms is inefficient in the DRAM model. It also defines the notion of a *conservative algorithm* as a concrete realization of a communication-efficient algorithm, and gives a "Shortcut Lemma" that forms the basis of the conservative algorithms in this paper. Section 3 presents a conservative "recursive pairing" technique that can be used to perform many of the same functions as recursive doubling. Section 4 presents a linear-space, conservative "tree contraction" algorithm based on the ideas of Miller and Reif [17]. Section 5 presents *treefix computations,* which are generalizations of the parallel prefix computation [3,7,18] to trees. We show that treefix computations can be performed using the tree contraction algorithm of Section 4. Section 6 gives short,

This research was supported in part by the Defense Advanced Research Projects Agency under Contract N00014-80-C-0622. Charles Leiserson is supported in part by an NSF Presidential Young Investigator Award. Bruce Maggs is supported in part by an NSF Fellowship.

[1] We assume that the communication network is an *interconnection network,* meaning that the processors are interconnected as a graph, and routing of messages is performed by the processors. The generalization to a *routing network,* where routing can be done by switches that are not processors, is straightforward, but complicates the definitions.

0190-3918/86/0000/0861 $01.00 © 1986 IEEE

efficient parallel algorithms for tree and graph problems, most of which are based on treefix computations. Section 7 contains some concluding remarks.

## 2. The DRAM model

This section presents the DRAM model. We show how a data structure can be embedded in a DRAM and define the load factor of a data structure. We demonstrate that many existing PRAM algorithms are not communication efficient in the DRAM model by examining the "recursive doubling" technique [21] used extensively by algorithms in the literature. We introduce the notion of a *conservative* algorithm as one in which the load factor of each set of memory accesses can be bounded above by the load factor of the input data structure. Our conservative algorithms are based on a simple lemma that shows how pointers in a data structure can be "shortcut" without increasing the load factor.

A DRAM consists of a set of $n$ processors. All memory in the DRAM is local to the processors, with each processor holding a small number of $O(\lg n)$-bit registers. A processor can read, write, and perform arithmetic and logical functions on values stored in its local memory. It can also read and write memory in other processors. (A processor can transfer information between two remote memory locations through the use of local temporaries.) Each set of memory accesses is performed in a memory access *step*, and any of the standard PRAM assumptions about simultaneous reads or writes can be made. Our algorithms use only mutually exclusive memory references, however, so these special cases never arise.

The essential difference between a DRAM and a PRAM is that the DRAM models communication costs. We presume remote memory accesses are implemented by routing messages through an underlying network. Each cut $S = (A, \overline{A})$ of the processors has an assigned capacity $\text{cap}(S)$. For a set $M$ of memory accesses, we define $\text{load}(M, S)$ to be the number of accesses in $M$ between a processor in $A$ and a processor in $\overline{A}$. The load factor of $M$ on $S$ is $\lambda(M,S) = \text{load}(M,S)/\text{cap}(S)$, and the load factor of $M$ on the entire network is $\lambda(M) = \max_S \lambda(M, S)$. The basic assumption in the DRAM model is that *the time required to perform a set $M$ of memory accesses is $\lambda(M)$.*

Because the load factor of a set of memory accesses is the time required to perform them, the embedding of a data structure within a DRAM affects the time required to perform operations on the data structure. A natural way to embed a data structure in a DRAM is to put one record of the data structure into each processor. The record can contain pointers to records in other processors, as well as auxilliary local storage. As an example of how the embedding of a data structure influences communication costs, consider an embedding of a list in which alternate list elements are placed on opposite sides of a narrow cut. If each element fetches a value from the next element in the list, the load factor across the cut is large. Thus, a set of memory accesses that theoretically takes unit time in the PRAM model can require considerably more time due to network congestion. On the other hand, there may be a better embedding for the list in which the number of list pointers crossing any cut is small compared to the capacity of the cut.

We can measure the quality of an embedding by generalizing the concept of load factor to a set of pointers. The *load* of a set $P$ of pointers across a cut $S = (A, \overline{A})$, denoted $\text{load}(P, S)$, is the number of pointers in $P$ from a processor in $A$ to a processor in $\overline{A}$ or vice versa, and the load factor is $\lambda(P) = \max_S \text{load}(P,S)/\text{cap}(S)$. The load factor of a data structure is the load factor of the set of its pointers.

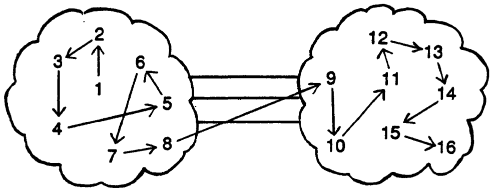


**Figure 1: A cut of capacity 3 separating two halves of a linked list. The load of the list on the cut is 1. At the final step of recursive doubling, each element on the left side of the cut accesses an element on the right, which induces a load of 8 on the cut.**

For many problems, good embeddings of data structures can be found (see Section 7). Even if a good embedding of a data structure is found, however, there is no guarantee that standard PRAM algorithms will behave efficiently on the data structure. To illustrate this point, consider the "recursive doubling" or "pointer jumping" technique which is used extensively by algorithms in the literature. The idea is that each element $i$ of a list initially has a pointer $p(i)$ to the next element in the list. At each step, element $i$ computes $p(i) \leftarrow p(p(i))$, doubling the distance $d(i)$ between $i$ and the element it points to (until it points to the end of the list). This technique can be used, among other things, to compute the distance of each element to the end of the list. For each element $i$, $d(i)$ is initially one. At each pointer jumping step, element $i$ computes $d(i) \leftarrow d(i) + d(p(i))$. In a PRAM model, the running time on a linked list of length $n$ is $O(\lg n)$. Variants of this technique are used for path compression [17,19,21], vertex numbering [20,21], and parallel prefix computation [21].

In the DRAM model, however, recursive doubling can be expensive even if a data structure has a good embedding. Figure 1 shows a cut of capacity 3 separating the two halves of a linked list of 16 elements. In the first step of recursive doubling, the load on the cut is only 1 because the only access across the cut occurs when element 8 copies the pointer of element 9. In the second step the load is 2 because element 7 accesses element 9 and element 8 accesses element 10. In the third step, the load is 4, and in the fourth step, the load is 8, as each of the first eight elements makes an access across the cut. Since the load factor of the cut in the fourth step is 8/3, this set of accesses will require a least 3 time units. Whereas the capacity of the cut was large enough to support the memory accesses across it in the first step, by the fourth step it was insufficient. The only way to guarantee that recursive doubling runs fast is to ensure that every cut of the network is sufficiently large to accommodate worst-case communication patterns. In the next section, we shall show how a *recursive pairing* strategy can perform many of the same functions as recursive doubling in a communication-efficient fashion.

All of our algorithms have the property that the load factor of memory accesses in any step is bounded by the load factor of the input data structure. We define a set $M$ of memory accesses to be *conservative* with respect to another set $M'$ of memory accesses if $\lambda(M) \leq \lambda(M')$, and we make the natural generalization of this definition to pointers and data structures. A *conservative algorithm* is one all of whose memory accesses are conservative with respect to the input data structure.

An algorithm that communicates only across pointers in the input data structure is conservative, but may require time linear in the diameter of the data structure to pass information between two elements. The following simple, but important, lemma shows how to shortcut pointers in the input data structure without in-

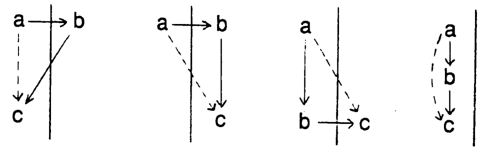


Figure 2: The Shortcut Lemma. In each of the four cases illustrated, the load factor across the cut is either unchanged or diminished by replacing $a \rightarrow b$ and $b \rightarrow c$ with $a \rightarrow c$.

creasing communication requirements.

**Lemma 1 (Shortcut Lemma)** *Let $P$ be the set of pointers in a data structure containing pointers $a \rightarrow b$ and $b \rightarrow c$. Then the set $P'$ of pointers defined by*

$$P' = P \cup \{a \rightarrow c\} - \{a \rightarrow b, b \rightarrow c\}$$

*is conservative with respect to $P$.*

**Proof:** We shall show that $\lambda(P', S) \leq \lambda(P, S)$ for any cut $S$ of the underlying network, which implies that $\lambda(P') \leq \lambda(P)$. Consider the eight ways in which $a$, $b$, and $c$ can be assigned to sides of the partition induced by a cut $S$. Half the cases can be eliminated by symmetry if we assume that $a$ is on the left side. In each of the four remaining cases, the load factor across the cut is either unchanged or diminished when $a \rightarrow b$ and $b \rightarrow c$ are replaced with $a \rightarrow c$, as is shown in Figure 2. ∎

We shall typically use a straightforward generalization of the Shortcut Lemma. Specifically, we can shortcut any set of pointer-disjoint paths in a data structure without increasing the load factor.

Because the algorithms presented in this paper are based on the Shortcut Lemma, they are not only conservative, but they are also independent of the cut capacities of the DRAM and of the embedding of an input data structure in the DRAM. Thus, independent of the underlying network, the algorithms are correct, and if the embedding of the input data structure is good, the algorithms run fast. Moreover, for a specific embedding on a specific DRAM, the running time can be analyzed precisely.

## 3. List contraction

In this section we show that a conservative "recursive pairing" algorithm, Algorithm LC can perform many of the same functions on lists as recursive doubling. The idea is to construct a $O(\lg n)$-height binary *contraction tree* whose leaves are the elements in the list. After building the contraction tree, operations such as broadcasting from the root or parallel prefix can be performed in a conservative fashion.

Algorithm LC is a randomized algorithm, and with high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$. A deterministic variant based on deterministic coin tossing [5] runs in $O(\lg n \lg^* m)$ steps, where $m$ is the number of processors in the DRAM, and produces trees of height $O(\lg n)$.

Algorithm LC requires a constant amount of extra space for each element in the input list. Each processor contains two elements, an element in the list, and a *spare* element that will act as an internal node in the contraction tree. We call the two elements in the same processor *mates*. Each element holds a pointer to an unused internal node, which for each list element initially points to its mate. The use of spare nodes allows the algorithm to distribute the space for the internal nodes of the contraction tree uniformly over the elements in the list. Spare internal nodes are used in [1] and [14] for similar reasons, but in a different context.

Algorithm LC now proceeds as follows.[2] (For a more complete description, see [16]). In each iteration, each element in the list randomly picks either its left or right neighbor. If two elements pick each other, they merge, and the left element takes control. A new internal node of the contraction tree is made using the spare of the left element. The spare for the new node is the spare of the right element. The new node's left child is the left element, and its right child is the right element. The new nodes and the unpaired nodes then form themselves into a "contracted" list in the terminology of Miller and Reif [17].

To describe the efficiency of randomized algorithms such as Algorithm LC, we shall use the term "with high probability," by which we shall mean "with probability $1 - O(1/n^k)$ for any constant $k > 0$," where $n$ is the size of the input.

**Theorem 2** *With high probability, Algorithm LC takes $O(\lg n)$ steps to construct a contraction tree for a circular list of $n$ elements.* ∎

**Theorem 3** *With high probability, the height of the contraction tree is $O(\lg n)$.*

**Proof:** The height of the contraction tree is not greater than the number of iterations of Algorithm LC. ∎

We can now prove that Algorithm LC is conservative.

**Theorem 4** *Algorithm LC is conservative.*

**Proof:** The key idea is that the order of the list elements and their spares is preserved by the operation of contraction. By convention, let the mate of an element in the input list lie in the order between that element and its right neighbor. Then in each iteration, an active element's spare lies between the element and its right neighbor in the contracted list. Thus, both the pointers of the contracted list and the pointers between active elements and their spares correspond to disjoint paths in the input list. The memory accesses in a step of the algorithm correspond to a set of pointers between active elements and their left or right neighbors in the contracted list, or to a set of pointers between active elements and their spares. ∎

Once a contraction tree has been constructed, it can be used to broadcast a value to all of the elements of the list, to accumulate values stored in each element of the list, and more generally, for performing *prefix computations*. Let $\mathcal{D}$ be a domain with a binary associative operation $\oplus$ and an identity element $\varepsilon$. A prefix computation [3,7,18] on a list with elements $x_1, x_2, \ldots, x_n$ in $\mathcal{D}$ puts the value $y_i$ in element $i$ for $i = 1, 2, \ldots, n$, where $y_i = x_1 \oplus x_2 \oplus \cdots \oplus x_i$.

A prefix computation on a list can be performed by a conservative, two-phase algorithm on the contraction tree. The leaves of the contraction tree from left to right are the elements in the list from $x_1$ to $x_n$. The first phase proceeds bottom up on the tree. Each leaf passes its $x$ value to its parent. As the algorithm proceeds, each internal node receives values from its left and right children, call them $z_l$ and $z_r$. The node saves value $z_l$, and passes $z_l \oplus z_r$ to its parent. The second phase proceeds top down after the root receives values from its children. The root then passes

[2] A similar algorithm works for circular lists.

$\varepsilon$ to its left child and its $z_l$ value to its right child. Each child receives a value from its parent, call it $z_p$, and passes that value to its left child and $z_l \oplus z_p$ to its right child. Each leaf receives the correct $y$ value.

The algorithm performs the prefix computation in $O(\lg n)$ steps. At each step, the algorithm communicates across a set of pointers in the contraction tree, all of which are the same distance from the leaves in the first phase and from the root in the second. That this computation is performed in a conservative fashion is a consequence of the following lemma.

**Theorem 5** *Let $CT$ be a contraction tree computed by Algorithm LC on an input list $L$, and suppose $P$ is a set of pointers of $CT$ in disjoint subtrees of $CT$. Then $P$ is conservative with respect to $L$.*

**Proof:** An inorder traversal of $CT$ alternately visits list elements (leaves) and their mates (internal nodes) in the same order that the list elements and mates appear in $L$. Thus, the pointers in $P$ correspond to disjoint paths in $L$. By the Shortcut Lemma, any set of pointers that correspond to disjoint paths in the list $L$ are conservative with respect to $L$. ▮

Algorithm LC, which constructs a contraction tree in $O(\lg n)$ steps, is a randomized algorithm. By using the "deterministic coin tossing" technique of Cole and Vishkin [5] the algorithm can be performed nearly as well deterministically. Specifically, the randomized pairing step can be performed deterministically in $O(\lg^* m)$ steps on a DRAM with $m$ processors, where $\lg^* m$ is the number of times the logarithm function must be successively applied to reduce $m$ to a value no greater than 1. The overall running time for list contraction is thus $O(\lg n \lg^* m)$.

## 4. Tree contraction

This section presents a conservative tree contraction algorithm, Algorithm TC, based on the tree contraction ideas of Miller and Reif [17]. The algorithm uses a recursive pairing strategy to build a contraction tree for an input rooted binary tree in much the same manner as Algorithm LC does for a list. Algorithm TC is a randomized algorithm, and with high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$. A deterministic variant based on deterministic coin tossing [5] runs in $O(\lg n \lg^* m)$ steps, where $m$ is the number of processors in the DRAM, and produces trees of height $O(\lg n)$.

We now outline Algorithm TC which performs tree contraction. (For a more complete description, see [16].) At each step, nodes in the input tree are paired. The pairing strategy has each node pick from among its neighbors according to how many children it has. A leaf picks its parent with probability 1. A node with exactly one child picks its child or its parent, each with probability 1/2. A node with two children picks each child with probability 1/2. The root, which has no parent, picks its children with equal probability. If two nodes pick each other, they merge and the parent takes over. A new internal node of the contraction tree is made from the spare of the parent in the pair. The spare of the new node is the spare of the child in the pair. One child in the contraction tree is the parent of the pair, and the other child in the contraction tree is the child of the pair. The new nodes and the unpaired nodes form themselves into a new tree, which is guaranteed to be binary by the the pairing strategy. The algorithm is applied recursively to the new tree.

After the input tree has been contracted to a single node, it may be "expanded" by undoing the contractions in the reverse of the order in which they occurred. When an internal node of the contraction tree expands to a parent–child pair, the tree pointers of these nodes are used to restore the tree to the state it had when the contraction took place. These pointers have been undisturbed by the algorithm since the nodes merged. The expansion requires only constant space at each node. In the next section we will see that tree expansion allows us to describe treefix computations recursively.

The proof that, with high probability, Algorithm TC takes $O(\lg n)$ steps to contract an input rooted binary tree to a single node requires three technical lemmas. The first lemma shows that in a binary tree, the number of nodes with two children and the number of leaves are nearly equal. The second lemma provides an elementary bound on the expectation of a discrete random variable with a finite upper bound. The last lemma presents a "Chernoff" [4] type bound on the tail end of a binomial distribution.

In the theorems and lemmas that follow, let $V_0$, $V_1$ and $V_2$ denote the sets of nodes (excluding the root) with zero, one, or two children, respectively, in a rooted binary tree of $|V|$ nodes. Let $|V_0|$, $|V_1|$, and $|V_2|$ denote the sizes of these sets and let $d(r)$ be the degree of the root.

The first lemma shows that in a binary tree, the number of nodes with two children (excluding the root) is equal to the number of leaves plus the degree of the root.

**Lemma 6** $|V_2| + d(r) = |V_0|$ ▮

The next lemma provides a lower bound on the probability that a discrete random variable with a finite upper bound will be larger than some fixed value.

**Lemma 7** *Let $X \leq b$ be a discrete random variable with expected value $\mu$. For $w < b$,*

$$\Pr(X \geq w) \quad \geq \quad \frac{\mu - w}{b - w} \, . ▮$$

The final lemma presents a bound on the tail end of a binomial distribution. Consider a set of $t$ independent Bernoulli trials, each occurring with probability $p$ of success. The probability that fewer than $s$ successful trials occur is

$$B(s,t,p) \quad = \quad \sum_{k=0}^{s-1} \binom{t}{k} p^k (1-p)^{t-k} \, .$$

The lemma bounds that probability $B(s,t,p)$ that fewer than $s$ successes occur in $t$ trials when $t > 2s$ and $p < 1/2$.

**Lemma 8** *For $t > 2s$ and $p < \frac{1}{2}$,*

$$B(s,t,p) \quad \leq \quad \left(\frac{1-p}{1-2p}\right) \left((1-p)^t\right) \left(\frac{et}{s}\right)^s \, . ▮$$

We now prove that with high probability, Algorithm TC takes $O(\lg n)$ steps to contract a rooted binary tree to a single node. The key observation in the proof is that for each node that pairs with its parent, the number of nodes in the tree decreases by one.

**Theorem 9** *With high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract a rooted binary tree of $n$ nodes to a single node.*

**Proof:** The proof has three parts. We use Lemma 6 to show that that if a rooted binary tree has $|V|$ nodes, the expected number of nodes pairing with a parent is at least $|V|/4$. Next, we call a pairing step "successful" if at least $|V|/8$ nodes pair with a parent. In the resulting contraction, the size of the tree decreases by at least a factor of 7/8. We use Lemma 7 to prove that the probability that a pairing step is successful is at least 1/3. Finally, we use Lemma 8 to show for any constant $k$ that after $\alpha \log_{8/7} n$ steps, for some constant $\alpha > 2$, the probability that fewer than $\log_{8/7} n$ successful steps occur is $O(1/n^k)$.

We first show that the expected number of nodes pairing with a parent is at least $|V|/4$. A node is picked by its parent with probability 1 when its parent is a degree 1 root, and 1/2 otherwise. Thus a leaf pairs with its parent with probability at least 1/2, and a node (other than the root) with one child picks its parent with probability at least 1/4. Let $P$ be the number of nodes pairing with a parent. Apply Lemma 6 to the simple bound on the expected value of $P$,

$$\mathrm{E}(P) \quad \geq \quad \frac{|V_0|}{2} + \frac{|V_1|}{4}$$

to yield the desired result:

$$\mathrm{E}(P) \geq \frac{|V_0| + |V_1| + |V_2| + d(r)}{4} \geq \frac{|V|}{4} .$$

Now we show that the probability that a pairing step is successful is at least 1/3. At most half of the nodes pair with their parents. Using Lemma 7 with $b = |V|/2$, $w = |V|/8$, and $\mu \geq |V|/4$ we have

$$\Pr(P \geq |V|/8) \geq \frac{\frac{|V|}{4} - \frac{|V|}{8}}{\frac{|V|}{2} - \frac{|V|}{8}} = \frac{1}{3} .$$

Finally, we show that with high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract the input tree to a single node. In the contraction following a successful pairing step, the size of the tree decreases by a factor of 7/8 or more. After $\log_{8/7} n$ successful steps, the tree must consist of a single node. By Lemma 8 with $p = 1/3$, the probability that fewer than $s = \log_{8/7} n$ successful steps occur in $\alpha s$ steps is

$$B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3) \quad \leq \quad 2\left(\left(\frac{2}{3}\right)^{\alpha} \alpha e\right)^{\log_{8/7} n} .$$

For any value $k$, we can choose $\alpha$ so that $B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3)$ is $O(1/n^k)$. In particular, for $k = 1$, setting $(2/3)^{\alpha}\alpha e \leq 7/8$ yields $\alpha \geq 8$. ∎

We now prove that Algorithm TC is conservative.

**Theorem 10** *Algorithm TC is conservative.*

**Proof:** The key idea is that each active element in the contracted tree is a "representative" of a subgraph of the input tree that has been contracted to a single node. The contracted subgraphs, which are trees, are disjoint in the input tree. The representative and spare of a subgraph are either elements in or mates of elements in the subgraph. The pairing strategy ensures that each subgraph is adjacent by an edge to at most one subgraph which is higher in the input tree, and to at most two subgraphs which are lower. The representative of the subgraph has pointers to the representatives of these subgraphs, and to the spare of the subgraph.

As in the list contraction algorithm, memory accesses in a step of the tree contraction algorithm corresponds to a set of disjoint paths in the input data structure. Since each subgraph is connected, the pointers between representatives and spares correspond to disjoint paths in the input tree. Similarly, any set of pointers between each representative and the representative of at most one of the two adjacent subgraphs lower in the input tree corresponds to a set of disjoint paths in the input tree. The memory accesses in a step correspond to a set of pointers between representatives and spares or to a set of pointers between each representative and the representative of at most one of the two adjacent subgraphs lower in the input tree. ∎

Tree contraction can be performed conservatively and deterministically in $O(\lg n \lg^* m)$ steps on a DRAM with $m$ processors using the deterministic coin tossing algorithm of Cole and Vishkin [5]. The key idea is that in Algorithm TC, the nodes in the tree that can pair form chains, and by Lemma 6 these chains contain at least half the tree edges. The chains can be oriented from child to parent in the tree, and deterministic coin tossing can be used to perform the pairing step in $O(\lg^* m)$ steps.

## 5. Treefix computations

This section presents a generalisation of the parallel prefix computation to binary trees. We present two kinds of *treefix* computations—*rootfix* and *leaffix*—and show how they can be implemented by an $O(\lg n)$-step conservative algorithm in linear space. As we shall see in Section 6, treefix computations can greatly simplify the description of many parallel graph algorithms in the literature, and moreover, treefix computations can be performed by conservative algorithms.

We begin with a definition of treefix computation.

**Definition 11** *Let $\mathcal{D}$ be a domain with a binary associative operation $\oplus$ and an identity element $\varepsilon$. Let $T$ be a rooted, binary tree in which each vertex $i \in T$ has an assigned value $x_i \in \mathcal{D}$. The rootfix problem is to compute for each vertex $i \in T$ with parent $j$, the value $y_i = y_j \oplus x_i$, where $y_j = \varepsilon$ if $i$ is the root. The leaffix problem is to compute for each vertex $i \in T$ with left child $j$ and right child $k$, the value $y_i = x_i \oplus y_j \oplus y_k$, where $y_j = \varepsilon$ if $i$ has no left child and $y_k = \varepsilon$ if $i$ has no right child.*

Simple examples of treefix problems are computing the depth of each vertex in a rooted binary tree and computing the size of each subtree. These and other examples appear in the next section.

Like the prefix computation on lists, treefix computations can be performed directly on the contraction tree. To simplify the description here, however, we describe a recursive version. We execute one contraction step and then recursively perform a treefix computation on the new tree. The treefix values for the input tree can be computed immediately from the treefix values for the new tree. The recursion level of a node in the contraction tree can be maintained by recording the step in which it was created.

**Theorem 12** *A rootfix or leaffix computation can be performed by a conservative randomized algorithm which, with high probability, takes $O(\lg n)$ steps, or by a conservative deterministic algorithm which takes $O(\lg n \lg^* m)$ steps, where $m$ is the number of processors in the DRAM.*

**Proof:** We first describe the computation for rootfix. First the input binary tree $T$ is transformed to a new tree $T'$ by one contraction step. Each new node $u$ in $T'$ resulting from the pairing of parent $p$ and child $c$ in $T$ passes input value $x_c$ to the child in $T'$ that $u$ inherited from $c$. Node $u$ passes $\varepsilon$ to its other child. Each unpaired node $v$ in $T'$ passes $\varepsilon$ to each of its children. Each node in $T'$ receives a value from its parent, call it $z$. Each new node $u$ computes $x'_u \leftarrow z \oplus x_p$. Each unpaired node $v$ computes $x'_v \leftarrow z \oplus x_v$. A rootfix computation is performed recursively on $T'$ using the $x'$ values as input and yielding $y'$ values as output. The contraction step from $T$ to $T'$ is then undone. Each new node $u$ passes $y'_u$ to $p$ and $c$. Node $p$ computes $y_p \leftarrow y'_u$ and $c$ computes $y_c \leftarrow y'_u \oplus x_c$. Each unpaired node $v$ computes $y_v \leftarrow y'_v$.

We now describe a computation of which leaffix is a special case. Each node $i$ in $T$ is assigned input values $x_i$, $l_i$, and $r_i$. Node $i$ with left child $j$ and right child $k$ computes output value $y_i = x_i \oplus y_j \oplus l_i \oplus y_k \oplus r_i$, where $y_j$ and $l_i$ are $\varepsilon$ if $i$ has no left child and $y_k$ and $r_i$ are $\varepsilon$ if $i$ has no right child. For the special case of leafix, $l_i$ and $r_i$ are both $\varepsilon$. First, a contraction step transforms $T$ to $T'$. Consider each new node $u$ in $T'$ resulting from the pairing of parent $p$ and left child $c$ in $T$ with input values $x_p$, $l_p$, $r_p$ and $x_c$, $l_c$ respectively. (The cases where $c$ is a right child, or where $c$ has a right child or is a leaf, are similar.) Node $u$ computes $x'_u \leftarrow x_p \oplus x_c$, $l'_u \leftarrow l_c \oplus l_p$, and $r'_u \leftarrow r_p$. Each unpaired node $v$ computes $x'_v \leftarrow x_v$, $l'_v \leftarrow l_v$, and $r'_v \leftarrow r_v$. The computation is performed recursively on $T'$ using the $x'$ values as input and yielding $y'$ values as output. Each node passes its $y'$ value to its parent in $T'$. Each node receives values from its left and right children, call these values $z_l$ and $z_r$. Each new node $u$ passes $z_l$ to $c$ and both $z_l$ and $z_r$ to $p$. The contraction step from $T$ to $T'$ is undone. Node $c$ computes $y_c \leftarrow x_c \oplus z_l \oplus l_c$. Node $p$ computes $y_p \leftarrow x_p \oplus x_c \oplus z_l \oplus l_c \oplus l_p \oplus z_r \oplus r_p$. Each unpaired node $v$ computes $y_v \leftarrow y'_v$. ∎

## 6. Conservative algorithms

This section presents a collection of conservative DRAM algorithms, all of which use treefix computations. The algorithms use two processors per edge of an input graph $G = (V, E)$ and require constant extra space in each processor. Since the algorithms are based on shortcutting pointers in the input data structure, they are independent of the underlying DRAM or embedding of the data structure.

We represent each vertex in an undirected graph $G = (V, E)$ by a doubly linked *incidence ring* of processors, one for each edge. Each element of the incidence ring contains pointers to the next and previous elements in the ring, and one pointer for a graph edge. For each edge $(u, v) \in E$ the element in the incidence ring for $u$ contains a pointer to an edge element in the incidence ring for $v$, and vice versa. A directed graph is represented in the same doubly linked fashion, but the graph edges are labeled with their directions.

We represent trees with arbitrary vertex degrees by an incidence ring structure as well. If the tree is directed, each ring has a unique *principal element* that points toward the root. Breaking the incidence ring before the principal element yields the standard binary tree representation of the tree [10, pp. 332–333].

We now present brief descriptions of the algorithms. The performance is given terms of the number of steps on a DRAM when the input representation has size $n$. We assume the implicit tree contractions in the algorithms are performed by the randomized Algorithm TC. Deterministic bounds can be obtained by multiplying the number of steps by $O(\lg^* m)$, where $m$ is the number of processors. An upper bound on the actual performance can be obtained by multiplying the number of steps by the load factor of the input.

**Generalized treefix.** *Perform a treefix operation on a directed tree with arbitrary vertex degree. The input values $\{x_i\}$ are stored in the principal elements of the tree, which is where the output values $\{y_i\}$ are to be placed. The leaffix value at a node $i$ whose children have values $y_1, y_2, \ldots, y_k$ is $y_i = x_i \oplus y_1 \oplus y_2 \oplus \cdots \oplus y_k$.* Each element that is not principal stores the identity element $\varepsilon$ for its value. A binary treefix computation performed on the binary tree representation underlying the tree computes the desired values. *Performance:* $O(\lg n)$.

**Tree functions.** *Given a directed tree, compute for each node the number of descendents, its height, or its depth.* The number of decendents for each node can be computed by a leaffix computation with $\oplus$ as integer addition and $x_i = 1$ for all nodes. The height of a node can also be computed by a leaffix computation where $a \oplus b = \max(a + 1, b + 1)$, the identity is $\varepsilon = -\infty$, and $x_i = 0$ for all nodes. The depth of a node can be computed by a rootfix computation with $\oplus$ as addition and $x_i = 1$ for all nodes except the root which has value 0. *Performance:* $O(\lg n)$.

**Rooting an undirected tree.** *Pick a root of a tree with undirected graph pointers and orient the graph pointers toward the root.* Form an "Eulerian tour" of the pointers of the representation [20] by directing each element of the tree to link its incoming ring pointer with its graph edge directed outward and its graph edge directed inward with its outgoing ring pointer. Each graph edge is used twice in the tour, once in each direction, but each ring pointer is used only once. Using the variant of Algorithm LC which works for circular lists, form a contraction tree of the tour. Choose the root of the contraction tree to be the root of the tree, and break the tour so that it begins with the root. Use parallel prefix to number each node according to its first occurrence in the tour. Use contraction trees to distribute the smallest value in each incidence ring to the elements of the ring. Orient each graph edge from the larger value to the smaller. *Performance:* $O(\lg n)$.

**Rerooting a directed tree.** *Given a directed tree and another distinguished vertex $k$, reorient the graph edges of the tree to point to $k$.* The algorithm for rooting a tree can be used by picking $k$ as the root instead of the root of the contraction tree, but a single treefix computation suffices. Perform a leaffix computation with $x_k = 1$ and $x_i = 0$ if $i \neq k$, and use Boolean OR for $\oplus$. Each principal element whose leaffix value is 1 lies on the path from $x_k$ to the root. Reverse the direction of the graph pointers of these elements. (Note: rerooting a tree changes the principal elements.) *Performance:* $O(\lg n)$.

**Tree-walk numberings of a binary tree.** *Number the nodes of a binary tree according to the order they would be visited in a preorder/inorder/postorder tree walk.* For each of the walks, we will compute $y_k$, the number of nodes visited before the left subtree of $k$. Use a leaffix computation to compute the number $size_k$ of the subtree rooted at $k$. We first compute the preorder numbering. (For the purposes of these numbering algorithms, we consider the root to be a left child.) If node $k$ is a left child, set $x_k$ to 1. If node $k$ is a right child, set $x_k$ to 1 plus the size of its sibling subtree. A rootfix computation with + yields $y_k$, which is the preoder numbering of node $k$. The inorder numbering can be computed similarly. If node $k$ is a left child, set $x_k$ to 0. If $k$ is a right child, set $x_k$ to 1 plus the size of its sibling subtree. Compute $y_k$ for each node using a rootfix computation with +. The inorder numbering of node $k$ is 1 plus $y_k$ plus the size of its left subtree. The postfix numbering can be computed by setting $x_k$ to 0 if node $k$ is a left child, and by setting $x_k$ to the size of its sibling subtree if $k$ is a right child. After computing $y_k$ using a rootfix computation with +, the postfix numbering of node $k$ is 1 plus $y_k$ plus the sizes of its two subtrees. *Performance:* $O(\lg n)$.

**Prefix/postfix numbering of a directed tree.** *Number the edges of an arbitrary directed tree according to the order they are visited in preorder/postorder tree walk.* The problem reduces to prefix/postfix numbering on the underlying binary tree representation. *Performance:* $O(\lg n)$.

**Diameter and center of a tree.** *The diameter is the length of the longest path in the tree. A center is a vertex $v$ such that the longest path from $v$ to a leaf is minimal over all vertices in the tree.* The diameter can be determined by rooting the tree and using rootfix to find the furthest leaf from the root. Reroot the tree at this leaf. The distance from the new root to the furthest leaf is the diameter. (Based on an analog algorithm attributed to J. Wennmacker [6].) A center of the tree can be determined by finding a median element of the path that realizes the diameter. *Performance:* $O(\lg n)$.

**Centroid of a tree.** *A centroid is a vertex $v$ such that the largest subtree with $v$ as a leaf is minimal over all vertices in the tree.* A centroid can be determined by rooting the tree and computing the size of each subtree. By broadcasting the size $m$ of the tree from the root, each graph edge in each incidence ring can determine the number of elements on the other side of the edge. For each incidence ring, compute the maximum of these values. A vertex with the minimum of these maximum values is a centroid. *Performance:* $O(\lg n)$.

**Separator of a tree.** *A separator* [15] *is a partition of the vertices of an $m$-vertex tree into three sets $A$, $B$, and $C$, with $|A| \leq \frac{2}{3}m$, $|B| = 1$, and $|C| \leq \frac{2}{3}m$, such that no edge of the tree goes between a vertex in $A$ and a vertex in $C$.* Determine a centroid of the tree. This vertex is the separator vertex in $B$. It remains to partition the remaining vertices between $A$ and $C$. For each graph edge in the incidence ring, count the number of vertices in the subtree on the other side of the edge. Put the largest subtree in $A$. Use parallel prefix on the incidence ring to compute a running sum of the sizes of the other subtrees. Put all subtrees whose prefix value is at most $\frac{2}{3}m$ in $C$, and put the remainder in $A$. *Performance:* $O(\lg n)$.

**Subexpression evaluation.** *Given a directed tree in which each leaf has a value and each internal node has an operator from $\{+, -, \cdot, \div\}$, compute for each internal node the subexpression rooted at that node.* A single leaffix computation suffices using the ideas of Brent [2] and Miller and Reif [17]. *Performance:* $O(\lg n)$.

**Minimum cost spanning tree.** *Given an undirected input graph $G = (V, E)$ and a cost function $w : E \to \mathbf{R}$, determine a set $F \subseteq E$ of edges such that each vertex in $V$ is incident on an edge of $F$, and the sum of the weights of the edges in $F$ is minimal.* We give a conservative DRAM implementation of Sollin's algorithm [8, section 5.5]. We assume without loss of generality that the edge weights are distinct—otherwise, we can assign the weight of a graph edge $e$ between two incidence ring elements with addresses $a$ and $b$ to be $(w(e), \max(a, b), \min(a, b))$ and then compare weights lexicographically. We determine $F$ by marking edges in $G$. Initially, no edges are marked. At each step of the algorithm, the currently marked graph edges form a subforest of $F$. Break each incidence ring by removing a single ring pointer and direct the resulting linear list. At each step of the algorithm, the marked graph edges and the ring pointers form a set $\{T_i\}$ of rooted trees, where the index $i$ of the tree is the address of the root. The algorithm proceeds as follows. For each tree $T_i$, use a rootfix computation to broadcast $i$ to all of the elements in $T_i$. Use a leaffix computation on $T_i$ to determine an edge $e \in E$ with the smallest weight $w(e)$ connecting an edge element $u \in T_i$ with an edge element $v \in T_j$, where $i \neq j$. If no such edge exists, the algorithm terminates. If $T_j$ picks the same edge as $T_i$, the tree with smaller index does nothing. Otherwise, mark $e$ as a member of $F$, directing it into $T_j$, and reroot $T_i$ with $u$ as the new root. Repeat this procedure until the algorithm terminates. *Performance:* $O(\lg^2 n)$.

**Connected components.** *Given an undirected input graph $G = (V, E)$, determine a labeling $l : V \to \mathbf{Z}$ such that such that $l(v) = l(v')$ if and only if $v$ and $v'$ are in the same connected component of $G$.* The algorithm is the same as the minimum spanning tree algorithm, choosing the weight of a graph edge $e$ between incidence ring elements with addresses $a$ and $b$ to be $\max(a, b), \min(a, b)$. The label of a vertex is the index of its tree. *Performance:* $O(\lg^2 n)$.

**Biconnected components.** *Two edges of an undirected graph $G = (V, E)$ are in the same biconnected component if they lie on a common simple cycle. Determine a labeling $l : E \to \mathbf{Z}$ such that $l(e) = l(e')$ if and only if $e$ and $e'$ are in the same biconnected component of $G$.* We give a conservative DRAM implementation of the biconnectivity algorithm of Tarjan and Vishkin [20]. We assume that the reader has some familiarity with that algorithm. Find a (directed) minimum spanning tree $T = (V, F)$ of $G$. Number the vertices in the minimum spanning tree in preorder. Use leaffix computations to compute for each vertex $v$ three values: $\mathrm{nd}(v)$, $\mathrm{low}(v)$, and $\mathrm{high}(v)$. Here $\mathrm{nd}(v)$ is the number of descendants of $v$, while $\mathrm{low}(v)$ and $\mathrm{high}(v)$ are the lowest and highest vertices (with respect to the preorder numbering of $T$) that are either a descendant of $v$ or adjacent to a descendant of $v$ by an edge of $E - F$. Build a new graph $G'$ where the edges of $F$ are the vertices of $G'$. Let $e$ be an edge from $u$ to $p(u)$, where $p(u)$ is the parent of $u$ in $F$. The adjacency ring for $u$ in $G$ acts as the adjacency ring for $e$ in $G'$. Add two kinds of edges to $G'$. For each edge $\{w, v\}$ in $E - F$ such that $v + \mathrm{nd}(v) \leq w$, add an edge $\{\{v, p(v)\}, \{w, p(w)\}\}$ to $G'$. For each edge $(v, p(v))$ of $F$ such that $v \neq 1$ and $p(v) \neq 1$, and $\mathrm{low}(v) < \mathrm{low}(p(v))$ or $\mathrm{high}(v) \geq p(v) + \mathrm{nd}(p(v))$, add an edge $\{\{v, p(v)\}, \{p(v), p(p(v))\}\}$ to $G'$. It can be verified that the representation of $G'$ is conservative with respect to the representation of $G$. Find the connected components of $G'$. Two edges of $F$ are in the same block if as vertices in $G'$ they are in the same connected component. Finally, for each edge $e = \{w, v\}$ in $E - F$, let $l(e) = l(\{w, p(w)\})$. *Performance:* $O(\lg^2 n)$.

**Eulerian cycle.** *An Eulerian cycle of an undirected graph $G = (V, E)$ is a cycle containing each edge in $E$ exactly once.* If any vertex has odd degree, then no Eulerian cycle exists. Form a set of disjoint cycles of the pointers of the representation of $G$ as in the algorithm for directing a tree. The cycles can be merged using an algorithm similar to the minimum spanning tree algorithm. *Performance:* $O(\lg^2 n)$.

## 7. Concluding remarks

The efficiency of a DRAM algorithm depends on how well its input is embedded in the DRAM, but this embedding problem must be faced by algorithm designers in any bandwidth-limited distributed network. In general, the problem of determining the best embedding is NP-complete, but for many common situations, good embeddings can be found. For example, any of our polylogarithmic-step conservative algorithms can be run on planar graphs properly embedded in an area-universal fat-tree, and they will exhibit polylogarithmic time performance.

As another example, a subproblem in switch-level simulation of a VLSI circuit is the finding of electrically equivalent portions of the circuit. A naive divide-and-conquer embedding of the circuit on an area-universal fat-tree [12] yields small load factors for every cut. Thus, our conservative connected components algorithm will never cause undue congestion in communicating messages in the underlying network, and the algorithm will run effectively as fast as on an expensive, high-bandwidth network.

Many other classes of algorithms can be implemented in a conservative fashion on a DRAM. Any algorithm that communicates olny across pointers in an input data structure is conservative. Passing a single datum between two processors, however, can require time linear in the diameter of the data structure, whereas our algorithms all run in a polylogarithmic number of steps. As another example, systolic array algorithms for matrix problems

[11,14] can be implemented efficiently if the matrices are properly embedded. In general, any fixed-connection network algorithm will run well on a DRAM if the communication required by the network can be supported by the underlying DRAM network.

As a final comment, it may well be that the notion of a conservative algorithm is too conservative. A contraction tree is not conservative with respect to its input tree (though the levels of the contraction tree are), but the load factor of the contraction tree is at most $O(\lg n)$ times the input load factor. As a practical matter, it is probably not worth worrying whether every set of memory accesses is conservative with respect to the input, as long as the load factor of memory accesses is polylogarithmically bounded. Algorithms with this looser bound are somewhat easier to code because of the relaxed constraint, and they should perform comparably.

## Acknowledgments

Thanks to Baruch Awerbuch, Ravi Boppana, Benny Chor, Tom Cormen, Johan Hastad, Alex Ishii, Joe Kilian, Tom Leighton, and Ron Rivest of MIT for numerous helpful discussions. Thanks to Ginny Loop for producing the figures.

## References

[1] S. N. Bhatt and C. E. Leiserson, "How to assemble tree machines," *Advances in Computing Research*, Vol. 2, *VLSI Theory*, F. P. Preparata, ed., JAI Press, Greenwich, Conn., 1984, pp. 95–114.

[2] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *JACM*, Vol. 21, No. 2, April 1974, pp. 201–208.

[3] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, Vol. C–31, No. 3, March 1982, pp. 260–264.

[4] H. Chernoff, "A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations," *Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 493–507.

[5] R. Cole and U. Vishkin, "Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms," *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing*, May 1986, pp. 206–219.

[6] A. K. Dewdney, "Computer Recreations," *Scientific American*, Vol. 252, No. 6, June 1985, pp. 18–29.

[7] M. J. Fischer and R. E. Ladner, "Parallel prefix computation," *JACM*, Vol. 27, No. 4, October 1980, pp. 831–838.

[8] S. E. Goodman and S. T. Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York, 1977.

[9] R. I. Greenberg and C. E. Leiserson, "Randomized routing on fat-trees," *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE, October 1985, pp. 241–249.

[10] D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

[11] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," *SIAM Sparse Matrix Proceedings*, I. S. Duff and G. W. Stewart, ed., 1978, pp. 256–282.

[12] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, Vol. C–34, No. 10, October 1985, pp. 892–901.

[13] F. T. Leighton, *Complexity Issues in VLSI*, MIT Press, Cambridge, Massachusetts, 1983.

[14] C. E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts, 1983.

[15] R. J. Lipton and R. E. Tarjan, "A planar separator theorem," *Siam J. of Applied Math*, Vol. 36, No. 2, 1979, pp. 177–189.

[16] B. M. Maggs, *Communication-Efficient Parallel Graph Algorithms*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1986.

[17] G. Miller and J. Reif, "Parallel tree contraction and its application," *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE, October 1985, pp. 478–489.

[18] Yu. Ofman, "On the algorithmic complexity of discrete functions," English translation in *Soviet Physics – Doklady*, Vol. 7, No. 7, 1963, pp. 589–591.

[19] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, Vol. 3, 1982, pp. 57–67.

[20] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, IEEE, October 1984, pp. 12–20.

[21] J. C. Wyllie, *The Complexity of Parallel Computations*, Ph.D. thesis, Cornell University, Ithaca, N. Y., August 1979.

# EFFICIENT PARALLEL ALGORITHMS FOR GRAPH PROBLEMS

**Clyde P. Kruskal**
Department of Computer Science
University of Maryland
College Park, Maryland 20742

**Larry Rudolph**
Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

**Marc Snir**
Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

**ABSTRACT** New efficient techniques for manipulating linked structures in parallel are presented. These techniques work on the EREW machine, which is the weakest shared memory machine. Using these techniques we develop algorithms for connected components, spanning trees, minimum spanning trees, and other graph problems. All of these algorithms achieve linear speed-up for all but the sparsest graphs. We also present a new parallel radix sort algorithm that is optimal for keys taken from a small range.

## 1. INTRODUCTION

We present new parallel algorithms for several important graph problems including spanning forest, connected components, biconnected components, and (undirected) minimum cost spanning tree (MST). All of the algorithms are improvements over the best results known in several respects: they assume the weakest model of shared memory parallel computation, i.e. the EREW (exclusive read, exclusive write) model; they are deterministic; they are fast on all graph densities. In addition, the input is not required to be in any prearranged order, such as adjacency lists; the input can consist of an unordered list of edges. When the problem size is large relative to the number of processors and the graphs are not extremely sparse (i.e. as long as $n = o(e)$), the algorithms achieve a linear speedup, which means they have optimal processor-time product. As a by product, we obtain a parallel version of radix sort that is optimal as long as the range of the elements sorted is at most polynomially larger than the number of processors.

The algorithms employ a set of useful techniques for efficient parallel manipulation of data structures to avoid conflicts between processors. The most powerful and original technique is a recursive two-step, block-based algorithm. It is used in an algorithm for solving the "parallel prefix" problem on a linked list of elements. In quite a different way, it is used to convert an edge list representation of a graph into an adjacency lists representation. The former representation is how a graph is often defined; the latter representation is often required for efficient parallel access to edges of the graph. Variants of the technique are shown to be useful for performing table lookup, update without conflicts, and radix sort.

Sections 2, 3, and 4 describe basic building blocks. Section 2 reviews the model and some work on solving linked list problems in parallel. In Section 3, we show how a graph can be converted from an edge list representation to an adjacency list representation efficiently in parallel. We also present the parallel radix sort algorithm. Section 4 shows how tree problems can be solved efficiently in parallel. We then use the basic building blocks in Section 5 to obtain parallel algorithms for finding spanning trees, connected components, biconnected components, and minimum spanning trees. A chart at the end of the paper summarizes our results and shows how they compare to the current best.

## 2. PRELIMINARIES

### 2.1. The Model

We assume the PRAM computation model: a PRAM consists of $p$ autonomous processors, all having access to a common memory. At each step, each processor performs one operation from its instruction stream. Instructions accessing shared memory are also assumed to be accomplished in one cycle. Borrowing the notation of [12], we distinguish three variants of the PRAM family: In a Concurrent Read, Concurrent Write (CRCW) model, processors may simultaneously access the same memory location; there are various schemes for resolving write conflicts. In a Concurrent Read, Exclusive Write (CREW) model, several processors may simultaneously read the value of a memory location, but exclusive access is required for write. Finally, in the Exclusive Read, Exclusive Write (EREW) model, a memory location cannot be simultaneously accessed by more than one processor. Our algorithms can be implemented on the weakest of these three models: the EREW PRAM.

### 2.2. Linear Linked List Problems

We now review our algorithm for solving problems on linear linked lists [7]; it will be extended in the following sections to more general linked structures. Although our results for linear linked lists have since been improved by Cole and Vishkin [4], their technique does not seem to extend to more general linked structures.

The *product computation* problem is to compute the product $a_0 \, o \, a_1 \, o \cdots o \, a_{n-1}$, given $n$ elements $a_0, a_1, \ldots, a_{n-1}$, and a binary, associative operation, denoted $o$, e.g. ordinary, matrix, and Boolean addition and multiplication. The *initial prefix* problem is to compute all $n$ initial prefixes $a_0$, $a_0 \, o \, a_1$, ..., $a_0 \, o \, a_1 \, o \cdots o \, a_{n-1}$. The initial prefix problem when solved in parallel is known as *parallel prefix*. When the elements are laid out in known memory locations there are well known methods to solve both problems in time $O(n/p + \log p)$ [9].

The problem is harder to solve when the elements are stored in a linear linked list. The locations of the elements is given, but it is not known which element is which. Only the location of the first element is given along with a map *Succ* from the $i$th element to the $(i+1)$st element, and a map *Pred* from the $i$th element to the $(i-1)$st element (the second mapping can be computed from the first in $O(n/p)$ time).

Part of the work was done while the first author was at the University of Illinois, Urbana-Champaign, the second author was at Carnegie-Mellon University, and the third author was at the Courant Institute of Mathematical Sciences, New York University.

0190-3918/86/0000/0869 $01.00 © 1986 IEEE

It is easy to solve the product and initial prefix problems sequentially in $O(n)$ time by starting at the first element and following the links. Wyllie [15] shows that this problem can be solved with $p < n$ processors in $O(n \log n / p)$ time, using a recursive doubling technique. The algorithm never achieves linear speedup.

An efficient parallel solution to this problem has the following general form.

```
General Parallel Prefix Algorithm
  if the list contains more than one element then
  begin
    pick a set $S$ of nonadjacent elements in the list;
    for each $a \in S$ do
    begin
      {replace a pair of adjacent elements by one
        element}
      $val(succ(a)) := val(a) \; o \; val(succ(a))$;
      $succ(pred(a)) := succ(a)$;
      $pred(succ(a)) := pred(a)$
    end;
    apply algorithm recursively to the list;
    for each $a \in S$ do
    begin
      {expand element back into a pair}
      $val(a) := val(pred(a)) \; o \; val(a)$;
      $succ(pred(a)) := a$;
      $pred(succ(a)) := a$
    end
  end
```

The first part of the algorithm solves the product problem, successively compacting the list until only one item is left; the second part, where the recursion unfolds, expands the list back, and computes the missing partial products. Any parallel algorithm that solves the product problem by successive compaction can be used to create a parallel prefix algorithm, that expands the list back by matching step by step the compression operations done by the parallel product algorithm. The resulting algorithm will have twice the running time of the original algorithm. We shall henceforth consider only the product part.

It is not obvious how to pick at each iteration a set $S$ of $\Omega(p)$ nonadjacent element, while devoting only constant time per element. Assume w.l.o.g. that both $n$ and $p$ are powers of 2 (this affects the result only by a constant factor), and that $p \leq n/2$. The product algorithm consists of $\Theta(\log n)$ iterations. At each iteration the elements are partitioned into $n/p$ blocks, each block containing $p$ elements. The processors visit the blocks one by one; the $i$ th processor always visits the $i$ th element within a block. It pairs this element with its successor, only if the element is not marked as deleted, and its successor belongs to another block. It takes $O(n/p)$ time to process all $n/p$ blocks.

After visiting the blocks, every element has formed the product with a neighbor, provided it and its neighbor were in different blocks; all pairings remaining to be done at this phase are internal to a block. Since there are $n/p$ blocks and $p$ processors, $p^2/n$ processors can be assigned to each block.

If $p^2/n > 1$, then we recursively apply the same routine within each block. If $p^2/n \leq 1$, assign each processor $n/p^2$ blocks, and pair elements sequentially with each block. Note that $p$ and $p^2/n$ are powers of 2.

After this, if an element has not been paired with one of its neighbors, then both of its neighbors must be paired (except for the first and last elements of the list), so at most $2n/3 + O(1)$ elements are left.

Let $H_p(n)$ be the number of steps required by one iteration on a list of length $n$.

$$H_p(n) = O(\frac{n}{p}) + H_{p^2/n}(p) \qquad \text{if } p > 1,$$

$$H_p(n) = O(n/p) \qquad \text{otherwise}.$$

It is easy to check that

$$H_p(n) = O(\frac{n}{p}\frac{\log n}{\log(n/p)}).$$

After each iteration, the remaining elements are packed to the top of array, being careful to maintain the integrity of the pointers (which can be done by first determining the location of each element, then updating pointers to point to the new locations, and finally moving the elements). Once there are less than $2p$ elements, the $O(n \log n / p) = O(\log p)$ parallel algorithm is used to complete the algorithm.

Let $T_p(n)$ be the number of steps it takes to solve a problem of size $n$, i.e. the execution time of the procedure Product. If $n < 2p$, $T_p(n) = O(\log n)$ using the standard parallel algorithm. Otherwise,

$$T_p(n) = O(H_p(n) + \frac{n}{p} + \log p) + T_p(\frac{2n}{3}).$$

It is not hard to show that

$$T_p(n) = O(\frac{n}{p}\frac{\log n}{\log(2n/p)}).$$

We obtain

**Theorem 2.1:** The parallel prefix problem for items represented by a linked list can be solved on an EREW machine in $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ time for $p < n$.

**Note:** This yields, for any constant $0 < \epsilon \leq 1$, a linear speedup when $p \leq n^{1-\epsilon}$ (or equivalently when the time $T_p(n) \geq n^{\epsilon}$).

This algorithm can be used to solve the *linked list packing problem*: Given a list of $n$ items, a subset of which are *active*, pack the active elements to contiguous locations of memory (starting at some specified location). We can use the parallel prefix algorithm to rank the active elements, and then move each one to the location indicated by its rank plus the starting location. As a by product, the elements are packed in order so that the $i$ th element is contiguous to the $(i-1)$st and $(i+1)$st elements.

A slight variation of the parallel prefix algorithm can be used to assign to each node on the list the value $a_1 \, o \, \cdots \, o \, a_n$. The second (list expansion) phase is modified so as to broadcast the product computed in the first phase. We call this the *product broadcast* algorithm. For example, we can use this algorithm to label each node in a list with the name of the first node in the list, or with the name of the least node in the list. The product broadcast algorithm can also be run on a linked circular list, provided that $o$ is a commutative operation.

If the parallel prefix algorithm is applied to a union of disjoint lists, than it will compute initial prefixes in parallel in each list. Similarly, if the product broadcast algorithm is applied to a union of disjoint lists (circular lists), then it will assign to each node the product of the elements in the list it belongs to.

## 3. GRAPH ADJACENCY LIST CONSTRUCTION

The adjacency lists representation of a (directed) graph is particularly useful for many parallel graph algorithms. Unfortunately, this is not always the input format of a graph. The *graph adjacency list construction* problem has as input the list of the edges of the graph (presented as pair of nodes). The edges are stored in an arbitrary order in an array. One constructs an adjacency lists representation of the graph, that is an array $L[1..n]$ of linked lists, where $L[i]$ is the list of nodes adjacent to node $i$.

Constructing the adjacency lists for a graph with $n$ nodes and $e$ edges requires only $O(e+n)$ serial time and $O(e+n)$ space. The array $L[1..n]$ is initialized to nil in time $O(e)$. The adjacency lists are then constructed in time $O(e)$ by inserting edges successively in the adjacency lists.

The parallel version using $p$ processors is not so simple. When the graph is dense, i.e. $e=\Theta(n^2)$, a matrix can be used to avoid interference. When the graph is sparse one must be careful to insure that multiple processors do not try to add elements to the same adjacency list at the same time.

Assume w.l.o.g. that $p$ and $e$ are powers of 2, and that $p \leq n/2$. We create an adjacency list representation, where each entry in the array $L[i]$ has a pointer to the head and a pointer to the tail of the $i$th list. Each entry in the edge list will point to to the corresponding entry in an adjacency list. Each adjacency list is doubly linked.

If $p^2 \leq e$ then the edges are split into $p$ blocks each of size $e/p$. Each processor is assigned a block and it sequentially creates adjacency lists for the edges in its block. This has to be done with some care: since $e/p$ might be much less than $n$, one cannot afford to pay the $O(n)$ overhead of initializing the array entries to nil. Instead we initialize correctly only those entries that are going to be used. There are two passes over the data. In the first pass no insertions are made; for each edge, (the header of) the list the edge is to be inserted into is initialized to nil. In the second pass the insertions are actually made. We create a structure where nonempty adjacency lists have valid format, whereas empty adjacency lists may contain garbage. This phase requires $O(e/p)$ time and $O(pn+e)$ space.

Next all the partial adjacency lists are linked up: We start with an empty structure and serially link to it the adjacency lists of the $p$ blocks, one at a time. All $p$ processors always process the same block at the same time. As above this is done in two passes. In the first pass each processor picks an edge, and if this edge is the first in its list then the processor initializes the corresponding array entry in the new structure to nil. In the second pass each processor picks an edge, and if this edge is the first in its list then the processor links that list at the head of the corresponding adjacency list in the new structure. Since each block structure contains one entry per node there are no conflicts.

The time to process a block in the second phase is $O(e/p^2)$. There are $p$ blocks, so the total time for the second phase is $O(e/p)$. The total time for the algorithm is $O(e/p)$ and the space is $O(pn+e)$.

If $p^2 > e$ then the edges are split into $e/p$ blocks of size $p$. The algorithm is recursively applied to each block in parallel, with $p^2/e$ processors per block. Note that $p^2/e$ and $p$ are powers of 2. Next the $e/p$ partial adjacency lists are linked up, with each block being processed one at a time by the $p$ processors. The same two-pass method is used to avoid initialization overheads. This part of the algorithm takes $O(e/p)$ time and $O(pn+e)$ space.

Let $T_p(e,n)$ be the running time of this algorithm, and let $S_p(e,n)$ be the space used by the algorithm. Then if $p^2 \leq e$,

$$T_p(e,n) = O(e/p) .$$

If $p^2 > e$, we get

$$T_p(e,n) = T_{p^2/e}(p,n) + O(e/p) .$$

This recursion solves to

$$T_p(e,n) = O(\frac{e}{p}\frac{\log e}{\log(e/p)}) .$$

It is easy to modify the above algorithm to set the empty list headers to nil at an additional expense of $O((n+e)/p)$ time.

There are at most $p$ graph adjacency list construction tasks running in parallel, with a total number of edges $e$. Thus, the space used is $O(pn + e)$.

**Theorem 3.1:** The edge list to adjacency list conversion problem can be solved on an EREW machine with $p \leq e/2$ processors in $O(\frac{e}{p}\frac{\log e}{\log(e/p)} + \frac{n}{p})$ time and $O(pn+e)$ space.

**Note:** This yields a linear speedup when $p \leq e^{1-\epsilon}$ for any constant $1>\epsilon>0$; the space requirement is no more than for an adjacency matrix.

**Corollary 3.2:** A list of $e$ numbers in the range $1..n^k$ can be sorted with $p \leq e/2$ processors in time $O(k(\frac{e}{p}\frac{\log e}{\log(e/p)} + \frac{n}{p}))$ and space $O(np + e)$.

**Proof:** Assume $k=1$. For each key value $i$ in the range $1..n$ we create a linked list containing the items with key value $i$. This is done in the same manner an adjacency list was created in the previous algorithm. It requires time $O(\frac{e}{p}\frac{\log e}{\log(e/p)} + \frac{n}{p})$.

We then pack the list of nonempty buckets, in time $O(n/p + \log p)$. A sorted list can now be created in time $O(n/p)$.

The result for general $k$ is obtained by running $k$ phases of a radix sort algorithm, with radix $n$. □

**Corollary 3.3:** A list of $m$ numbers in the range $1,...,R$ can be sorted with $p \leq m$ processors in time $O(\frac{m}{p}\frac{\log R}{\log(m/p)})$.

**Proof:** We use the previous corollary, with $R = n^k$. The time is minimized when $n \approx \frac{m}{p}\frac{\log m}{\log(m/p)}$. Thereby, the sort takes time $O(\frac{m}{p}\frac{\log R}{\log(m/p)})$. □

The sorting time is optimal when $p \leq R^{O(1)}$.

**Corollary 3.4:** Given an adjacency lists representation of a graph with $e$ edges and $n$ nodes, the adjacency lists representation of the reversed graph (i.e. the graph where the direction of each edge is reversed) can be computed in time

$O(\frac{e}{p}\frac{\log e}{\log(e/p)} + \frac{n}{p})$.

These results can also be used to compute the composition of mappings. Let $f$ be a function with domain $1...m$ and range $1...n$, and let $g$ be a function with domain $1...n$. Assume that $f$ is given as a (not necessarily sorted) list of pairs $<i,f(i)>$, and $g$ is represented by an array of values, with the $i$th entry storing $g(i)$. We compute a representation of $g\,o f$ as a list of pairs $<i,g(f(i))>$ using the following steps.

(1) The pairs $<i,f(i)>$ are stored in bucket lists, indexed by the value of the second coordinate $f(i)$.

(2) The first element of each list is marked.

(3) Each marked element $<i,j>$ $(j=f(i))$ performs a look-up for the value of $g(j)$, and stores this value.

(4) The value of $g(j)$ is broadcast to all elements in the bucket, using the product broadcast algorithm.

Empty buckets are never accessed, so that it is not necessary to initialize them. We obtain

**Theorem 3.5:** The composition problem can be solved with $p$ processors on an EREW machine in time $O(\frac{m}{p}\frac{\log m}{\log(m/p)})$ and space $O(np+m)$.

If the function $g$ is also represented by an unsorted list of pairs, the the composition can be computed in time $O(\frac{m+n}{p}\frac{\log(m+n)}{\log((m+n)/p)} + \frac{n}{p})$, using the composition algorithm given in [11].

## 4. TREE PROBLEMS

### 4.1. Tree Recursions

The parallel prefix algorithm can be extended to "parallel prefix" computations on a tree. This has been independently noted for mesh connected computers [1,13].

Let $T$ be tree represented by its adjacency list (each child has an edge directed to its parent). For each node $u$ of $T$ let $val(u)$ be a value stored at that node. Let

$$F(u) = \sum_{v \text{ descendant of } u} val(v)$$

(where a node is taken to be descendant of itself). The addition is an arbitrary associative and commutative operation. The function $F$ can also be defined by the recursion

$$F(u) = \sum_{v \text{ child of } u} F(v) + val(u)\ .$$

We wish to compute the value of $F$ at each node of the tree. The parallel prefix problem is a particular case of this general problem, when the tree degenerates into a linked list. Other interesting cases are

(1) Computing $d(u)$, the number of descendants of each node, including itself. We take $val(u)$=1, and + is usual addition.

(2) Computing the path length of the tree. We take $val(u)=d(u)$, and + is usual addition.

(3) Computing the height of each node in the tree. We have $val(u)$=1, and + is the minimum operation.

An efficient parallel algorithm for solving tree recursions has the following form

**General Parallel Tree Recursion Algorithm**

```
if Tree contains more than one element then
begin
   Pick a set $S$ of nodes such that
     (1) no two nodes in $S$ are adjacent or have
           the same parent
     (2) each node of $S$ has at most one child;
   for each node $u \in S$ do
   begin
      add the value of $u$ to the value of $parent(u)$;
      delete $u$;
      if $u$ has a child then
     link the child to $parent(u)$
   end;
   execute algorithm recursively;
   for each node $u \in S$ do
   begin
      link $u$ to $parent(u)$;
      if $u$ has a child then
      begin
         add the value of $child(u)$ to $u$;
         link $child(u)$ to $u$
      end
   end
end
```

It is easy to see that this algorithm solves the tree recursion problem correctly. One has to show that a set $S$ containing a fixed fraction of the nodes can be picked at each iteration, while spending only constant time per node.

Assume that the tree $T$ has degree bounded by two. We shall divide each iteration in the execution of the algorithm in two stages, one where left children are active, the second where right children are active. This guarantees that children of the same parent do not conflict. Conflicts between a child and its parent are handled exactly as in the parallel prefix algorithm, by dividing the edges into blocks that are handled one at a time.

At least half of the nodes of $T$ have no more than one child. At least 1/3 of the nodes with no more than one child are paired at each iteration. We thus obtain that the recursion can be solved with $p$ processors for a binary tree with $n$ nodes in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ in the EREW model.

Let $T$ be now a general tree. The tree $T$ can be mapped onto a binary tree $T^b$, as described in [6, § 2.3.2]. The mapping is readily computed from the adjacency list representation of $T$, with edges going from parent to child: $v := left(u)$, if $v$ is the first child in the adjacency list of $u$; $v := right(u)$ if $v$ follows $u$ in the adjacency list of their common parent.

Let $F^b$ be the function computed on the binary tree $T^b$ by solving the tree recursion problem on this tree. Then $F^b(u)$ is the sum of $val(v)$, taken over all descendants of $u$ and descendants of elder siblings of $u$. We have

$$F(u) = \begin{cases} F^b(u) & \text{if } right(u)=nil \\ F^b(left(u)) + val(u) & \text{otherwise} \end{cases}$$

Thus, $F$ can be easily computed from $F^b$. We obtain

**Theorem 4.1:** Given an adjacency list representation of a valued tree with $n$ nodes, the tree recursion problem for the tree can be solved on an EREW machine with $p$ processors in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ for $p<n$.

### 4.2. Tree Traversals

Given a tree with $n$ nodes represented by an adjacency list (with edges going from parent to child) a linear linked list of the nodes arranged in preorder can be produced by $p$ processors in time $O(n/p)$. A similar construction is given by Wyllie [15] for binary trees.

Let $u_L$ and $u_R$ be two copies of the node $u$ in the tree. Define the mapping *next* as follows.

$$next(u_L) = \begin{cases} v_L & \text{if } v \text{ is first child of } u \\ u_R & \text{if } u \text{ has no children} \end{cases}$$

$$next(u_R) = \begin{cases} v_L & \text{if } v \text{ follows } u \text{ in the adjacency list of } parent(u) \\ parent(u)_R & \text{if } u \text{ is last child of } parent(u) \end{cases}$$

This mapping corresponds to a traversal of the tree in the order Root-Children-Root ($u_L$ represents the first traversal of $u$, and $u_R$ represents the second traversal).

Each node of the tree (with the exception of the root) occurs exactly once in an adjacency list. The linked list defined by the relation *next* can be created by $p$ processors in the time it takes to traverse a linked list in parallel: Each processor is assigned $n/p$ entries from the adjacency list; the processor creates those nodes of the list corresponding to the nodes of the tree it was assigned. The linked list traversal is required for each last child to determine its parent.

A preorder list of the nodes can be obtained in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ from this list by deleting all the "$u_R$" nodes, and shrinking the list accordingly. This is done by running the parallel product algorithm.

Applying parallel prefix to this list will provide the preorder number of each node with linear speedup for $p \leq n^{1-\epsilon}$.

If we apply this algorithm to a forest then we shall obtain a preorder list for each tree in the forest, and compute the preorder number of each node in the tree it belongs to. Postorder on trees or forests and inorder on binary trees can be handled in a similar manner. Note too that the adjacency list of a tree can be recreated from the list defined by the *next* relation in time $O(n/p)$.

### 4.3. Unicycular Graphs

In this section we show how a "unicycular" graph can be converted to a forest with the same connected components. Although algorithms on unicycular graphs are not by themselves particularly important, this routine is useful in later sections. A *unicycular graph* is a tree with one cycle; it consists of a set of $n$ nodes, each with a parent pointer to another node in the set. Since each node in a unicycular graph has a unique parent, one can define on such a graph the mapping *next* used in the last section.

**Lemma 4.2:** Let $G$ be a unicycular graph, and let *next* be the mapping defined above. Let $G_{next}$ be the graph defined by this mapping: The nodes of $G_{next}$ consist of two copies of each node of $G$, and $uv$ is an edge of $G_{next}$ iff $v = next(u)$. Then

(1) The graph $G_{next}$ consists of exactly two cycles.

(2) The nodes $u_L$ and $u_R$ occur in distinct cycles of $G_{next}$ iff $u$ is on the cycle of $G$.

**Proof:** Each node has exactly one successor and one predecessor in $G_{next}$, so that this graph is a union of cycles. The node $u$ is on the cycle of $G$, iff a traversal starting at $u$ reaches $u$ again before it backtracks. Hence $u$ is on a cycle of $G$ iff the path in $G_{next}$ starting from $u_L$ returns to $u_L$ before it reaches $u_R$. The graph $G$ can be transformed in a tree by deleting one edge from its cycle. This causes the deletion of one edge, and the change of one edge in $G_{next}$, thereby obtaining a linear graph. It follows that $G_{next}$ contains exactly two cycles. □

Let $G$ be a graph that is the union of disjoint unicycular components. The graph $G'$ is a *forest decomposition* of $G$ if $G'$ is obtained from $G$ by deleting one edge from each cycle, thereby replacing each unicycular component with a tree.

**Theorem 4.3:** Let $G$ be a graph with $n$ nodes that is the disjoint union of unicycular components, represented by its adjacency list. The adjacency list of a forest decomposition of $G$ can be computed with $p$ processors in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ in the EREW model.

**Proof:** The following algorithm will do.

(1) Construct the linked structure defined by the *next* relation.

(2) Assign a distinct label to each circular list (e.g. the least name, in lexicographic order, of an item in the list); label each item with the label of the list it belongs.

(3) Mark those items $u_L$ such that $u_R$ occurs in a list distinct from the list containing $u_L$.

(4) Mark on each list the least item $u_L$ that was marked in the previous phase.

(5) Delete from the original adjacency list the edge leading to $u$, for each item $u_L$ that was marked at the previous stage.

Each of these phases can be implemented to run within the required time. □

## 5. SPANNING FOREST AND CONNECTED COMPONENTS

Connected components can be derived from a spanning forest by creating a preorder list for each tree, and marking each node with the label of the first node in its list. This can be done with $p < n/2$ processors in the EREW model in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$.

Our algorithms for finding a spanning forest are in the spirit of most of the previous algorithms for this problem: (super) nodes are continually combined into super nodes. When no more combining is possible, each super node will represent a connected component, and a spanning tree will also exist for each component.

Eckstein [5] has shown that a spanning forest can be found in $O(e/p + n)$ time using either depth-first or breadth-first search [5]. This is optimal when the number of processors is small relative to the denseness of the graph, i.e. $p = O(e/n)$).

Let $G$ be an (undirected) graph with $e$ edges and $n$ nodes. Our algorithms use two auxiliary graphs: the graph $G_f$ is a forest that is a subgraph of $G$; the graph $G_s$ consists of the supernodes of $G$. Each connected component (tree) of $G_f$ is represented by one supernode in $G_s$. Two supernodes are connected if the corresponding trees are connected by an edge in $G$. Initially $G_f$ contains all the nodes of $G$ and no edges; $G_s$ is initially identical to $G$. At each iteration supernodes that are connected in $G_s$ are combined into a new supernode; edges are added to $G_f$ to merge the corresponding trees into a new tree. The algorithm terminates when $G_s$ does not contain any more edges. At that point $G_f$ is a spanning forest for $G$.

The graph $G_f$ is represented by marking in the adjacency list of $G$ those edges that belong to $G_f$. Separate adjacency lists represent $G_s$. In order to achieve the desired running time, the number of accesses to each edge entry in this list must be bounded by a constant. Therefore, it is not possible to update the edge entries at each iteration. Instead, each edge in an adjacency list of a supernode of $G_s$ is represented by an entry with the name of the incident node in $G$. Some of the edges in this adjacency list may be self loops, i.e. entries in the list of a supernode with the name of a node belonging to this supernode.

We keep a membership list for each supernode, i.e. a list of the nodes belonging to each supernode. Each supernode has a *weight* count, which is the number of nodes belonging to it. We also keep an inverse directory, i.e. an array that indicates for each node the name of the supernode it belongs to.

At the end of each iteration we pack the adjacency list of $G_s$ so that nodes with nonempty lists of incident edges are in the first locations. As we do not want to modify the edge entries, the packing is actually done on a separate array of pointers to the supernodes.

The initial adjacency list for $G_s$ can be created in $O((n+e)/p)$ steps; it can be packed in time $O(n/p + \log p)$. The membership list and inverse directory for supernodes can be created in $O(n/p)$ time.

We terminate the "graph compaction" iterations when the number of nonisolated supernodes is $O(\max(n/p, p^{1+\epsilon}))$. We can then use Eckstein's algorithm to terminate in time $O(n/p + p^{1+\epsilon} + e/p)$.

Assume that the number of supernodes with nonempty adjacency lists is at least $p^{1+\epsilon}$. At each iteration we serially perform the following steps.

(a) Pick the first edge from each nonempty adjacency list of $G_s$. Delete these edges from the graph $G_s$.

(b) Delete from this set of edges those edges that are self loops.

(c) Create an adjacency list for the graph consisting of the remaining edges, and their endpoints. This graph consists of a union of unicycular components.

(d) Create a generating forest $F$ for the graph, by deleting edges from the new adjacency list. Each tree in this forest will be replaced by one new supernode.

(e) Mark the edges of $F$ as belonging to $G_f$.

(f) Select in each tree of $F$ the supernode of largest weight (this is the new supernode), and label all the supernodes in the tree with the name of the selected supernode. Mark all supernodes with the exception of the new supernodes $F$ as inactive.

(g) Update the membership list and inverse directory for supernodes.

(h) Link the adjacency lists of supernodes marked inactive in (f) to the adjacency list of the new supernode they belong to.

(i) Pack the list of supernodes with nonempty adjacency lists.

It is easy to check that this iteration performs a valid compaction of the graph. We shall now estimate the running time of each step. Let $a$ be the number of active supernodes (i.e. supernodes with nonempty adjacency lists) at the start of the iteration.

The edges picked in (a) are <supernode,node> pairs. We compute the <supernode,supernode> edge represented by this pair by composing the mapping represented by the pairs picked in (a) with the mapping represented by the supernode inverse directory. This is done in time $O(\frac{a}{p}\frac{\log a}{\log(a/p)}) = O(a/p)$. Self loops can then be deleted in time $O(a/p)$.

Thus, step (b) can be performed in time $O(a/p)$. It is easy to see that each of the remaining steps, with the possible exception of step (g) can also be executed within the same time bound. As $a$ edges are deleted from $G_s$ at this iteration, it follows that the total time spent executing these steps is $O(e/p)$. We shall now estimate the time spent in updating membership lists and inverse directories for supernodes.

Let $m$ be the number of nodes that change supernode during the iteration. The number of affected supernodes is bounded by $a$. Using the membership list it is possible to create a list of affected nodes in time $O(a/p)$. This can be used to update the inverse directory in time $O(m/p)$. The membership list (and the weights associated with it) is updated by moving the member lists of inactivated supernodes in time $O(a/p)$

Let $u_i$ be the number of times node $i$ changes supernode membership. The last discussion shows that the total amount of time spent while executing step (g) is $O(\frac{e}{p} + \frac{1}{p}\sum_{i=1}^{n} u_i)$.

Since smaller supernodes are merged into larger ones, the size of the last supernode node $i$ belongs to is at least $2^{u_i}$. When the last merge is performed we have more than $n/p$ components. This implies the inequality

$$\sum_{i=1}^{n}\frac{1}{2^{u_i}} > \frac{n}{p} .$$

The maximum of $\sum_{i=1}^{n} u_i$ under the above constraint is equal to $n\ lg\ p$. Thus

$$\sum_{i=1}^{n} u_i < n \log p .$$

Hence, the total time spent performing step (g) is $O(\frac{e}{p} + \frac{n \log p}{p})$.

It follows that the total time taken by the algorithm is $O(\frac{e}{p} + \frac{n \log p}{p} + p^{1+\epsilon})$.

**Theorem 5.1:** A spanning forest for a graph with $n$ nodes and $e$ edges can be computed by $p < n/2$ processors in the EREW model in time

$$O(\frac{e}{p} + \frac{n \log p}{p} + p^{1+\epsilon}).$$

and space

$$O(pn + e) .$$

**Corollary 5.2:** The connected components of a graph with $n$ nodes and $e$ edges can be computed with $p < (n+e)/2$ processors in the EREW model in time $O(\frac{n+e}{p}\frac{\log(n+e)}{\log((n+e)/p)})$ and space $O(pn+e)$.

### 5.1. Biconnected Components

Tarjan and Vishkin [14] developed a parallel algorithm to compute the biconnected components of a connected graph. Their algorithm, however, required that the graph be dense, the input already be in the form of an adjacency list, and that the model of parallel computation allow concurrent reads and writes. Given our algorithms to compute the spanning forest of a graph as well as the ability of construct preorder and postorder numberings and lists of a tree, we can use their method but improve the result:

**Corollary 5.3:** The biconnected components of a graph can be computed in $O(\frac{e}{p} + \frac{n \log p}{p} + p^{1+\epsilon})$ time, provided $p \leq n$.

**Proof:** The algorithm of Tarjan and Vishkin requires the computation of a spanning tree for the graph, the computation of the preorder number of each node in this spanning tree, and the solution of several tree recurrences. We can perform each of these operations within the claimed time bound. □

### 5.2. Minimum Spanning Tree

The spanning forest algorithm can be modified to give a minimum spanning tree algorithm. A sequential algorithm for minimum spanning tree merges supernodes iteratively. At each iteration a supernode is chosen; the least cost edge outgoing this supernode is used to merge it with another supernode. The order the supernodes are chosen is arbitrary (see [3]).

We can merge supernodes in parallel provided that the merging is consistent with some serial execution of the algorithm above. Let $G_s$, the supernode graph, and $G_f$, the spanning forest graph, be defined as in the previous algorithm. A general parallel algorithm that finds a minimum cost spanning tree for a connected graph $G = <V,E>$ has the following form.

```
General parallel algorithm for MST
$G_f = <V, empty>$; $G_s = G$;
while $G_s$ has more than one node do
begin
    1. pick least cost outgoing edge from each node of $G_s$;
       let $U$ be the subgraph of $G$ containing these edges
       ($U$ is the disjoint union of unicycular graphs);
    2. delete from each cycle of $U$ an edge with largest
       cost; let $F$ be the resulting graph ($F$ is a forest);
    3. add the edges of $F$ to $G_f$;
    4. combine supernodes of $G_s$ that belong to the same
       tree of $F$ into one supernode
end
```

The number of supernodes is at least halved at each iteration, so that at most $\lg n$ iterations are performed.

We implement this general algorithm using data structures similar to those used for the spanning tree algorithm: We keep an adjacency lists representation of $G_s$, and represent $G_f$ by marking edges in the adjacency list of $G$. We assume that $p < n, e$.

(1) A least cost edge is picked in each adjacency list of $G_f$ by running a product broadcast algorithm on the lists of edges. This can be done in time $O(\frac{e}{p}\frac{\log e}{\log(e/p)})$.

(2) A trivial modification of the algorithm of Theorem 4.3 can be used to delete from each cycle of $U$ an edge of highest cost (rather than an edge with least index, as done in the original algorithm). This is done in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$.

(3) The new edges can be added to $G_f$ in time $O(e/p)$.

(4) A supernode is selected from each tree of $F$, and each node of the tree is marked with the label of this supernode, by running a product broadcast algorithm in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$. The edges can be now updated to point to the new supernodes in time $O(\frac{e}{p}\frac{\log e}{\log(e/p)})$. This is done by computing the composition of two mappings: the mapping that maps an edge to the old incident nodes, and the mapping that maps these nodes to the new supernode they belong to. A new adjacency lists representation is built from these updated edges in time $O(\frac{e}{p}\frac{\log e}{\log(e/p)})$. While the new list is created self-loops can be deleted.

Each iteration takes time

$$O(\frac{e}{p}\frac{\log e}{\log(e/p)} + \frac{n}{p}\frac{\log n}{\log(n/p)}) .$$

Since there are at most $\lg n$ iterations we obtain

**Theorem 5.4:** A minimum cost spanning tree of a connected graph can be computed with $p \leq e, n$ processors on an EREW machine in time

$$O(\frac{(e+n)\log n}{p}\frac{\log(e+n)}{\log((e+n)/p)}) .$$

**Note:** This algorithm is efficient relative to Kruskal's algorithm provided that $p = O((e+n)^{1-\epsilon})$.

A minimum cost spanning forest for a general graph can be computed withing the same time bounds: the algorithm is modified so that supernodes with no outgoing edges are not visited. This is done by packing the list of nodes at each iteration.

## 6. CONCLUSION

Many simple and fast serial algorithms are often hard to parallelize. In many cases there is enough work that can be performed in parallel, but the challenge is to ensure that the processors do not conflict when performing this work, and that different processors do not replicate the same computation. The problem is harder when sparse structures are handled: If a compact data representation is used then the data layout is irregular, and it is hard to distribute work efficiently among processors. If, on the other hand, a regular data structure is used then superfluous work is performed.

We have presented techniques for working with sparse, irregular structures, and used these techniques to solve several important graph problems. We believe these techniques are generally applicable to other sparse problems. We still do not know whether large, extremely sparse graph problems can be solved efficiently, i.e. when $p << n$ and $e = \Theta(n)$.

## REFERENCES

[1] Mikhail J. Atallah and Susanne E. Hambrusch, Solving Tree Problems on a Mesh-Connected Processor Array, *26th Annual Symposium on Foundations of Computer Science,* pp. 222-231, Oct. 1985.

[2] B. Awerbuch and Y. Shiloach, New Connectivity and MSF Algorithms for Ultracomputer and PRAM, *Proc. of 1983 ICPP,* Aug. 1983, 175-179.

[3] D. Cheriton and R. E. Tarjan, Finding Minimum Spanning Trees, *SIAM J. Comput.,* 1976, 724-742.

[4] Richard Cole and Uzi Vishkin, Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking, *ACM Symposium on Theory of Computation,* 1986, to appear.

[5] D. M. Eckstein, Parallel Processing Using Depth-First and Breadth-First Search, Ph.D. Thesis, Univ. of Iowa, July 1977.

[6] D. E. Knuth, *The Art of Computer Programming. Vol 1 / Fundamental Algorithms,* Addison-Wesley, Reading Mass., 1973.

[7] C. P. Kruskal, L. Rudolph, and M. Snir, Efficient Parallel Algorithms for Graph Problems, International Conference on Parallel Processing, Aug. 1985.

[8] S. C. Kwan and W. L. Ruzzo, Adaptive Parallel Algorithms for Finding Minimum Spanning Trees, *Proc. of 1984 ICPP,* Aug. 1984, 439-443.

[9] R. Ladner and M. Fischer, Parallel Prefix Computation, *JACM,* 1980, 831-838.

[10] G. F. Lev, N. Pippenger, and L. G. Valiant, A Fast Parallel Algorithm for Routing in Permutation Networks, *IEEE Trans. on Computers,* Vol. C-30, Feb. 1981, 93-100.

[11] J. T. Schwartz, Ultracomputers, *ACM Transactions on Programming Languages and Systems,* Dec. 80, 484-521

[12] M. Snir, On Parallel Searching, *ACM Symposium on Distributed Computing,* Aug. 1982, 242-253

[13] Quentin F. Stout, Tree-Based Graph Algorithms for Some Parallel Computers, *1985 Intl. Conf. on Parallel Processing,* Aug. 1985, 727-731.

[14] Robert E. Tarjan and Uzi Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Time, 25th Annual Symposium on Foundations of Computer Science, Oct. 1984, 12-20.

[15] J. C. Wyllie, The Complexity of Parallel Computation, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

| PROBLEM | MODEL | TIME | LINEAR SPEEDUP | RESEARCHER |
|---|---|---|---|---|
| prefix (linked list), tree computations | EREW | $\frac{n \log n}{p}$ | — | Wyllie |
| | EREW | $\frac{\log n}{\log(2n/p)} \frac{n}{p}$ | $p \leq n^{1-\epsilon}$ | Kruskal, et al. |
| radix sort range $\{1,...,R\}$ | EREW | $\frac{\log R}{\log(n/p)} \frac{n}{p}$ | $p \leq n^{1-\epsilon}$ | *This paper* |
| graph conversion | EREW | $\frac{\log n}{\log(2n/p)} \frac{n}{p}$ | $p \leq n^{1-\epsilon}$ | *This paper* |
| spanning forest, connected components | CRCW | $\frac{e \log n}{p}$ | — | Awerbuch and Shiloach |
| | CREW | $\frac{e \log n}{p} + (\log n)(\log p)$ | — | Kwan and Ruzzo |
| | EREW | $\frac{e}{p} + \frac{n \log p}{p} + p^{1+\epsilon}$ | $p \leq e^{1/2-\epsilon}$<br>$\log p \leq \frac{e}{n}$ | *This paper* |
| | CREW | $\frac{e}{p} + \frac{n \log p}{p} + p \log p$ | $p \leq \sqrt{\frac{e}{\log e}}$<br>$\log p \leq \frac{e}{n}$ | *This paper* |
| biconnected components | CRCW | $\frac{e \log n}{p}$ | — | Tarjan and Vishkin |
| | EREW | $\frac{e}{p} + \frac{n \log p}{p} + p^{1+\epsilon}$ | $p \leq e^{1/2-\epsilon}$<br>$\log p \leq \frac{e}{n}$ | *This paper* |
| | CREW | $\frac{e}{p} + \frac{n \log p}{p} + p \log p$ | $p \leq \sqrt{\frac{e}{\log e}}$<br>$\log p \leq \frac{e}{n}$ | *This paper* |
| minimum spanning forest | CRCW | $\frac{e \log n}{p}$ | $p \leq e$ | Awerbuch and Shiloach |
| | CREW | $\frac{e \log n}{p} + (\log n)(\log p)$ | $p \leq \frac{e}{\log e}$ | Kwan and Ruzzo |
| | EREW | $\frac{e \log n}{p} + (\log p)^2$ | $p \leq \frac{e}{\log e}$ | *This paper* |

# A PARALLEL ALGORITHM FOR DOMINATORS [+]

**Shaunak Pawagi**
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

**P. S. Gopalakrishnan**
Department of Computer Science
University of Maryland
College Park, Maryland 20742

**I. V. Ramakrishnan**
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

**Abstract** -- We present a fast parallel algorithm for computing the dominators of a directed acyclic graph. The model of computation used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. Let $P_t(n)$ be the processor complexity of computing the transitive closure of an n-vertex directed acyclic graph on this model. Our algorithm for computing the dominators requires $O(\log^2 n)$[†] time using $O(P_t(n))$ processors. The only known parallel algorithm for this problem [6] requires $O(nP_t(n))$ processors. Our algorithm therefore improves the processor complexity of this algorithm by a factor of n, but has the same time complexity.

## 1. Introduction

Computing the dominators of a directed acyclic graph (DAG) is a very important code optimization step in compilers (see [2] for details). Consequently, this problem has attracted widespread attention and several excellent sequential algorithms have been developed for it (see [1]). Of late, growing interest in parallel computation has led to a proliferation of parallel algorithms for graph problems on a model of synchronous parallel computation [6, 7]. Surprisingly, despite the importance of the dominator problem the only known parallel algorithm for it is due to Savage [6]. This algorithm computes dominators using a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. We refer to this model as R-PRAM. A powerful variation of this model that allows simultaneous writes into the same memory location by more than one processor, is referred to as W-PRAM.

Let $G = (V, E)$ be a DAG rooted at r, and $|V| = n$, and $|E| = m$. We say that vertex i is a *dominator* of vertex j if i is on every path from r to j. For each vertex k, Savage first constructs a graph $G_k$, by deleting from G all edges leaving k. Next, the transitive closure of each of these graphs is computed in parallel. Let $G^*$ and $G_k^*$ denote the transitive closures of G and $G_k$ respectively. Dominators are then computed by using the simple observation that if i is reachable from r in $G^*$ and not in $G_k^*$, then k is a dominator for i. Savage's algorithm requires $O(\log^2 n)$ time and uses $O(nP_t(n))$ processors, where $P_t(n)$ is the processor complexity of computing the transitive closure. The two known parallel algorithms for doing so are due to Hirschberg [4] and Chandra [3], and require $O(n^3/\log n)$ and $O(n^{2.81}/\log n)$ processors respectively.

In the following section we will describe our algorithm for computing the dominators on a R-PRAM. Our approach is radically different from both Savage's as well as those used in sequential algorithms. The time complexity of our algorithm is $O(\log^2 n)$, and it uses $O(P_t(n))$ processors. Observe here that the processor requirement our algorithm has decreased by a factor of n over Savage's algorithm.

It is interesting to note that computing the transitive closure seems to be an important step in parallel algorithms for various properties of directed graphs, and hence their processor complexities are determined by that of the transitive closure algorithm. Our algorithm does achieve this bound for processor complexity. This is similar to computing properties of undirected graphs in parallel, where the processor complexities are determined by the connected component algorithm [7].

## 2. Preliminaries

We begin with a review of graph-theoretic terms.

Let $G=(V,E)$ denote a *graph* where V is a finite set of vertices and E is a set of pairs of vertices called *edges*. If the edges are unordered pairs then G is *undirected* else it is *directed*. Throughout this paper we assume that V consists of the set of vertices $\{1,2,...,n\}$ and $|E|=m$. We denote the undirected edge joining the vertices u and v by (u,v) and the directed edge from u to v by $<u,v>$. An *adjacency* matrix A of G is an $n \times n$ boolean matrix such that $A[u,v]=1$ if and only if $(u,v) \in E$. A *directed path* in G joining two vertices $i_0$ and $i_k$ is defined as a sequence of vertices $(i_0,i_1,i_2,..i_k)$ such that all of them are distinct and for each $0 \leq p < k$, $<i_p,i_{p+1}>$ is an edge of G. An undirected path is defined similarly. If $i_0 = i_k$ then the path is called a *cycle*. A *directed acyclic graph* (DAG) has no cycles. We denote a directed path from u to v by $[u \rightarrow v]$. We say that an undirected graph G is *connected* if for every pair of vertices u and v in V, there is a path in G joining u and v. A *tree* is a connected undirected graph with no cycles in it.

[+] The support of the first author by the Air Force Office of Scientific Research under Contract F-49620-85-K-0009, and of the second author by grants from the NSF to the Machine Intelligence and Pattern Analysis Laboratory, and of the third author by the Office of Naval Research under Contract N00014-84-K-0530, and by the National Science Foundation under grant ECS-84-04399, is gratefully acknowledged.

[†] Throughout this paper, we use log n to denote $\lceil \log_2 n \rceil$

A *rooted* directed tree has a distinguished vertex called *root* from which every other vertex is reachable via a directed path. We say that vertex u is an *ancestor* of vertex v if u is on the path from the root to v. The father of a vertex is its immediate ancestor. A *descendant* of a vertex is defined similarly. The *lowest common ancestor* (LCA) of vertices x and y in T is the vertex z such that z is a common ancestor of x and y, and any other common ancestor of x and y in T is also an ancestor of z in T.

A directed graph is *rooted* at r if there is a path from r to every vertex in V. For the rest of this paper, without loss of generality we shall assume that G is a directed acyclic graph rooted at r. We say that vertex i is a *dominator* of vertex j if i is on every path from r to j. In particular, for every i in V, r and i are dominators of i. Dominators exhibit transitivity, that is, for vertices i,j and k in V, whenever i is a dominator of j and j is a dominator of k, then i is a dominator of k. Therefore it is easy to see that the set of dominators of a vertex j can be linearly ordered by their order of occurrence on a path from r to j. The dominator of j closest to j (other than j) is called the *immediate* dominator of j. It follows from the definition that immediate dominator of every vertex is unique. We can now express the dominator relation as a directed tree $T_d$ rooted at r called the dominator tree. If u is the immediate dominator of v then <u,v> is an edge of $T_d$. Note now that i is a dominator of j if i is an ancestor of j in $T_d$.

The transitive closure matrix $A^*$ is a boolean matrix such that $A^*[i,j] = 1$ if there is directed path from i to j in G, else $A^*[i,j] = 0$. For completeness, we first describe the parallel algorithm for the transitive closure.

It has been observed in [4] that the transitive closure of a directed graph be computed in $O(\log^2 n)$ time on a R-PRAM by straightforward parallelization of the known sequential algorithm that is based on repeated multiplication of the adjacency matrix. In this parallel algorithm (see Algorithm 2.1 below) *and* and *or* operations replace multiplication and addition operations of an inner product step. We refer to this as the *and-or* multiplication of two matrices. The algorithm initializes the transitive closure matrix $A^*$ to the adjacency matrix A and then performs (log n) iterations of the and-or multiplication of $A^*$ by itself. The matrix DD is used as temporary storage for clarity.

```
// All steps involving i and j are executed for all i, j,
$1 \leq i \leq n$ and $1 \leq j \leq n$ //
1.   $A^*[i,j] := A[i,j]$     //Initialize//
2.   for t:=1 to log(n-1) do
         2a.  $DD[i,j] := \overset{n}{\underset{x=1}{or}} \{ A^*[i,j], A^*[i,k] \text{ and } A^*[k,j] \}$
2b.       $A^*[i,j] := DD[i,j]$
```

Algorithm 2.1

**Lemma 2.1:** The above algorithm computes the transitive closure $A^*$ in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors.

**Proof:** The proof is immediate from the steps (1) and (2) of Algorithm 2.1. If Chandra's [3] parallel algorithm for matrix multiplication is used in step (2a), then the processor complexity of this algorithm is $O(n^{2.81}/\log n)$. Throughout this paper we refer to the processor complexity of this algorithm as $O(P_t(n))$. □

We now describe our algorithm for computing dominators.

## 3. The Algorithm

In order to compute the dominator tree we first construct a spanning tree for G that is rooted at r. We then compute the set of dominators for each vertex in a matrix DOM such that DOM[i,j] = 1 if i is a dominator of j otherwise DOM[i,j] = 0. The computational steps are as follows.

1. Compute the transitive closure matrix $A^*$ for G. By Lemma 2.1, this computation requires $O(\log^2 n)$ time and uses $O(P_t(n))$ processors.
2. Construct a directed spanning tree $T_s$ from the adjacency matrix A and the transitive closure matrix $A^*$. This is done by specifying the father of each vertex i, the smallest vertex j such that <j,i> is an edge of G. This selection can be done in O(log n) time by assigning n processors to each vertex. Since there are n vertices in G, we need $n^2$ processors for this step.
3. For every vertex, mark all its ancestors in $T_s$ as its dominators. That is, set DOM[i,j] = 1, if i is an ancestor of j, else set DOM[i,j] to 0. Ancestor computation can be done in O(log n) time using $O(n^2)$ processors (see [7]). Initialization of the matrix DOM requires constant time and $O(n^2)$ processors.
4. For every vertex v, consider the non-tree edges incident on it. For all such edges <x,v>, compute the lowest common ancestor of x and v in $T_s$. Among these LCAs, select h(v) = LCA(u,v) be the vertex closest to the root r (h stands for highest). The lowest common ancestors for all vertex pairs can be computed in O(log n) time using $O(n^2)$ processors (see [7]). For each vertex v, h(v) can be determined in O(log n) time using n processors.
5. Now, the edge <u,v> provides a path from h(v) to v passing through u, other than the path present in $T_s$. Therefore all vertices on the path [h(v)→v] are not dominators of v. For all such vertices j, set DOM[j,v] = 0. Since the number of vertices on any path is at most n, we need $O(n^2)$ processors to do this step in constant time.
6. For every vertex y, if i is not a dominator of y then i is not a dominator of any vertex reachable from y. Therefore for every vertex j, i is not a dominator for j if there exists at least one vertex x, such that i is not a dominator for x and j is reachable from x. Therefore set DOM[i,j] = 0, if

$$\mathop{or}_{x=1}^{n} \{ (\text{DOM}[i,x] = 0) \text{ and } (A^*[x,j] = 1) \}$$

evaluates true. This computation is equivalent to *and-or* multiplication of DOM and $A^*$. Therefore this step requires O(log n) time and $O(P_t(n))$ processors.

This completes the description of our algorithm for dominators. We now provide the proof of its correctness.

**Lemma 3.1:** If vertex u is not a dominator of vertex v then at the end of our algorithm DOM[u,v] will be set to 0.

**Proof:** If u is not on the path from r to v in $T_s$ then DOM[u,v] is set to 0 in step 2 of our algorithm and it stays 0 for all following steps. If u is on the path from r to v in $T_s$ but it is not a dominator of v then there must exist a path from r to v that does not pass through u. There are two types of paths to be considered. First, v has a non-tree edge <x,v> incident on it such that u is on the path from the LCA(x,v) to v. In step 5 of our algorithm, we select h(v) to be the closest LCA to the root r. Therefore u must be on the path [h(v)→v], and DOM[u,v] will be set to 0. Second, v is reachable from some vertex y such that y has a non-tree edge incident on it, providing another path to y from r that does not use u. In step 5 of our algorithm DOM[u,y] will be set to 0, and since v is reachable from y, in the next step, DOM[u,v] will be set to 0. Hence the Lemma. □

**Theorem 3.1:** The above algorithm computes the dominator matrix DOM in $O(\log^2 n)$ time using $O(P_t(n))$ processors.

**Proof:** The correctness of our algorithm is proved in Lemma 3.1 and the processor and time complexities are immediate from steps 1 to 6 of our algorithm. □

Given the matrix DOM, the dominator tree can be constructed by determining the immediate dominator for each vertex. Recall that the immediate dominator of a vertex is unique, and it is the closest dominator of that vertex. In the dominator tree, the closest dominator of a vertex is its father. The steps for construction of the dominator tree are as follows.

1. For every every vertex i, count the number of dominators of i by summing the entries in the $i^{th}$ column of DOM. This summation requires $n^2$ processors and O(log n) time.
2. For every vertex i, determine the immediate dominator of i. If i has d dominators then the immediate dominator of i is a dominator of i that has (d-1) dominators. This can be done in constant time using $n^2$ processors.
3. The father of every vertex i in the dominator tree is its immediate dominator. The root of the tree is r.

The above procedure constructs the dominator tree from the matrix DOM in O(logn) time using $n^2$ processors. The time and processor complexities are immediate from the steps given above, and the correctness is direct from the following lemma.

**Lemma 3.2:** Let d be the number of dominators of i. A dominator j of i is the immediate dominator of i iff j has (d-1) dominators.

**Proof:** The number of ancestors of any vertex in the dominator tree is equal to its dominators. Since the dominator tree is unique, the immediate dominator, which is the father of i in the dominator tree, must have (d-1) dominators. □

## 4. Conclusions

We have described an $O(\log^2 n)$ time algorithm for dominators of a DAG that uses $O(P_t(n))$ processors, where $P_t(n)$ is the processor complexity of computing the transitive closure of a directed graph. This improves the processor complexity of Savage's [6] algorithm, by a factor of n. Finally, it is worth mentioning that the algorithm presented here would require O(log n) time on a W-PRAM that allows simultaneous writing in the same memory location by more than one processor. The only step in our algorithm that requires $O(\log^2 n)$ time is computing the transitive closure matrix. All other steps require O(log n) time. Since the transitive closure can be computed in O(log n) time on a concurrent write model [5], our algorithm would therefore require O(log n) time on a W-PRAM.

## References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "*The Design and Analysis of Computer Algorithms*", Addison-Wesley, Reading, Mass., 1974.

[2] A. V. Aho and J. D. Ullman, "*Principles of Compiler Design*", Addison-Wesley, Reading, Mass., 1977.

[3] A. K. Chandra, "Maximal Parallelism in Matrix Multiplication", RC 6193, IBM Rept., 1976.

[4] D. Hirschberg, "Parallel Algorithms for the Transitive Closure and the Connected Component Problem", *Proc. Eighth STOC*, 1976, pp. 55-57.

[5] L. Kucera, "Parallel Computation and Conflicts in Memory Access", *Inf. Proc. Letters 14* (1982), pp. 93-96.

[6] C. Savage, "Parallel Algorithms for Some Graph Problems", TR 784, Dept. of Mathematics, Univ. of Illinois, Urbana, 1977.

[7] Y. Tsin and F. Chin, "Efficient Parallel Algorithms for a Class of Graph-Theoretic Problems", *SIAM J. Comp. 14* (1984), pp. 580-599.

# A Graph Model for Parallel Computations Expressed in the Computation Structures Language*

Ashok K. Adiga & James C. Browne
Department of Computer Science
University of Texas at Austin,
Austin, Texas 78712

## Abstract

The design of parallel computations involves numerous decisions whose effect on execution efficiency is not immediately clear. A Petri Net based model is presented which has the representational power to model a wide range of computations while retaining the flexibility to alter configuration parameters with ease. The model can be used to predict optimal values for configuration parameters such as degree of parallelism and task granularity on the efficiency of the computation.

## 1 Introduction

This paper describes a model for representing and evaluating the performance of parallel computations. An implementation of the model is being used to analyze the behavior of programs expressed using the Computation Structures Language (CSL), a language designed and implemented at the University of Texas.

The model can be used as a vehicle for studying the configuration space of a computation. A parallel computation [1] consists of a collection of programs (called *tasks*) and the specification of dependencies between them. The dependencies could be simple synchronization or sequencing relationships, constraint relationships such as mutual exclusion for shared resources, or communication relationships signifying data transfer between two or more tasks. The performance of a parallel computation is therefore architecture dependent, since different architectures would support these dependencies with varying degrees of efficiency. The basis for a model for parallel computations is to be able to predict the efficiency of execution of a parallel computation on a given architecture, and to study the effect of varying different configuration parameters of the computation. The configuration space of interest includes the following parameters:

- shared memory/ message model: the interaction between the tasks of a computation can be specified using a message model, a shared memory model, or a mixture of the two. This choice is based on the support provided by the host architecture and the volume and size of the information being exchanged.
- granularity of a unit: the efficiency of the computation is directly dependent on the size of each schedulable unit of computation. Larger units would usually reduce the amount of parallelism possible, while smaller units could entail additional overheads of data movement.
- computation structure: this includes the specification of synchronization and sequencing of the units composing the computation.
- underlying or host architecture: all the earlier parameters are implicitly dependent on the choice of the host architecture.

The model described here was initially designed to represent computations specified using the Computation Structures Language (CSL) ([2], [3]). CSL is a language that allows specification and programming of multitype, multiphase parallel computations. It supports dynamic structuring of computations through multiple phases, each of which may display different types and degrees of parallelism and differing requirements for sharing of data and interprocess communication. The model is a direct abstraction from CSL. Each instance of a computation expressed in CSL has an equivalent representation using our model. CSL programs are directly derivable from the Petri Net model and vice versa.

The proposed model is based on Petri Nets with extensions to allow the representation and performance evaluation of parallel computations. The extensions are predicated upon a performance evaluation viewpoint rather than a theoretical view of Petri Nets. Performance statistics for a computation are achieved by first expressing the computation in terms of the model, and then simulating the behavior of this net instance. The choice of a Petri Net based model allows most of the communication and synchronization primitives of the computation to be directly modeled. Any non-determinacy in the computation can also be easily captured in the model. Extensions are provided to allow dynamic structuring and scalability of the modeled computation.

The rest of this paper is organized as follows. Section 2 gives a brief introduction of Petri Nets and highlights some of the previous work in this area. Section 3 contains the definition and details of the proposed model, while the implementation is briefly described in Section 4. The

* This research was sponsored by the Department of Energy under grant number DE-AS05-81ER10987 and grant number DE-FG05-85ER25010.

utility of the model is demonstrated by means of an example in Section 5.

## 2 Petri Nets

Informally, a Petri Net [4] consists of two sets of vertices (called *places* and *transitions*) in a bipartite, directed graph. Arcs connect places to transitions or transitions to places. If an arc exists from a place to a transition, the place is an *input* place for that transition. Similarly, if an arc exists from a transition to a place, the place is an *output* place for that transition. Places can contain *tokens*. One interpretation given to Petri Nets is that places represent conditions, transitions represent actions, and the presence of a token in a place represents the presence of that condition. The set of input places for a transition signifies the conditions that should be present for that event to occur. The occurrence of an event is represented by the *firing* of a transition. For a transition to fire, all its input places must contain tokens, and upon firing, the transition removes one token from each of its input places and puts one token each in its output places. The state of the net is simply the distribution of tokens in the places of the net. Petri Nets are represented graphically as shown in Fig. 1 with the places shown as circles, the transitions as bars, and the tokens as dots.

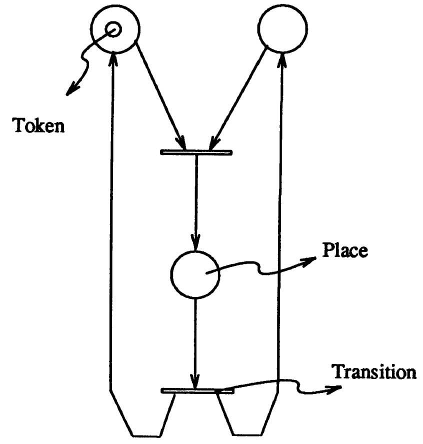


**Figure 1:** A Petri Net

Extensive work has been done to include timing information in Petri Nets. The most common addition is that of associating delays with transitions. When a transition begins to fire, it removes its input tokens as before, but places tokens in its output places only after a period of time equal to its delay value. E-nets [5] had fixed delays associated with transitions, Timed Petri Nets [6] allowed arbitrary delays; while extensions to their model in [7] allowed state dependent delays. While these models are suitable for modelling parallel systems, they lack the ability to model data-dependent control flow with ease.

An extension was proposed by Keller [8] to help model this control flow. His model included a set of program variables. Each transition had a transition procedure and predicate based on the program variables. In addition to requiring tokens on all input places, a transition was now considered enabled only if its predicate was true. Also, when a transition fired, its procedure was executed, thus altering the program variables. This model was designed for verifying correctness properties of parallel algorithms, and did not include the notion of time.

The model proposed in this paper attempts to combine these ideas, while also including additions to support features such as dynamic structuring and scalability of computations.

## 3 PCM: A model for parallel computations

This section contains a description of the model, along with some details of the simulation procedure used to do the performance evaluation.

### 3.1 Definition

The Parallel Computation Model (PCM) is defined by the three-tuple

$$\mathrm{PCM} = (\ \mathrm{PN},\ \mathrm{V},\ \mathrm{TA}\ )$$

where

PN = (P, T, I, O), a Petri Net, where
  $P = \{p_1,\ldots,p_n\}$, a set of places, $n \geq 0$
  $T = \{t_1,\ldots,t_m\}$, a set of transitions, $m \geq 0$
  $I \subseteq P\mathrm{x}T$, the transition input function
  $O \subseteq T\mathrm{x}P$, the transition output function
where P and T are disjoint

$V = \{v_1, v_2, \ldots, v_r\}$, a set of program variables

TA = is a three-tuple $<\Pi,\ \Phi,\ T>$;
  and
    $\Pi(V) = \{\pi_1(V),\ \pi_2(V),\ldots,\pi_m(V)\}$,
      a set of predicates,
    $\Phi(V) = \{\phi_1(V),\ \phi_2(V),\ldots,\phi_m(V)\}$,
      a set of procedures,
    $T(V) = \{\tau_1(V),\ \tau_2(V),\ldots,\tau_m(V)\}$,
      a set of delays.

The basic underlying structure is the standard Petri Net PN. In addition, the model allows a set of program variables (V), and attributes associated with each transition.

Each transition $t_i$ in the model defined above has a corresponding predicate $\pi_i(V)$ and a transition procedure $\phi_i(V)$. The predicates are defined on program variables, and are used to enable transitions under the modified firing rules. The transition procedures also operate on elements of V and can be used to modify them. The function $\tau_i(V)$ specifies the delay associated with transition $t_i$, and can also be a function of the set of program variables.

### 3.2 Firing Rules

The definition of the model is completed by specifying the modified firing rules. Each place in the net is conceptually divided into an *enable* and a *hold* region (Fig. 2). This division helps in describing the behavior of the net. A transition can be in one of three states: ***disabled***, ***enabled***, and ***firing***. The rules governing transition state changes are as follows.

- A disabled transition $t_i$ is enabled if each of its input places contains at least one token in its enable region and the predicate $\pi_i(V)$ corresponding to the transition is true.
- An enabled transition $t_i$ enters the firing state by moving one token in each of its input places from the enable to hold region. If this transition shares input places with other enabled transitions, they must satisfy rule 1 to remain enabled, failing which they become disabled.
- A transition $t_i$ remains in the firing state for a period of time specified by its delay function $\tau_i(V)$. A the end of this period, it removes one token from the hold region of each of its input places, and places a token in the enable region of each output place, and executes its associated procedure $\pi_i(V)$. It then returns to the disabled state.

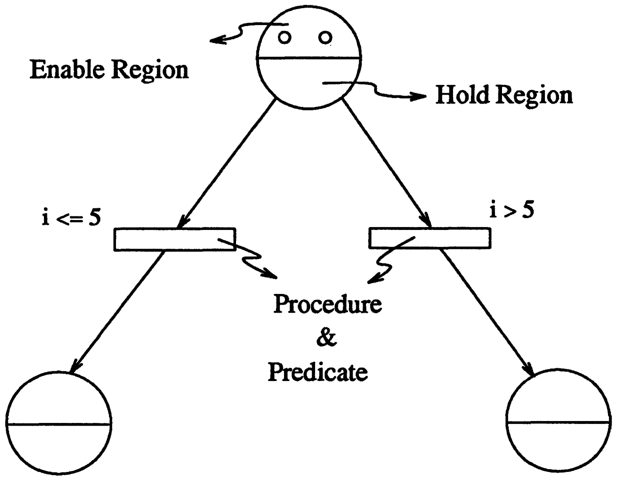


**Figure 2:** Execution of Net

Transition firings are no longer instantaneous, as in the case of standard Petri Nets. When a transition starts firing, it removes its enabling tokens , which can no longer contribute to the firing of any other transition. Tokens appear on the transition's output places only after a certain delay. There are two ways in which the firing of a transition can effect other transitions. When a transition starts firing, the removal of tokens from its input places could cause other enabled transitions to become disabled. Also, when a transition completes firing, it modifies the marking as well as the state of the program variable vector, thereby causing other transitions to be either enabled or disabled.

Transition predicates, procedures and program variables provide the means to represent standard looping and branching constructs with ease. Figure 2 shows the branching construct where the enabling of the two transitions is dependent on the current value of the program variable *i* - both transitions can not be enabled simultaneously.

The state of this system at any time is given by the current marking along with the vector of program variables and the vector of 'remaining firing times' for the transitions. The marking expresses the 'control' state, while the program vector specifies the 'data' state. If a transition is in the process of firing, its remaining firing time specifies the period of time after which firing will be complete. If the transition is not firing, its RFT is zero.

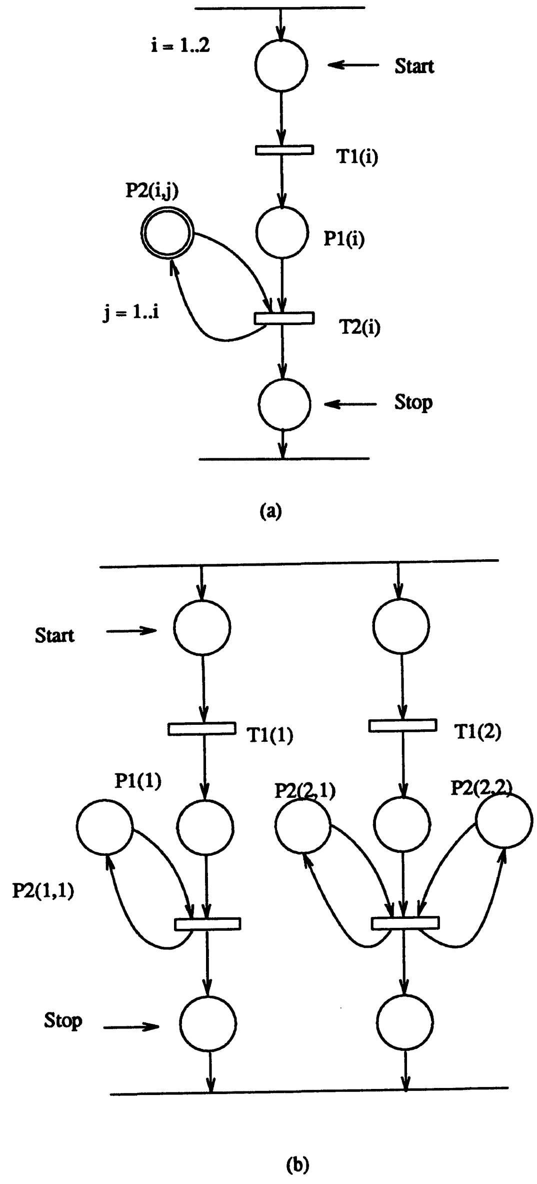
### 3.3 Hierarchical Modelling

PCM supports hierarchical modelling by allowing transitions to be designated as subnets. A subnet transition is constrained to begin and end with special places called the start and stop places (Figure 3a). When the subnet transition begins firing, a token is placed in its start place, and the subnet becomes active. When a token appears in the stop place, the subnet ceases to be active. Only transitions in active subnets can fire..

In addition, subnets can be parameterized in two ways. A subnet can have a set of parameters which cause the entire subnet to be replicated upon its activation. Subnets can also contain special places which have parameters associated with them. When the subnet becomes active, these places, along with their connecting arcs, are expanded to produce simple places. In Figure 3a, P2(i,j) is a parameterized place with j ranging from 1 to i. When this net becomes active, all parameters are applied to the nodes of the net, resulting in the net shown in Figure 3b. Dynamic structuring of the graph can be achieved by using program variables as parameter ranges.

## 4 Model Behavior

The PCM simulator package has been implemented in C on a SUN workstation. A menu-driven graphics front-end is provided to create and modify net instances. The current implementation supports a wide range of net attributes. Transition delays can be specified either as instantaneous (zero delay), fixed delay, or random based on exponential or uniform distributions. In addition, the delays can be expressed as functions of the program variables. The transition procedures and predicates are specified as standard C functions, which are precompiled. Places have types associated with them to specify the kind of performance values expected from them. For example, a place modelling a queue would yield figures on throughput, mean wait and service times, etc. The length of the simulation run is specified by the user. The simulation proceeds by moving time-stamped tokens through the net and collecting timing



(a)

(b)

**Figure 3:** Subnets

information at the places. Alternately, a state history of the execution of the net is maintained, which can be used to determine the sequence of events in the computation. The simulator has been validated using analytical results obtained using the Stochastic Petrinet Analyzer [9].

## 5 An Example

In this section, the utility of the model is demonstrated by means of an example. The computation, suggested by J. Dongarra and D. Sorensen of Argonne National Laboratories, is the solution of a lower triangular matrix:

$$Tx = b$$

where T is an nxn lower triangular matrix, x and b are n-vectors. The algorithm used is to decompose the matrix into blocks as illustrated in Figure 4. There are three types of tasks in the computation: *init*, *solve* and *matvect*. The tasks are described next:

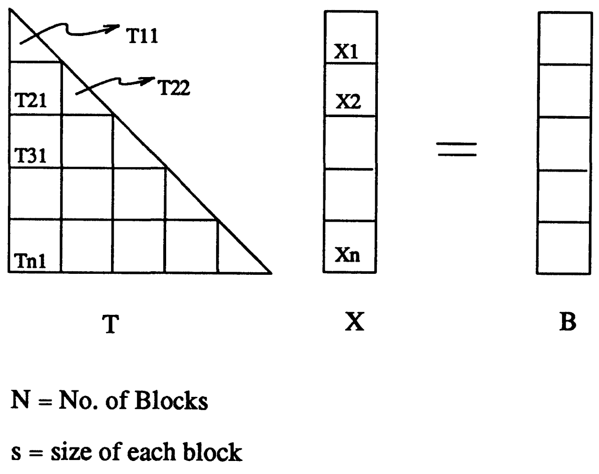


**Figure 4:** A Triangular Solver

init(i) initializes one block row of the system.

solve(i) triangular solver for the *i*th triangular diagonal block. It solves for $x_i$ in the system:

$$T_{ii}x_i = b_i$$

This can be done only after all matvect tasks for row i have completed. Observe that solve(1) (solve for block 1) can begin immediately after initialization, and does not depend on any matvect tasks.

matvect(j, i) executes the transformation

$$T_{ji}x_i - b_j \text{ --> } b_j$$

on the *j*th block in column i. This step can only be executed if solve(i) has been completed.

Figure 5 shows the CSL program which specifies the above computation. The CONSTRUCT statement contains declarations of the tasks, shared variables and communication channels involved in the computation. The declaration of Solve, for example, specifies N tasks Solve(1), Solve(2)...Solve(N). All Solve tasks are identical, and their compiled (object) code is found in file C2. A list of shared objects accessible to each task completes the specification of the Solve tasks. The Matvect tasks process the non-diagonal blocks of the matrix. Associated with each Matvect is a boolean task condition which is set after each execution of the task.

The program starts off N parallel streams, as denoted by the outer COBEGIN. Each parallel stream begins by executing a diagonal (solve) task first, and then all non-diagonal (matvect) tasks in that column in parallel (inner cobegin). The WAIT statement ensures that the execution of each solve task begins only after all matvect tasks in its row

```
JOB Triangle;

VAR N : integer;

BEGIN
  N := 4;
  CONSTRUCT
      TASKS
        Init(i)  : C1 [ T(i,j), X(i) ]
                  RANGE j = 1 to i,
                        i = 1 to N;
        Solve(i) : C2 [T(i,i), X(i) ]
                  RANGE i = 1 to N;
        Matvect(i,j) : C3[T(i,j), X(i), X(j)]
                       CONDITION C(i,j)
                  RANGE j = 1 to i-1,
                        i = 1 to N;
  END; { CONSTRUCT }


  WITH T(i,j), X(i) DO
    EXECUTE Init(i)  RANGE j = 1 to i,
                           i = 1 to N;

  COBEGIN
    ( // WAIT C(i,j) RANGE j = 1 to i-1;
         WITH X(i), T(i,i) DO
             EXECUTE Solve(i);
         COBEGIN
           (// WITH T(j,i), X(j) : X(i) DO
                  EXECUTE Matvect(j,i)
           ) RANGE j = i+1 to N;
         COEND
    ) RANGE i = 1 to N;
  COEND;
END.
```

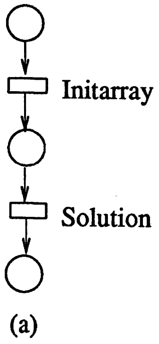
**Figure 5:** CSL Program for Triangular Solver

signal their completion by setting their task conditions. The statement:

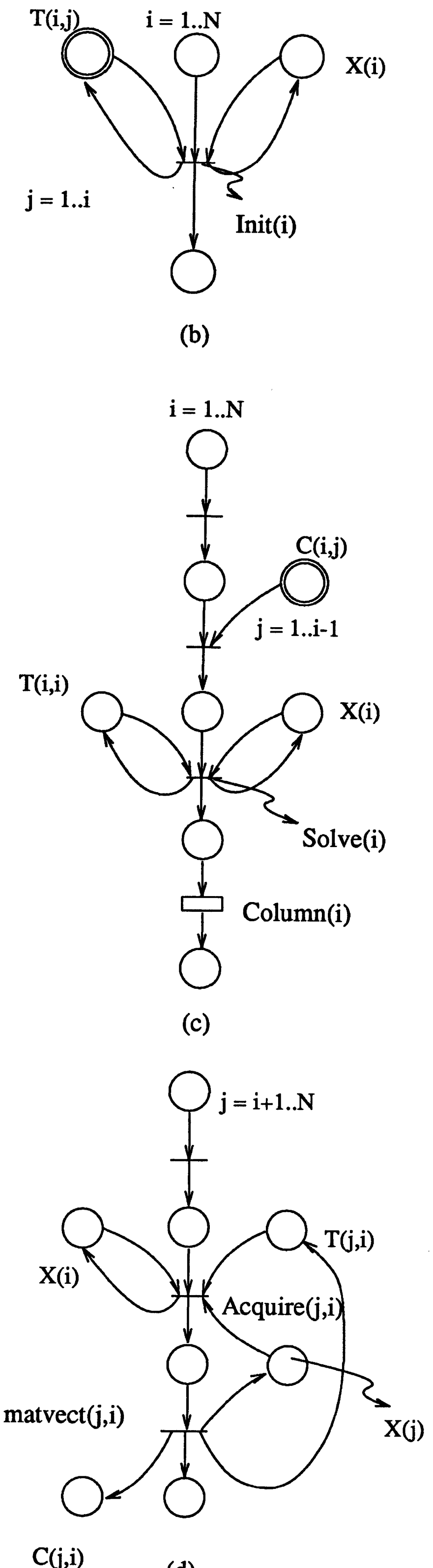
```
WITH T[j,i], X[j] : X[i] DO
   EXECUTE  Matvect (j,i);
```

specifies that the task matvect requires exclusive access to shared objects T[j,i] and X[j]; and wishes to use X[i] in read-only, or non-exclusive mode. Finally, the CSL program demonstrates the power of the RANGE statement as a construct for parameterizing the computation.

The equivalent representation of the computation in the PCM model, shown in Figure 6, consists of four subnets.



(a)



(d)

**Figure 6:** PCM solution

Figure 6(a) shows the top level decomposition of the computation into the initialization phase (Initarray), and the solution phase (Solution). *Initarray* (Figure 6(b)) models N parallel streams, with each stream containing one execution of Init. The place T(i,j) is parameterized, and is expanded when the subnet becomes active. For example, if i is 3, T(i,j) would be expanded to T(3,1), T(3,2) and T(3,3). The subnet *Solution*, shown in Figure 6(c) models the rest of the computation. Each solve task solve(i), requires exclusive access to T(i,i) and X(i), and can execute only after conditions C(i,1), C(i,2)..C(i,i-1) are set. After completion of solve(i), subnet *Column* is activated, which models the matvect task executions. The transition Acquire is used to gain exclusive access to shared objects T(j,i) and X(j), and non-exclusive access to X(i). The non-exclusive access is modelled here by merely copying the contents of X(i) into local memory. A more elaborate reader-writer scheme could be modelled by means of auxiliary program variables and appropriate transition predicates and procedures.

To calculate the delays in the model, the following architectural model is assumed. The system contains several independent processors, each with sufficient local memory. In addition, all processors can access a shared memory, and can set locks on parts of this memory. Access times are assumed to be equal for both local and remote accesses. The only overhead for remote accesses is for setting or releasing locks. Figure 7 shows the delays assumed for some operations. Also shown are delays associated with the transitions of the model, in terms of the size of each block.

The simulation results presented here are for a 48x48 system. The execution speed is measured for different values of N, the number of blocks. Figure 8 shows the various execution speeds as N is varied from 1 to 6. The structure of the computation is such that the parallelism is mostly contained in the concurrent executions of the matvects for a column. This explains the limited speedup when N is increased from 1 to 2. The small speedup that does appear seems to be due to the parallelism in the initialization tasks. As N is increased to 3 and 4, a substantial decrease in execution time is observed. Eventually, the bottleneck is due to conflicts for access to X(i) by all matvect tasks in column i.

| Operation | No. of Cycles |
|---|---|
| multiply + add (V) | 30 |
| acquire/release shared memory (c) | 10 |
| local store in memory (l) | 5/element |
| copy from shared memory (r) | 10/element |
| For block size s, | |
| matvect | $s^2V + 2c$ |
| Solve | $s^2V/2 + 4c$ |
| Acquire | rs + 4c |
| Init(i) for ith row | ls(s(i+1/2) + 1)<br>+ 2(i+1)c |

**Figure 7:** Delays associated with some operations

Using the powerful parameterization features provided in this model, the simulations can be repeated for various values of N with no changes to the net itself. The model is also capable of evaluating resource utilizations for communication channels and throughputs at various points in a computation, which can be used to determine bottlenecks in the computation.

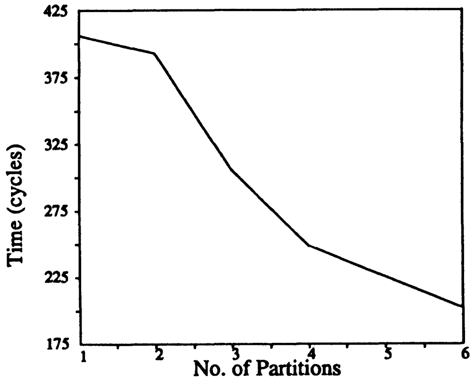


**Figure 8:** Execution Time vs No. of Partitions

## 6 Conclusions

In this paper, we have presented a model for representation and performance evaluation of parallel computations. In addition to modelling the algorithm behavior, the model incorporates features to study the execution overheads such as synchronization and communication delays. The model provides a vehicle for studying the configuration space of a given parallel computation, and deciding which configuration is most suitable for a given computation and architecture.

The parameterization mechanism described here leads to a powerful and compact representation which is extremely flexible. Dynamic graph structures can be represented using this parameterization scheme. Only small changes need to be made to a model instance to vary parameters like the granularity of a task or the degree of parallelism.

The model has been designed to provide a straight forward mapping from CSL programs to net instances. This mapping could be automated, providing a useful tool for analyzing and debugging CSL programs. A unique element of this modelling system is that it models the execution of programs, thus capturing the effects of different representations of algorithms. The Petri Net model is directly coupled to CSL constructs and vice versa.

## References

[1] Browne, J. C., "Formulation and Programming of Parallel Computations: A Unified Approach", *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp.624-631.

[2] Browne, J.C., Tripathi, A., Fedak, S., Adiga, A. and Kapur, R.,"A Language for Specification and Programming of Reconfigurable Parallel Computation Structures", *Proceedings of the 11th International Conference on Parallel Processing,* August 1982,

[3] Adiga, A., Fedak, S., Tripathi, A., and Browne, J. C., "A Computation Structures Language: Revised", Preliminary Technical Report TRAC-27, Department of Electrical and Computer Engineering, University of Texas at Austin, July 1981, pp.142-149.

[4] Peterson, J.L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Inc., Englewood Cliffs, N.J.07632, 1981, 290 pages.

[5] Nutt, G.J., "Evaluation Nets for Computer Systems Performance Analysis", *Proceedings of the 1972 Fall Joint Computer Conference,* Montvale, New Jersey: AFIPS Press, Dec. 1972, pp.279-286.

[6] Razouk, R., and Phelps, C., "Performance Analysis using Timed Petri Nets", *Proceedings of the 1984 International Conference on Parallel Processing,* August 1984, pp.126-128.

[7] Holliday, M. A. & Vernon, M. K., "A Generalized Timed Petri Net Model for Performance Analysis", *Computer Sciences Technical Report #593,* Computer Sciences Department, University of Wisconsin-Madison, May 1985.

[8] Keller, R., "Formal Verification of Parallel Programs", *Communications of the ACM,* Vol. 17, No. 7, July 1976, pp.371-384.

[9] Molloy, M.K., "Performance Analysis using Stochastic Petri Nets",*IEEE Trans. on Computers* Vol.C-31 No.9 pp. 913-917.

# Parallel Parsing on a One-Way Array of Finite-State Machines*

*Jik H. Chang and Oscar H. Ibarra*
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

*Michael A. Palis*
Department of Computer Science
University of Pennsylvania
Philadelphia, PA 19104-3897

*Abstract.* We show that a one-way two-dimensional iterative array of finite-state machines can recognize and parse strings of any context-free language in linear time. What makes this result interesting and rather surprising is the fact that each processor of the array holds only a fixed amount of information (independent of the size of the input) and communicates with its neighbors in only one direction. This makes for a simple VLSI implementation. Although it is known that recognition can be done on such an array, previous parsing algorithms require the processors to have unbounded memory, even when the communication is two-way.

## 1. Introduction

The parallel recognition of context-free languages by iterative arrays of finite-state machines was first considered in [11], where it was shown that context-free languages can be recognized by two-way two-dimensional iterative arrays in linear time. The construction in [11] is a parallel implementation of the well-known Cocke-Younger-Kasami (CYK) dynamic programming algorithm for recognizing the strings generated by a context-free grammar in Chomsky normal form [1]. Later in [10], the problem of computing the cost of an optimum binary search tree was shown to be solvable by a one-way two-dimensional array of unbounded-memory processors in linear time. It follows from the construction in [10] that context-free language recognition can also be done on a one-way two-dimensional iterative array (see, e.g., [3]). A recognition algorithm akin to the CYK algorithm is Earley's algorithm [1]. The advantage of the latter algorithm is that it does not require the context-free grammar to be in Chomsky normal form. A parallel implementation of Earley's algorithm was recently reported in [2]. Both recognition and parsing were considered, and detailed VLSI implementations were described. The structure of the iterative array implementation of the recognition algorithm is similar to that in [10] with each processor of the array being finite-state. However, the array implementation of the parsing algorithm required the processors to store and manipulate data which grow with the length of the input string being parsed. This is because of the way the parse is generated: the "indices" specifying the decompositions of the input string in the dynamic programming method are stored in the processors. Moreover, each processor must be capable of comparing index values, thus making the "control mechanism" (for transmitting and distributing data) of the array more complex. Thus, the parallel parser in [2] is not finite-state and its VLSI hardware implementation is not as simple as the one for recognition.

In this paper, we show that we can produce the parse without storing the "indices". In fact, we show that both recognition and parsing can be carried out on a one-way two-dimensional iterative array of finite-state machines (2DIA) in linear time. Because the array is finite-state, the processor's logic is simple and independent of the input size. This, together with the one-way communication, makes VLSI implementation simpler.

A 2DIA [4] is shown in Figure 1-(a). It consists of an n×n array of identical finite-state machines (nodes) that operate synchronously at discrete time steps by means of a common clock (not shown in the figure). The node at the upper right hand corner receives the serial input $a_1a_2 \cdots a_n\$$ beginning at time 0. The serial output $b_1b_2 \cdots b_k\$$ is observed from the node at the lower left hand corner, and is output beginning at time 2n-1. Thus, $b_i$ occurs at time 2n-2+i. (Note that all nodes generate outputs, but only the output of the node at the lower left hand corner is observed.) The $a_i$'s and $b_j$'s come from finite input and output alphabets $\Sigma$ and $\Delta$, respectively. Both input and output strings are terminated by a special endmarker \$, which is not in $\Sigma \cup \Delta$. We assume that, unlike the $a_i$'s, \$ is not "consumed" when read by the array, and is always available for rereading. The state and outputs of a node at time t are functions of its state and the states of its neighbors (or the external input in the case of the node at the upper right hand corner) at time t-1. At time 0, each node is in a distinguished quiescent state $q_0$, with its outputs set to the blank symbol $\lambda$. It remains in this state until one of its neighbors enters a non-quiescent state. The 2DIA has time complexity T(n) on input $a_1a_2 \cdots a_n\$$ if it outputs \$ after at most T(n) steps. Clearly, $T(n) \geq 3n-1$. We note that the 2DIA and the triangular array of Figure 1-(b) are equivalent. Clearly, the former can simulate the later. For the converse, the triangular array simulates in parallel the computations of the nodes that overlap when the 2DIA is folded along its diagonal. Our algorithms will be carried out on 2DIA's, since they are more convenient to use in our constructions. It is clear, however, that they can also be carried out on triangular arrays.

The 2DIA, being finite-state and one-way, is not so easy to program. Concurrency and synchronization of

* This research was supported in part by NSF Grant MCS 83-04756. O. H. Ibarra was also supported by a John Simon Guggenheim Memorial Foundation Fellowship.

processes are not easy to handle. Thus, for the proofs, we use an equivalent formulation of the 2DIA in terms of a restricted type of uni-processor sequential machine called 2DSM [7,8] (see Figure 2). The 2DSM consists of a finite-state control, with an input terminal from which it receives the serial input $a_1a_2 \cdots a_n\$$ and an output terminal from which the serial output $b_1b_2 \cdots b_k\$$ is observed. As in a 2DIA, we assume that \$ is not consumed when read by the machine and is always available for rereading. The 2DSM operates on a two-dimensional worktape of $n \times n$ cells. Initially, all cells of the worktape are set to $\lambda$. The 2DSM operates in sweeps as shown in Figure 3. A sweep begins with the machine in a distinguished start state $q_0$ and the read-write head (RWH) scanning the rightmost cell of the first row. The machine then reads an input symbol and scans the first row from right to left. For each cell scanned, the machine rewrites the cell, outputs a symbol, and changes state (except into $q_0$). When the machine has processed the leftmost cell, the RWH is reset to the rightmost cell of the next row in state $q_0$. The operation is repeated for all other rows. After scanning the bottom row, the RWH is reset to the rightmost cell of the first row to begin the next sweep. (Thus, a sweep begins at the rightmost cell of the first row and ends at the leftmost cell of the bottom row.) The RWH has the following capability: when scanning a cell in some row, it can also read (but not rewrite) the content of the cell directly above the cell currently scanned. The output symbols $b_1, \ldots, b_k, \$$ are the outputs produced by the leftmost cell of the bottom row. The 2DSM has sweep complexity S(n) on input $a_1a_2 \cdots a_n\$$ if it outputs \$ after at most S(n) sweeps. Clearly, $S(n) \geq n+1$.

In [7,8], the following result was shown:

**Theorem 1.** Let $T(n) \geq 3n-1$. If $M_1$ is a 2DIA with time complexity T(n), then we can effectively construct an equivalent 2DSM $M_2$ with sweep complexity $T(n)-2n+2$, and conversely.

Thus, we need only show the constructions on a 2DSM, since automatic conversion to the corresponding 2DIA is guaranteed by the constructive proof of this result.

## 2. Context-free Language Recognition and Parsing on a 2DIA

Our construction is based on the CYK algorithm [1]. The same techniques should apply to Earley's algorithm [1].

Let $G = (N, \Sigma, S, P)$ be a context-free grammar (CFG), where N and $\Sigma$ are finite nonterminal and terminal alphabets, respectively, $S \in N$ is the start symbol, and P is a finite set of productions (or rules) in Chomsky normal form. Thus, the productions in P are of the form $A \rightarrow BC$ or $A \rightarrow a$, $A,B,C \in N$ and $a \in \Sigma$. Let L(G) denote the language generated by G. For any string $x = a_1a_2 \cdots a_n$, $n \geq 1$ and $a_i \in \Sigma$, let $Q(i,j) = \{A \mid A \in N \text{ and } A \Rightarrow a_ia_{i+1} \cdots a_j\}$, $1 \leq i \leq j \leq n$. Then, $Q(i,i) = \{A \mid A \in N \text{ and } A \rightarrow a_i \in P\}$, $1 \leq i \leq n$, and $Q(i,j) = \bigcup_{i \leq k < j} Q(i,k) \cdot Q(k+1,j)$, $1 \leq i < j \leq n$, where $X \cdot Y = \{A \mid (\exists B,C)\ (B \in X, C \in Y, \text{and } A \rightarrow BC \in P)\}$. The CYK algorithm builds table Q by dynamic programming. Then, $x \in L(G)$ if and only if $S \in Q(1,n)$.

The above algorithm can be modified so as to also output a parse of x if $x \in L(G)$. This is accomplished by associating with each Q(i,j) another set R(i,j) containing "rules" of the form $[A \rightarrow BC]$ or $[A \rightarrow a]$, $A,B,C \in N$ and $a \in \Sigma$. The R(i,j)'s are computed as follows. Suppose that A becomes a member of Q(i,j), $i<j$, as a result of the rule $A \rightarrow BC$, $B \in Q(i,k)$ and $C \in Q(k+1,j)$, for some $i \leq k < j$. Then $[A \rightarrow BC]$ is in R(i,j). Similarly, if A becomes a member of Q(i,i) as a result of the rule $A \rightarrow a_i$, then $[A \rightarrow a_i]$ is in R(i,i). Note that, in fact, the Q(i,j)'s need not be computed, since they can be derived from R(i,j)'s. (Q(i,j) is simply the set of all nonterminal symbols appearing on the left-hand sides of rules in R(i,j).)

Figure 4-(a) gives an example of a CFG G. For input x = *baaba*, the table R of R(i,j)'s is shown in Figure 4-(b) (ignore the *'s for now). Observe that *baaba* is in L(G) since R(1,5) contains at least one rule whose left-hand side is S. The same table, with the entries relabeled, is shown in Figure 4-(c).

If $x \in L(G)$, a parse tree of x can be obtained by backtracking. That is, one starts by choosing from R(1,n) a rule of the form $[S \rightarrow AB]$. Then, for $k = 1, \ldots, n-1$, R(1,k) and R(k+1,n) are searched for rules of the form $[A \rightarrow CD]$ and $[B \rightarrow EF]$, respectively. (If the search is successful for more than one value of k, one is chosen arbitrarily.) The process is repeated for each newly found successor until the R(i,i)'s are reached. Figure 4-(d) gives a parse tree of the string x = *baaba*, obtained by backtracking on the table R of Figure 4-(b). In each node, the number below the rule represents the set from which the rule was chosen. (The chosen rules are also marked * in Figure 4-(b).) The parse itself can be specified in several ways. For our purposes, we shall output the *right parse* of the input string, which is obtained by traversing the parse tree in reverse preorder. For the given example, the right parse is $[S \rightarrow BC]$, $[C \rightarrow AB]$, $[B \rightarrow CC]$, $[C \rightarrow a]$, $[C \rightarrow AB]$, $[B \rightarrow b]$, $[A \rightarrow a]$, $[A \rightarrow a]$, $[B \rightarrow b]$. Thus, the right parse is obtained from the sequence of nodes 15-14-12-5-8-4-3-2-1.

We now show that the above parsing algorithm can be carried out on a 2DSM operating in O(n) sweeps. It then follows that the corresponding 2DIA also operates in linear time. The construction is complicated by the fact that the 2DSM is finite-state, since in this case it cannot store numbers whose values grow with n. As we shall see, this restriction makes the generation of the parse quite difficult. To simplify the presentation, we only illustrate the construction by means of an example, using table R of Figure 4-(b).

Let M be a 2DSM. Without loss of generality, assume that M has a worktape consisting of $3n \times 3n$ cells. (The worktape can always be reduced to $n \times n$ by simulating a $3 \times 3$ subarray in one cell.) Each row of the worktape is divided into three tracks. M operates in several phases: Table Construction, Tape Reversal, Regeneration, Parse-Tree Extraction, and Outputting of the Right Parse.

**Table Construction.** M starts by constructing table R on the first track of the worktape. The operation of M during this phase is a modification of that given in [8] (where only context-free language recognition was considered). The idea is to compute the entries on the ith diagonal of the table (Figure 4-(c)) during the ith sweep of M. Figure 5 shows the pattern we wish to achieve. The jth row of sweep i, $1 \leq j \leq i$, computes the jth entry of diagonal i of the table (starting with the topmost entry). For instance, during sweep 3, M computes *3*, *7* and *10* in rows 1, 2 and 3 of the worktape, respectively. Note that an entry is computed only after forming the "convolving" pairs on which it depends. For example, in the third sweep, *10* (represented by the pair *(10,10)*) is computed by first forming the pairs *(7,1)* and *(3,6)* on the first two cells of the row.

A careful study of the profile will reveal how M should generate the pairs. Except for the leftmost pair of each row, the left element of each pair is propagated diagonally downwards (i.e., one row down and one cell to the left). The right elements of the pairs are obtained by shifting them one row down during each subsequent sweep. The leftmost pair of each row is simply the value of the entry computed on this row. All of these actions can be performed by M. However, a problem arises when generating the left element of the first cell of each row. For instance, consider rows 3 and 4 of sweep 4 of the profile. The pair *(11,1)* in row 4 can only be formed by propagating *11* of the pair *(11,11)* in row 3 into this pair. However, M cannot pass information from left to right since it changes states only during a right-to-left row scan. This problem is overcome by using the following "folding technique". The idea is to fold the profile of each sweep along the dotted lines shown in Figure 5. The folded profile is shown in Figure 6. As a result of the folding, all the necessary propagation and shifting of elements can now be performed by M. For instance, in sweep 4 of the profile, pair *(11,1)* is now directly below pair *(11,11)*. Hence, *11* can be propagated from the latter pair to the former during a right-to-left row scan of M. Figure 7 gives the rewriting rules necessary to achieve the folding. In the figure, the boxed cell is the cell currently being rewritten; the solid (dotted) arrow means: copy the current (previous) contents. Thus, the right element of a pair is obtained from the *previous* right element of a pair in the row above it. The second track is used to hold these previous values.

While performing the above steps, M also creates, on the third track, a table T whose ith row, $1 \leq i \leq n$, is simply the bottom row of sweep i of M (see Figure 8). Thus, at the end of n sweeps M has the last sweep profile of Figure 6 on the first track and table T on the third track of the worktape. When $ is first read (on the (n+1)st sweep), M can then check whether *15* has a rule whose left-hand side is S (the start symbol). If no such rule exists, M outputs "error" and halts. Otherwise, it chooses one such rule, marks it by *, and proceeds to the next phase.

**Tape Reversal.** During this phase, M reverses the tape contents by first reversing the rows, then the columns of the worktape. To reverse the rows, M proceeds as follows. When $ is first read (on the (n+1)st sweep), M rewrites the rightmost cell of the first row by '-' and stores the previous contents of this cell on the left end of the row (i.e., after the last 'B'). (M in fact stores the previous contents in reverse. That is, if the previous contents were (a,b)(c,d), then M stores (d,c)(b,a)). M also marks the left end of the row by @. The same is done for all other rows. In succeeding sweeps and for each row, M moves past the '-'s and rewrites the first cell encountered by '-'. It then stores the previous contents of this cell in the cell marked @, while shifting the tape contents to the left. M repeats the process until the next cell to be rewritten is the one marked @. The tape profile of M after reversing the rows is shown in Figure 9-(b). Next, M reverses the columns. The steps carried out are the same except that they are done vertically (see Figure 9-(c)). M also marks the bottom row of the table by @. This phase takes 2n sweeps to complete. The configuration of the worktape after reversal is shown in Figure 10.

Once the tape contents have been reversed, M generates a parse tree of the input string by backtracking. It does this by performing the next two phases simultaneously.

**Regeneration.** During this phase, M regenerates the sweep profiles shown in Figure 6 in reverse order starting with sweep 5. In general, M can regenerate the profile of sweep i from the profile of sweep i+1 and from table T. For instance, to regenerate the profile of sweep 4, M starts by erasing the top row of the current profile and replacing the ***right*** elements of the pairs in the second row by the corresponding elements in the second row of table T. The left elements of the second row and the pairs of the remaining rows are obtained by shifting and propagating the entries according to the rules given in Figure 12, which are essentially the "inverses" of the rules given in Figure 7. Again, the boxed cell represents the cell currently being rewritten; the solid (dotted) arrow means: copy the current (previous) contents. The same steps are carried out to regenerate the remaining sweeps. In each case, the top row of the current profile is erased, and the right elements of the next row is replaced by the right elements in the corresponding row of table T. The left elements of the row and the pairs of the remaining rows are formed using the rules in Figure 12. The regeneration phase terminates after the bottom row of the table has been erased. To know when this happens, M does the following. At the start of the regeneration phase, M initializes a marker on the rightmost cell of the top row (*(15,15)(15,15)* in Figure 11). In each subsequent sweep, it shifts the marker one cell to the left. The regeneration phase terminates when the marker moves past the leftmost column of the table.

**Parse-Tree Extraction.** While executing the regeneration phase, M simultaneously performs a marking phase through which it extracts a parse tree of the input string. Recall that at the end of the table construction phase, M has marked by * a rule in *15* which has S (the start symbol) on its left-hand side. This rule is the root of the parse tree generated by M. The rest of the tree is generated as follows. When scanning a row, M checks if the rightmost (non-'B') entry has a rule marked *. If no rules are marked, M writes '-' at the left end of the row. Otherwise, the rule is either of the form [A→a] or [A→BC]. In either case, M remembers the rule in its state and writes the rule at the left end of the row. In addition, if the rule is of the form [A→BC], M does the following. While moving left on the worktape, M searches for the first pair of entries (X,Y) in which B appears on the left-hand side of a rule in X and C appears on the left-hand side of a rule in Y. Let the two rules be [B→DE] and [C→FG]. M then marks both [B→DE] and [C→FG] by *.

During the regeneration phase, the entries (along with any marked rules) are shifted and propagated according to the transition rules given in Figure 12. Thus, at some point an entry with a marked rule will reach the rightmost (non-'B') cell of some row.

The execution of the above steps results in the generation of a matrix of rules, as shown in the last sweep profile of Figure 11 (to the left of the dotted vertical line). This represents essentially a parse tree of the input string (see Figure 4-(d)). This phase terminates at the same time as the regeneration phase.

**Outputting the Right Parse.** What remains to be done is to output the right parse itself. Intuitively, this can be done by scanning the matrix of rules one column at a time, starting with the rightmost column. Within each column, the rules are output one at a time, starting with the topmost one. However, a problem arises because M cannot tell when it has exhausted one column and proceed to the next. To overcome this problem, M performs a "shift-and-accumulate" phase during which the matrix is shifted downwards and the rules are accumulated on the bottom row of the matrix. (The bottom row of the matrix is marked @ at the end of the column-reversal phase.) The details of this phase are shown in Figure 13. The bottom cells can hold at most two rules. The cell acts as a queue so that when a rule is shifted into a cell which is already full, the first rule shifted in is taken out and passed to the next cell. Note that there are exactly 2n-1 rules in the matrix, and these can all be accommodated in the bottom cells. This "shift-and-accumulate" phase can be performed by M in n sweeps. Once completed, M can then output the right parse of the string by outputting the rules from right to left. (Note that the output is observed at the bottom row of the worktape, which is different from the bottom row of the matrix. In this case, M simply outputs the same rule in each of the remaining rows of the tape.) We leave it to the reader to verify that outputting the rules in this order does indeed give the right parse of the input string.

M takes 7n sweeps to complete the computation. The corresponding 2DIA has time complexity 9n-2. Thus, we have

**Theorem 2.** Context-free language recognition and parsing can be carried out on a 2DIA in linear time.

**Remark 1.** The 2DSM described above can be made to operate in time (n+1)+$\epsilon$n for any real number $\epsilon$>0, by simulating k sweeps in one sweep and allowing the machine to output k symbols at a time. It follows from the characterization that the 2DIA can be made to operate in time (3n-1)+$\epsilon$n.

**Remark 2.** Strings of length less than n can also be recognized/parsed by the 2DSM. This follows from the fact that the length of the input determines the size of the table built by the 2DSM during the table construction phase. Moreover, during the outputting of the parse, the answer is propagated to all rows lying below the table.

## 3. Conclusion

We have shown that context-free language recognition/parsing can be carried out on a 2DIA in linear time. One can show that a (one-way) 2DIA operating in linear time can be simulated by an unbounded two-way 2DIA (Figure 14) also operating in linear time. Moreover, the computation of the two-way 2DIA can be sped up to operate in time (n+1)+$\epsilon$n, for any real number $\epsilon$>0 [7]. It follows that CFL recognition/parsing can be carried out on a two-way 2DIA in (n+1)+$\epsilon$n time.

Our techniques are applicable to other problems involving dynamic programming, e.g., to the problem of finding approximate patterns in strings [14,15], the string-to-string correction problem [12,16], the longest common subsequence (LCS) problem [5,6,12,16], dynamic time warping, optimum generalized alignment, error-correction, etc. [13]. For these problems, we are interested in the "parse", rather than the value, of an optimal solution. For example, for the LCS problem, what we require as output is an LCS of the two given strings, not its length. If only the value of an optimal solution is required, the problems can be carried out in linear time on a one-dimensional array of non-finite-state processors [9].

## References

[1] Aho, A. and J. Ullman, *The Theory of Parsing, Translation, and Compiling*, Volume 1: *Parsing*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] Chiang, Y. and K. Fu, Parallel parsing and VLSI implementations for syntactic pattern recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6, 3 (1984), pp. 302-313.

[3] Chu, K. and K. Fu, VLSI architectures for high speed recognition of context-free languages and finite-state languages, *The Ninth Annual International Symp. Computer Architecture*, April 1982.

[4] Cole, S., Real-time computation by n-dimensional iterative arrays of finite-state machines, *IEEE Transactions on Computers*, 18, 4 (1969), pp. 346-365.

[5] Hirschberg, D., A linear space algorithm for computing maximal common subsequences, *Communications of the Association for Computing Machinery*, 18, 6 (1975), pp. 341-343.

[6] Hirschberg, D., Algorithms for the longest common subsequence problem, *Journal of the Association for Computing Machinery*, 24, 4 (1977), pp. 664-675.

[7] Ibarra, O. and M. Palis, Two-dimensional systolic arrays: characterizations and applications, submitted to *Journal of Parallel and Distributed Computing.*

[8] Ibarra, O., S. Kim, and M. Palis, Designing systolic algorithms using sequential machines, to appear in *IEEE Transactions on Computers*; extended abstract in *Proceedings 25th IEEE Annual Symposium on Foundations of Computer Science*, 1984, pp. 46-55.

[9] Ibarra, O. and M. Palis, VLSI algorithms for solving recurrence equations and applications, submitted to *Journal of Parallel and Distributed Computing.*

[10] Guibas, L., H. Kung, and C. Thompson, Direct VLSI implementation of combinatorial algorithms, *Proceedings of Caltech Conference on VLSI,* 1979, pp. 509-525.

[11] Kosaraju, S., Speed of recognition of context-free languages by array automata, *SIAM Journal on Computing,* 4, 3 (1975), pp. 331-340.

[12] Masek, W. and M. Paterson, A faster algorithm for computing string-edit distances, *Journal of Computer and System Sciences,* 20 (1980), pp. 18-31.

[13] Sankoff, D. and J. Kruskal (eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison,* Addition-Wesley, Reading, Massachusetts,1983.

[14] Sellers, P., The theory and computation of evolutionary distances: pattern recognition, *Journal of Algorithms,* 1 (1980), pp. 359-373.

[15] Ukkonen, E., Finding approximate patterns in strings, *Journal of Algorithms,* 6 (1985), pp. 132-137.

[16] Wagner, R. and M. Fischer, The string-to-string correction problem, *Journal of the Association for Computing Machinery,* 21 (1974), pp. 168-173.

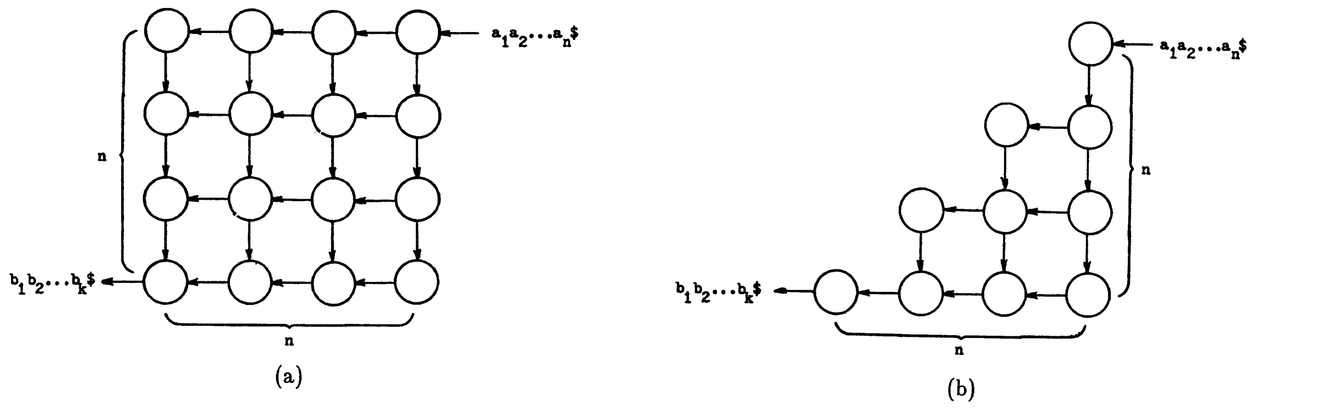


**Figure 1.** (a) A 2DIA; (b) A triangular 2DIA.

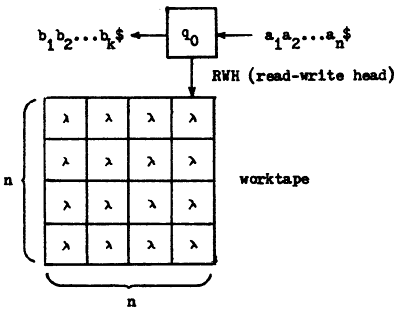
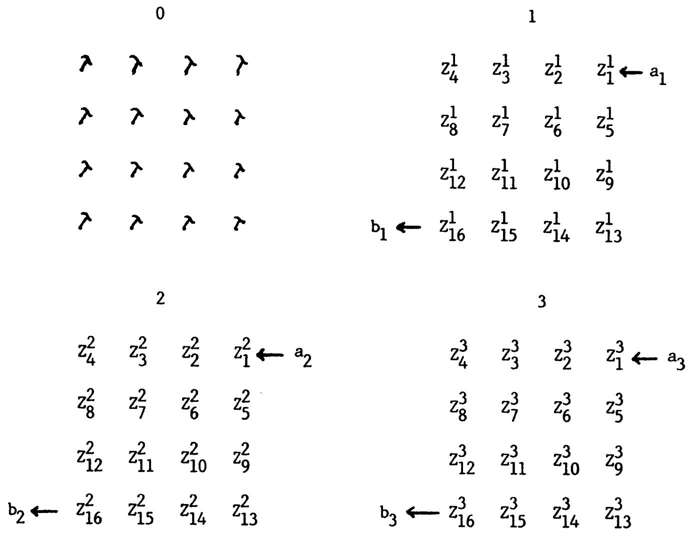


**Figure 2.** A 2DSM.



0
λ λ λ λ
λ λ λ λ
λ λ λ λ
λ λ λ λ
1
$z_4^1$ $z_3^1$ $z_2^1$ $z_1^1$ ← $a_1$
$z_8^1$ $z_7^1$ $z_6^1$ $z_5^1$
$z_{12}^1$ $z_{11}^1$ $z_{10}^1$ $z_9^1$
$b_1$ ← $z_{16}^1$ $z_{15}^1$ $z_{14}^1$ $z_{13}^1$
2
$z_4^2$ $z_3^2$ $z_2^2$ $z_1^2$ ← $a_2$
$z_8^2$ $z_7^2$ $z_6^2$ $z_5^2$
$z_{12}^2$ $z_{11}^2$ $z_{10}^2$ $z_9^2$
$b_2$ ← $z_{16}^2$ $z_{15}^2$ $z_{14}^2$ $z_{13}^2$
3
$z_4^3$ $z_3^3$ $z_2^3$ $z_1^3$ ← $a_3$
$z_8^3$ $z_7^3$ $z_6^3$ $z_5^3$
$z_{12}^3$ $z_{11}^3$ $z_{10}^3$ $z_9^3$
$b_3$ ← $z_{16}^3$ $z_{15}^3$ $z_{14}^3$ $z_{13}^3$

**Figure 3.** Computation profile of a 2DSM for 3 complete sweeps.

```
N = {S,A,B,C}
Σ = {a,b}
S = start symbol

P = {S → AB | BC,
     A → BA | a,
     B → CC | b,
     C → AB | a }
```

(a)

| b | a | a | b | a |
|---|---|---|---|---|
| R(1,1)<br>{B→b*} | R(2,2)<br>{A→a*<br>C→a} | R(3,3)<br>{A→a*<br>C→a} | R(4,4)<br>{B→b*} | R(5,5)<br>{A→a<br>C→a*} |
| R(1,2)<br>{A→BA<br>S→BC} | R(2,3)<br>{B→CC} | R(3,4)<br>{S→AB<br>C→AB*} | R(4,5)<br>{A→BA<br>S→BC} | |
| R(1,3)<br>∅ | R(2,4)<br>{B→CC} | R(3,5)<br>{B→CC*} | | |
| R(1,4)<br>∅ | R(2,5)<br>{S→AB<br>C→AB*<br>A→BA<br>S→BC} | | | |
| R(1,5)<br>{S→BC*<br>A→BA<br>S→AB<br>C→AB} | | | | |

(b)

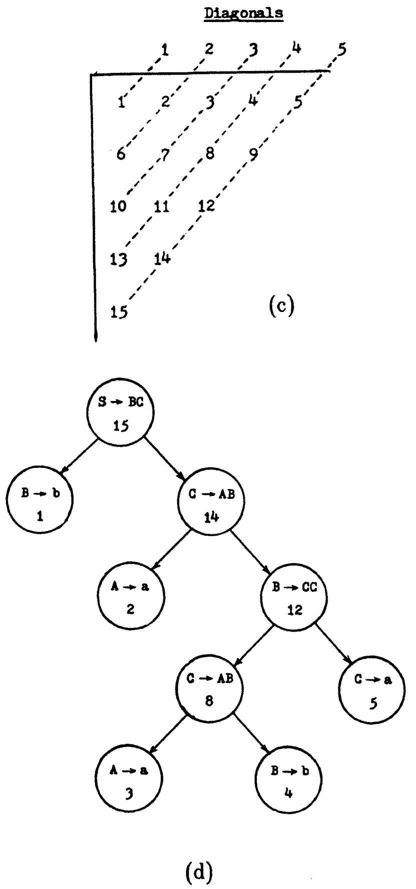


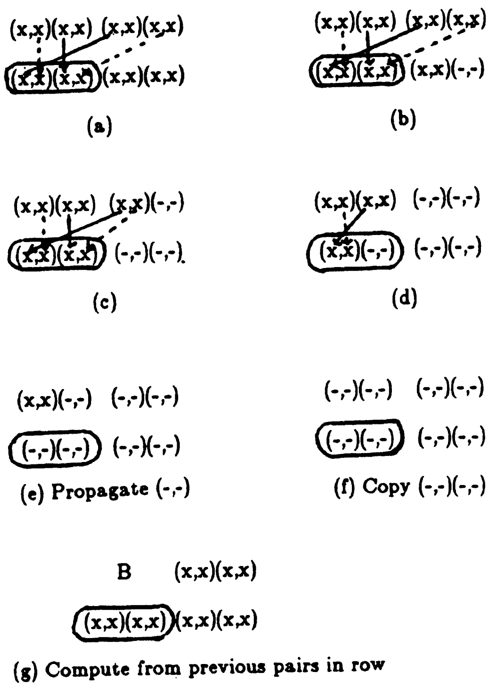
**Figure 4.** (a) A CFG G; (b) Table R for x = baaba;
(c) Table R, relabeled; (d) A parse tree for x = baaba.

```
                                          (1,1) ← a1

                                B         (2,2) ← a2
                              (6,6)       (2,1)

                    B           B         (3,3) ← a3
                    B         (7,7)       (3,2)
                 (10,10)      (3,6)       (7,1)

         B          B           B         (4,4) ← a4
         B          B         (8,8)       (4,3)
         B       (11,11)      (4,7)       (8,2)
     (13,13)      (4,10)      (8,6)      (11,1)

  B      B          B           B         (5,5) ← a5
  B      B          B         (9,9)       (5,4)
  B      B       (12,12)      (5,8)       (9,3)
  B   (14,14)     (5,11)      (9,7)      (12,2)
(15,15) (5,13)    (9,10)     (12,6)      (14,1)
```

**Figure 5.** Possible "layout" of the computation on the worktape of M.

```
                                                              (1,1)(1,1) ← a1

                                                    B         (2,2)(2,2) ← a2
                                               (6,6)(6,6)     (2,1)(-,-)

                                  B                 B         (3,3)(3,3) ← a3
                                  B            (7,7)(7,7)     (3,2)(-,-)
                          (10,10)(10,10)       (3,6)(7,1)     (-,-)(-,-)

                B                 B                 B         (4,4)(4,4) ← a4
                B                 B            (8,8)(8,8)     (4,3)(-,-)
                B         (11,11)(11,11)       (4,7)(8,2)     (-,-)(-,-)
        (13,13)(13,13)     (4,10)(11,1)        (8,6)(-,-)     (-,-)(-,-)

  B             B                 B                 B         (5,5)(5,5) ← a5
  B             B                 B            (9,9)(9,9)     (5,4)(-,-)
  B             B         (12,12)(12,12)       (5,8)(9,3)     (-,-)(-,-)
  B     (14,14)(14,14)     (5,11)(12,2)        (9,7)(-,-)     (-,-)(-,-)
(15,15)(15,15)  (5,13)(14,1)     (9,10)(12,6)  (-,-)(-,-)     (-,-)(-,-)
```

**Figure 6.** Computation profile of M obtained by folding the profile of Figure 5 along the dotted lines.



(e) Propagate (-,-)

(f) Copy (-,-)(-,-)

(g) Compute from previous pairs in row

**Figure 7.** Rewriting rules for M during the table construction phase.

| | | | | |
|---|---|---|---|---|
| (-,-)(-,-) | (-,-)(-,-) | (6,12)(10,9) | (1,14)(13,5) | (15,15)(15,15) |
| (-,-)(-,-) | (-,-)(7,9) | (2,12)(11,5) | (14,14)(14,14) | B |
| (-,-)(-,-) | (3,9)(8,5) | (12,12)(12,12) | B | B |
| (-,-)(4,5) | (9,9)(9,9) | B | B | B |
| (5,5)(5,5) | B | B | B | B |

(a)

| | | | | |
|---|---|---|---|---|
| (-,-)(-,-) | (-,-)(-,-) | (6,12)(10,9) | (1,14)(13,5) | (15,15)(15,15) |
| (-,-)(-,-) | (-,-)(6,8) | (1,11)(10,4) | (13,13)(13,13) | B |
| (-,-)(-,-) | (1,7)(6,3) | (10,10)(10,10) | B | B |
| (-,-)(1,2) | (6,6)(6,6) | B | B | B |
| (1,1)(1,1) | B | B | B | B |

(b)

**Figure 10.** Contents of (a) first track and (b) third track (i.e., Table T) after reversing the rows and columns.

| | | | | |
|---|---|---|---|---|
| B | B | B | B | (1,1)(1,1) |
| B | B | B | (6,6)(6,6) | (2,1)(-,-) |
| B | B | (10,10)(10,10) | (3,6)(7,1) | (-,-)(-,-) |
| B | (13,13)(13,13) | (4,10)(11,1) | (8,6)(-,-) | (-,-)(-,-) |
| (15,15)(15,15) | (5,13)(14,1) | (9,10)(12,6) | (-,-)(-,-) | (-,-)(-,-) |

**Figure 8.** Table T.

| | | | | |
|---|---|---|---|---|
| B | B | B | B | 1 |
| B | B | B | 3 | 2 |
| B | B | 6 | 5 | 4 |
| B | 10 | 9 | 8 | 7 |
| 15 | 14 | 13 | 12 | 11 |

(a)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1' | B | B | B | B@ | - | - | - | - | - |
| 2' | 3' | B | B | B@ | - | - | - | - | - |
| 4' | 5' | 6'' | B | B@ | - | - | - | - | - |
| 7' | 8' | 9' | 10' | B@ | - | - | - | - | - |
| 11' | 12' | 13' | 14' | 15'@ | - | - | - | - | - |

(b)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - | - | - |
| #<br>11' | #<br>12' | #<br>13' | #<br>14' | #<br>15' | - | - | - | - | - |
| 7' | 8' | 9' | 10' | B | - | - | - | - | - |
| 4' | 5' | 6' | B | B | - | - | - | - | - |
| 2' | 3' | B | B | B | - | - | - | - | - |
| 1'<br>@ | B<br>@ | B<br>@ | B<br>@ | B<br>@ | - | - | - | - | - |

(c)

**Figure 9.** (a) Profile of M after n sweeps;
(b) Profile after reversing the rows;
(c) Profile after reversing the columns.
(*if* i = (a,b)(c,d) *then* i' = (d,c)(b,a).)

```
             *                                             * *             *  *   *  *
             15 | (-,-)(-,-)  (-,-)(-,-)  (6,12)(10,9)    (1,14)(13,5)    (15,15)(15,15)
             *                             * *                *      *
             14 | (-,-)(-,-)  (-,-)(7,9)  (2,12)(11,5)    (14,14)(14,14)        B
             *                      * *       *      *
             12 | (-,-)(-,-)  (3,9)(8,5)  (12,12)(12,12)        B               B
                          *
              - | (-,-)(4,5)  (9,9)(9,9)        B               B               B
              *      *    *
              5 | (5,5)(5,5)    B           B               B               B

             *
           - 15 |   —           —           —               —               —
             *                             *
           - 14 | (-,-)(-,-)  (-,-)(6,8)  (1,11)(10,4)    (13,13)(13,13)        B
             *                 *
           - 12 | (-,-)(-,-)  (2,8)(7,4)  (11,11)(11,11)        B               B
           *            * *    *    *
           8  - | (-,-)(3,4)  (8,8)(8,8)        B               B               B
           *  *      *    *
           4  5 | (4,4)(4,4)    B           B               B               B

             *
        -  - 15 |   —           —           —               —               —
             *
        -  - 14 |   —           —           —               —               —
             *                 *
        -  - 12 | (-,-)(-,-)  (1,7)(6,3)  (10,10)(10,10)        B               B
           *            *
        -  8  - | (-,-)(2,3)  (7,7)(7,7)        B               B               B
        *  *  *    *    *
        3  4  5 | (3,3)(3,3)    B           B               B               B

             *
     -  -  - 15 |   —           —           —               —               —
             *
     -  -  - 14 |   —           —           —               —               —
             *
     -  -  - 12 |   —           —           —               —               —
           *            *
     -  -  8  - | (-,-)(1,2)  (6,6)(6,6)        B               B               B
     *  *  *  *    *    *
     2  3  4  5 | (2,2)(2,2)    B           B               B               B

             *
  -  -  -  - 15 |   —           —           —               —               —
             *
  -  -  -  - 14 |   —           —           —               —               —
             *
  -  -  -  - 12 |   —           —           —               —               —
           *
  -  -  -  8  - |   —           —           —               —               —
  *  *  *  *  *    *    *
  1  2  3  4  5 | (1,1)(1,1)    B           B               B               B
```

**Figure 11.** Computation profile of M during the regeneration and parse tree extraction phases.

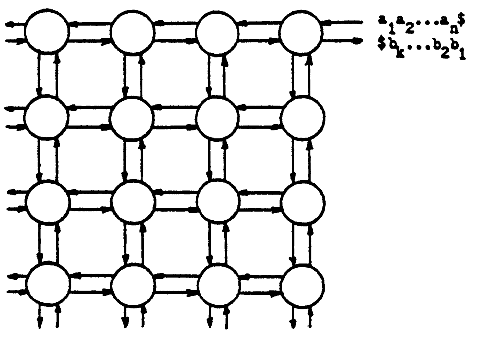
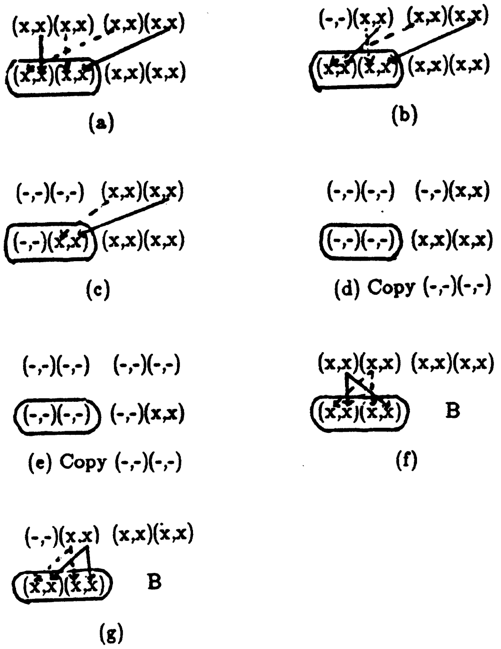


**Figure 14.** A two-way unbounded 2DIA.



(a)

(b)

(c)

(d) Copy (-,-)(-,-)

(e) Copy (-,-)(-,-)

(f)

(g)

**Figure 12.** Rewriting rules for M during the regeneration phase.

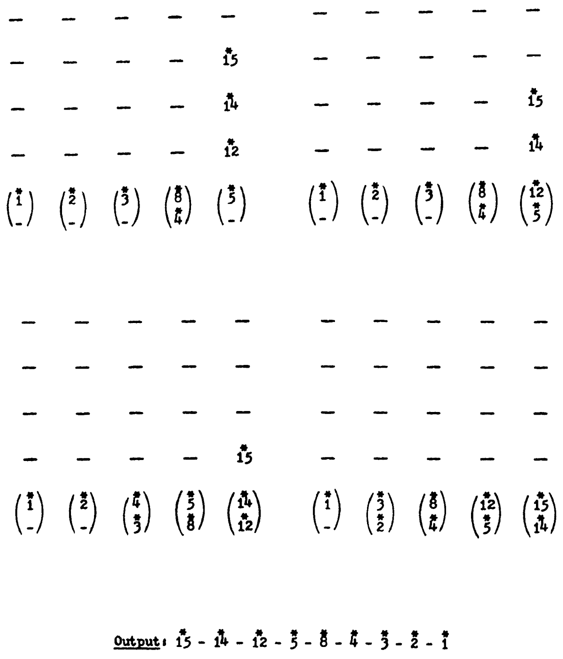


**Figure 13.** Computation profile of M during the shift-and-accumulate phase.

# Parallel Processing of Quadtrees on a Horizontally Reconfigurable Architecture Computing System

*Maurice Martin*

*Donald M. Chiarulli*

*S. Sitharama Iyengar*

Computer Science Department
Louisiana State University
Baton Rouge, Louisiana 70803

## ABSTRACT

The recent advances in computer architecture for image processing will provide unprecedented systems capabilities in the upcoming decade. Among the most significant aspects of the architectures will be their increasing ability to process image data in a parallel fashion. This paper presents a new computing architecture called "DRAFT" (Dynamically Reconfigurable Architecture for Factoring Things) used in parallel processing of image data structures. Furthermore, we show that quadtree data structures can be processed efficiently on this new parallel computing architecture. Algorithms are given for constructing and pruning the quadtree structure and for findings neighbors in a parallel fashion. The computational requirements of the "DRAFT" system in image processing are examined and their analysis is presented in detail.

## INTRODUCTION

The quadtree has received much attention in recent years as an efficient data structure for a variety of image processing applications. Recursively defined a quadtree may be empty or it may consist of a root with either none or exactly four sons consisting of quadtrees. For this paper a region will be the BLACK portion of a 2**n x 2**n array made up of a unit square pixels colored BLACK or WHITE. A sample region is presented in Fig. 1 and its quadtree is given in Fig. 2. We define a node in a quadtree to be a record containing the following fields. If P is a (pointer to a) node and D is in the set of directions {NW,NE,SW,SE}, then we may define the fields as follows. **COLOR(P)** has value WHITE or BLACK for a leaf, GRAY for an interior node. **SON(P)** (pointer to) a collection of four nodes which are the sons of P in each direction; NIL if no such node exists. **FATHER(P)** (pointer to) the father of P; NIL if P is the root. **PATH(P)** a path code recording the position of the node as an encoded series of directions from the root to the node as proposed by Jones, Raman and Iyengar [1, 2]. The (pointer to the) root of the quadtree will be denoted by ROOT. In our figures, the offspring of a node are drawn in the canonical order NW-NE-SW-SE.

Many algorithms have been written to process patterns stored as quadtrees. Important to these algorithms are procedures which allow determination of the neighbors of any given nongray node. These "neighbors" are the closest nodes in any of four directions, referred to as N,E,S, and W. An algorithm for traversing a quadtree to find these neighbors is given in Samet[3]. For a broader treatment of quadtree related research see [4, 5, 6, 7, 8].

Research supported in part by the National Science Foundation and the National Security Agency under grants NSF DCR 83-115-80 and NSA MDA904-85-H-0006.

This paper presents a scheme for performing quadtree functions in parallel on a new type of architecture called the DRAFT architecture. We begin with a brief overview of the architecture in section two. Section three outlines several programming concerns designed to maximize parallelism. Next a parallel data structure for representing quadtrees in a DRAFT machine is presented in section four, followed by an algorithm for building these structures from raster input in section five. Section six demonstrates parallel operation in the DRAFT environment with a sample neighbor find algorithm. An analysis of the expected performance improvement for common quadtree operations is provided in section seven.

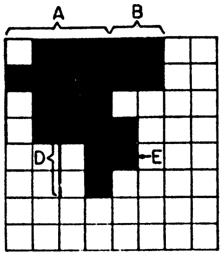


Figure 1

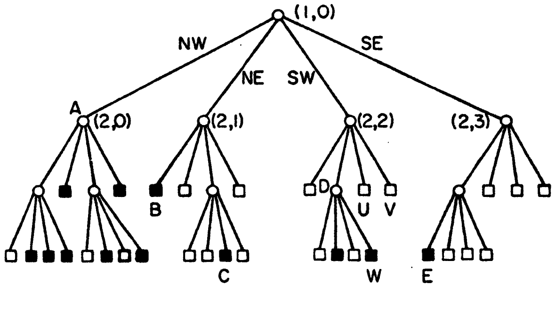


Figure 2

## A HORIZONTALLY RECONFIGURABLE ARCHITECTURE (DRAFT)

The DRAFT is an extended word length machine, 256 bits in the current implementation. This 256-bit word is constructed of 32-bit slices. A switching network is placed between each of the slices and is controlled by an 8-bit field in the microinstruction word. By setting or resetting the bits in this field the microprogrammer can at the micro-instruction level join one or more adjacent slices into a single processing unit or separate them into independently operating parallel processors. Thus the machine is horizontally reconfigurable along its word length into any combination of processing elements which can be constructed by joining adjacent slices. Possibilities range from a single 256-bit processor to eight 32-bit processors. For clarity we will refer to processing elements constructed in this manner as segments and the underlying 32-bit building blocks as slices. Figures 3 - 5 illustrate the details of the DRAFT architecture machine.

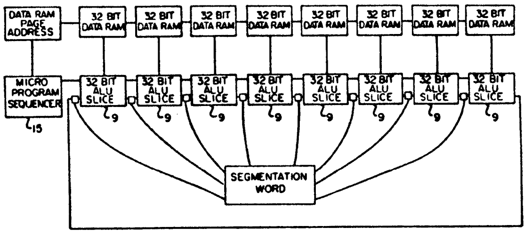


Figure 3

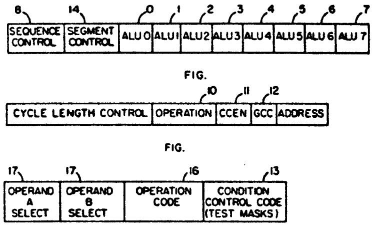


Figure 4

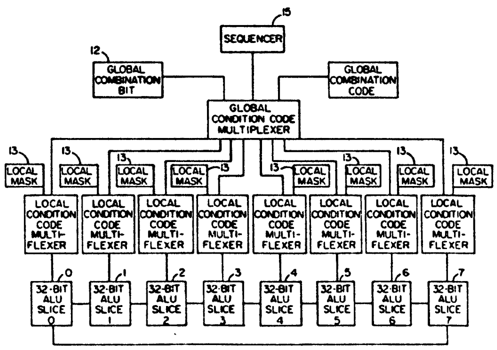


Figure 5

As stated, the slices and hence the segments configured from the slices are independently programmable. The major components of each slice include a control RAM and instruction pipeline, a 32-bit ALU built from VLSI bit-slice devices, a 64k-by-32-bit data RAM, and the switching circuitry necessary for constructing segments. The local control and data RAMs provide each segment with an instruction and data stream separate from the other segments in a given configuration. When combined into a segment, the control RAMs of each slice are simply programmed with the same instruction. The data RAMs of adjacent slices combine into higher- and lower-order 32-bit pieces of the segment data word.

Two components of the DRAFT machine which are global to the slices are the microinstruction sequencer and the global condition multiplexer. The microinstruction sequencer broadcasts a common next instruction address to all segments and is itself programmed with a separate sequencer instruction. The resulting program structure is unique in that a single control structure encloses parallel operations taking place within different processors of the same machine.

To provide for conditional execution of the segments, a slice level control mechanism is added by the global condition multiplexor. Between the slices and the global sequencer is a hierarchical arrangement of condition codes. At the bottom of this hierarchy is the slice level condition code bit. This bit is set by a combination of inputs, including the status output from the ALU, a slice level condition mask which selects the appropriate status output, and the segmentation word which propagates condition codes from the high-order to low-order slices within a segment.

The sequencer requires a single binary condition code to control branching within the global control structure. The function of the global condition mux is to generate the logical AND and logical OR combination of the local condition codes and to select under program control the appropriate combination for use by the sequencer. Thus, transfer of control instructions for the sequencer have, instead of the conventional conditional and unconditional analogs, unconditional, conditional all slices, and conditional any slice versions. The slice level control structure is implemented as conditional and unconditional analogs of all slice operations. Unconditional versions operate as normal instructions, while conditional analogs execute only so long as the local condition code is false. Once the local condition becomes true, these instructions become NOPs, and the slice essentially drops out of any parallel computations occurring within the global iterative or conditional structure. It is also notable that each segment has the ability to directly set or reset the local condition code independent of the actual status. This mechanism completes the condition code structure by adding the capability to mask out a given slice from consideration in a conditional all or conditional any decision for the global control structure. For a broader treatment of the DRAFT machine see[9].

## DRAFT PROGRAMMING PARADIGMS

What most distinguishes DRAFT based parallel algorithms from other types of parallel algorithms is the unique mix of global and local control structures. Because of the common instruction word address, each segment must wait until all segments have finished with a block of code before continuing. The conditional execute option allows a slice to shut itself off while waiting. At the programming level, this provides an excellent environment for converting sequential algorithms into parallel algorithms. The parallel algorithm is written as though it were sequential, with the added specification on conditional branching instructions indicating whether the branch is dependent on the condition testing true for all of the segments or just one of the segments.

Obviously it is possible to lose parallelism if some of the segments are shut off. This is the cost one must pay for the simplicity of the DRAFT programming environment. However, there are several ways in which can parallelism can be maximized by the programmer.

**Paradigm 1:** Since each slice processor includes its own local slice of the DRAFT memory word, the arrangement of data structures in memory is key to parallel operation. As the slices combine into segments of longer word length so do the memories. If a data item is to be considered as a

256 bit item, it should be right justified in the memory of the 256 bit segment to be used. Conversely if eight items are to be used in parallel they should be placed adjacent to one another in the slice level store of each segment.

**Paradigm 2:** Whenever possible, replace a block of code involving a conditional branching operation with a functionally equivalent block without conditional branching. Conditional branches depend on the status of all segments either ANDed or ORed together to generate global transfers of control. It is possible that when a conditional structure is executed, some of the segments will invoke conditional execution and shut themselves off. Since the objective is to maximize parallelism and minimize non-executing conditional code, the substitution of logical operations for conditional control structures such as if-then and case is advised wherever possible. If the function can be implemented without any type of conditional branching, this insures that all segments will remain active throughout. Although this seems to be a restriction on the programmer, examination of code and judicious selection of internal representations for data will reveal some branches that are the result of programming style and which can be eliminated without undue cost.

**Paradigm 3:** DRAFT micro-code allows parallel programming of the global micro-sequencer concurrently with the operation of the slices. The programmer has the opportunity to test condition bits at each instruction and branch out of redundant or useless operations. Although this branching is usually done at the end of a block of code on a sequential machine, it is possible to speed up an algorithm on the DRAFT by redundant testing in the microprogram sequencer.

## A PARALLEL DATA STRUCTURE FOR QUADTREES

The data structure used by Samet and others [4, 1, 2, 7, 8] for image quadtrees contains a color value and pointers to each of the four sons representing the NW,NE,SW, and SE quadrants of the image section. The structure proposed here is similar but includes additional fields for a parent pointer and a path code to be used in neighbor finding. The major difference in the DRAFT implementation is that the son quadtree nodes will be processed in parallel and reside adjacent to one another in a single segmented location of DRAFT memory. Thus the parent requires only a single pointer to this location. Figures 6 and 7 show a 4 x 4 raster scan and its corresponding full (worst case) quadtree representation. Figure 8 illustrates how the quadtree representation can be partitioned in a the prototype DRAFT machine.

<table>
<tr><td colspan="8">1</td></tr>
<tr><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td></tr>
<tr><td>6</td><td>7</td><td>10</td><td>11</td><td>14</td><td>15</td><td>18</td><td>19</td></tr>
<tr><td>8</td><td>9</td><td>12</td><td>13</td><td>16</td><td>17</td><td>20</td><td>21</td></tr>
</table>

**Figure 8**

The layout parallels the segmentation to be used in processing at each level: 256 bit uni-processing at the root, four segments of 64 bits each at the next lowest level, etc. Since the current DRAFT prototype is an eight-slice machine, nodes at levels three and below must be arranged as might be expected in a single processor environment. This amounts to a data partitioning between the slices, so that one eighth of the total area represented is handled by each processor and each of eight image octants will be processed in parallel. However, the DRAFT architecture imposes no limit on the number of slices. The current prototype has eight, but since only neighboring interconnections are required, it is feasible to string together a large number of slices to meet the requirements of some particular imaging task.

Note how this arrangement of the quadtree into DRAFT memory greatly increases the available parallelism in quadtree algorithms. Assuming that the number of slices is sufficient to hold all leaf nodes in a quadtree of depth N. It is possible by successive reconfigurations to access each node in a full quadtree in N steps, compared with the 4**N-1 steps required for traversals in a sequential environment. If the number of slices is not sufficient to hold every leaf node, the access time degenerates to that of a sequential access in all levels below log4(P) where P is the number of slices. In such a case, each slice is executing a sequential traversal with all slices running in parallel.

Other quadtree operations work well in this data partitioned format. Rotation of the image in 90 degree increments can be achieved by changing the order in which the child nodes are accessed; the pointer to the NW quadrant becomes the pointer to the SW quadrant, etc. No actual movement of data is required and pointer updates proceed in parallel. The same is true for a transposition of the image across some axis inside the plane of the image. Since superimposing one quadtree image upon another is a task which requires traversing two quadtrees simultaneously while constructing a third, one can expect timing improvements for this operation to be proportional to the improvements in traversal time. The time complexity for finding the intersection of two images should also be improved from 4**N to N, assuming a processor slice is available for every pixel.

## QUADTREE CONSTRUCTION FROM RASTER INPUT

For testing on the DRAFT prototype, two implementations of the quadtree structure were proposed: a vertical storage, where each field is placed in a different data location with the color at the base of the data page, and a horizontal packing where all fields are kept in a single data location. Vertical storage assigns one node to every data page; therefore horizontal packing is 16 times more memory efficient. However, horizontal packing has a disadvantage that extra code is required to extracted each field from the data word before it can be used. The procedures below were coded assuming horizontal packing.

Having chosen a representation scheme for the quadtree nodes the next problem is loading image data for an input device, such as a raster scanner into the DRAFT memory. This operation is important since the arrangement of pixels in the proper slice order is critical in constructing the parallel quadtrees as shown in Figure 8. For a DRAFT machine implementation sufficiently wide to provide a slice for each pixel this problem is relatively trivial. More difficult is the arrangement of nodes for the case of a limited number of slices, each processing a sub-quadrant of the image. Such was the case with the following test algorithms designed for the eight slice prototype. For loading an N x M pixel raster the data must be divided into eight octants of size N/2 x M/4. As the raster is scanned pixels from lines 1 to M/4 are placed in slice one for n < N/2 and slice three for n >=N/2. The next quarter for m are placed in slices two and four, then five and seven, and the remaining quarter loads into slices six and eight. Shown below is a sample implementation of this algorithm for the DRAFT machine host processor. The routine *get_pixel* reads the next pixel from the scanner and *store_pixel* places the pixel into the memory of the second parameter "slice".

```
/* sample algorithmfor loading an N x M
  raster into an 8 slice DRAFT machine */
int slice[4] = 0,1,4,5;

for m = 1 to max_lines
{/* lower half of line */
 for n = 1 to pixels_per_line/2;
 {
 pixel = get_pixel;
 store_pixel(pixel,slice[int(m/4)]);
 }
 for n = 1 to pixels_per_line/2;
 { /* upper half of line */
 pixel = get_pixel;
 store_pixel(pixel,slice[int(m/4)]+2);
 }
}
```

The store_pixel routine also performs an additional function to ease the construction of quadtrees for each of the octants in the next step. For this construction step the sequence of pixels in sequential memory locations will need to be the ordering from left to right of the leaves in the full quadtree to be constructed. This is not the order in which the pixels are received from the scanner. To achieve the required shuffling of the pixel locations the loading program uses an M/4 by N/2 translation raster. Each location in the translation raster holds the offset from raster base into which the pixel is to be stored. Figure 9 illustrates a translation raster for an eight-by-four octant processed at the slice level in a 16-by-16 raster.

**Input Raster**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |

**Translation Raster**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 9 | 10 | 3 | 4 | 11 | 12 |
| 17 | 18 | 25 | 26 | 19 | 20 | 27 | 28 |
| 5 | 6 | 13 | 14 | 7 | 8 | 15 | 16 |
| 21 | 22 | 29 | 30 | 23 | 24 | 31 | 32 |

**Figure 9**

The final step in preparing a raster for DRAFT quadtree operations is to construct and color the parent nodes ascending upwards to the root. Phase one scans in parallel the pixels of each of the eight octants. Every fourth pixel generates a new node colored black if all four pixels are black, white if all four are white, and grey if a mixture of black and white exists. Also during this scan the parent field of each pixel is filled with the address of the new node and the son field of the new node is set to the base address of the group of four. After scanning the pixel level nodes the process is repeated for each of the new parent parent nodes created until in the final iteration a single root node is generated for the octant quadtree processed by each slice. The procedure below shows the code for each slice. The actual code is implemented in DRAFT microcode.

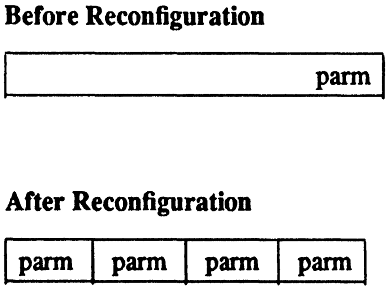
```
/* sample slice level algorithmfor generating
  quadtreesfor raster octants */
nodecount = n*m/8;
while(nodecount > 0)
 for i = 1 to nodecount by 4
 { new_parent = make_parent_node;
   son(new_parent) = i;
   color(parent) = color(i)
   for j = i to i+3
   { parent(j)=new_parent;
     color(parent)==color(j) : color(parent)=GREY;
   }
 }
```

Note that the conditional testing of child colors appears to be backwards in this algorithm. In fact the statement is an example of conditional execution. The program sequencer will cause the assignment statment to execute in all slices, however in those slices for which the local condition code is true, the assignment statement is converted to a nop and the parent color is unchanged.

Phase two of quadtree construction completes the upper levels of the image quadtree as shown in Figure 7. In the eight-slice prototype this will be two additional levels. Since all of the octant quadtrees roots lie in adjacent slices of the same location, the completion of the address fields is trivial. Each level is executed using a successively larger segmentation until the root executes in single segment mode. Having constructed the quadtree, the next step is the implementation of common quadtree operations. A complexity of analysis of most such operations will inevitably be dependent on the complexity of quadtree traversal. As stated, a sufficiently wide DRAFT implementation reduces such traversals to complexity proportional to the depth. When DRAFT segments must traverse entire subtrees in parallel the single instruction address forces the complexity to the worst case of the eight (for the prototype) subtrees. This is because all other segments must wait for the "slowest" segment to complete its traversal before continuing.

## FINDING NEIGHBORS

One essential quadtree operation that has received a good deal of attention [2, 10] is the neighbor-find operation. The DRAFT implementation of this algorithm is a good example of how varying segmentation can not only assist in quadtree traversal but can also return information such as the size of a neighboring block. The algorithm proceeds recursively with an N bit parameter holding the path code of the node for which the neighbors are to be located. N in this case is the number of bits in the current segment. At each level of the recursion the machine reconfigures to divide the current segment into four new segments which will simultaneously search the four children. Each of these segments is given an N/4 bit copy of the original parameter arranged across the parameter word as shown in Figure 10.



Before Reconfiguration

After Reconfiguration

Figure 10

A identical mechanism is used in reverse for the return parameter. In this case the segment locating a neighbor places an N bit copy of the path code for that neighbor into the corresponding segment of the return parameter word. When the recursion returns to the root the resulting word contains a string of four values representing the four neighbors of the node in question. The segment size of each of this values determine the size of the neighbor block returned. A recursive slice level algorithm for nearest neighbor is shown below. The actual micro-code for nearest neighbor on the DRAFT prototype is a non-recursive implementation using the parent pointers. This is due to the limited size of the stack memory for the sequencer chip used.

```
find_neighbor(node,quadtree)
{
/* locates the neighbors of "node"
   in "quadtree" */

if(color(quadtree) <> GREY)
   then if(is_neighbor(node,quadtree))
           then return(0);
           else return(quadtree);
   else { re_seg(4);
          find_neighbor(node,son(quadtree));
        }
}
```

The *re_seg* procedure above resets the segmentation and regenerates the parameter as above. *Is_neighbor* is a boolean function which compares the path codes of "node" and "quadtree" to determine if they are neighbors.

Algorithms have also been devised for *union* and *intersection* functions of two image quadtrees. For these functions the result quadtree is the union or intersection of the blacks regions of the two inputs Both algorithms are based on a top down traversal like the one discussed in section three and thus have complexity proportional the depth of the inputs.

## ANALYSIS

Analysis of DRAFT algorithms must be made based on two cases. In case one the slice width of the DRAFT implementation is equal the number of nodes in the worst case quadtree to be constructed. For this case quadtree construction, traversal, and neighbor finding reduce to O(N) since each step processes an entire level and N is equal to the depth of the quadtree. While this is not an unrealistic assumption based on current VLSI capabilities, no such machine for non-trivial rasters has been built. For operations on the existing eight slice DRAFT prototype the analysis divides into two phases. Phase one is the analysis above for the upper levels of the quadtree. Phase two, is an eight processor parallel version of essentially sequential operations on the eight octant subtrees. The proposed data structure divides all work evenly so that the time required to execute in parallel is the same as the equivalent sequential algorithm divided by p, where p is some even number of slices. The cost paid for this improvement is log4(p) steps representing those levels which can be fit into adjacent slices. For quadtree construction each quadtree is constructed in with O[(4(N/p)-1)/3] where N is the number of pixels in the raster. The entire quadtree for rasters of N pixels is thus constructed in O[log4(p) + (4(N/p)-1)/3] Assuming an N log4(N) complexity for sequential neighbor finding yields an identical analysis for the DRAFT neighbor find algorithm of complexity O[log4(p) + N/p log4(N/p)]. Traversals and traversal oriented operations improve from O(N) for the sequential case to O[log4(p)+ N/p] in the DRAFT algorithm.

In Iyengar and Moitra [5], a paradigm is given for measuring EPU, the Effective Processor Utilization, is defined as (complexity of the fastest sequential algorithm for a problem)/( the number of processors used * time complexity of the parallel algorithm). Using this measurement, DRAFT algorithms have an EPU of 1.

## FUTURE RESEARCH

The DRAFT machine has great potential for use in an image processing environment. Its quasi-SIMD structure makes it possible to create algorithms which make good use of parallel processing while retaining the familiar programming structures of a sequential environment. In addition to the work in new algorithms for this machine, two other areas are currently being given attention. The first is an attempt to reduce the essential switching functions to custom VLSI. The existence of such a chip set would greatly enhance the ability to apply this new processor design to a wide range of special purpose machines. Second is the development of a high-level programming environment. Whereas the current micro-code environment is well suited to optimizing performance, high-level language to support such operations as the segmented recursion discussed in this paper would be of great advantage.

## References

[1] V. Raman and S. S. Iyengar , "Properties and Applications of Forests of Quadtrees for Pictorial Data Representation," *BIT*, vol. 23, pp. 472-486, 1983.

[2] L. P. Jones and S. S. Iyengar, "Space and Time Efficient Virtual Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 2, pp. 244-248 , March 1984.

[3] H. Samet, "A Top-Down Quadtree Traversal Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, pp. 94-97, January 1985.

[4] G. M. Hunter and K. Steiglitz, "Operations on Images using Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, pp. 145-153, April 1979.

[5] A. Moitra and S. S. Iyengar, "Parallelism from Recursive Programs," *Advances in Computers*, June 1986 .

[6] H. Samet and M. Tamminen , "Computing Geometric Properties of Images Represented by Linear Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, No. 2, pp. 229-239, March 1985.

[7] D. Mark and D. Abel, "Linear Quadtrees from Vector Representation of Polygons," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 3, May 1985.

[8] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, No. 2, pp. 187-260, June 1984.

[9] D. M. Chiarulli, W. G. Rudd and D. A. Buell , "DRAFT: A Dynamically Reconfigurable Processor for Integer Arithmetic," *Proceedings, 7th International Symposium on Computer Arithmetic*, pp. 309-317, 1985.

[10] H. Samet and C. A. Shaffer, "A Model for the Analysis of Neighbor Finding in Pointer-Based Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 6, November 1985.

# SHEAR SORT: A TRUE TWO-DIMENSIONAL SORTING TECHNIQUE FOR VLSI NETWORKS

Isaac D. Scherson†, Sandeep Sen† and Adi Shamir‡

† Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106

‡ Department of Applied Mathematics
Weizman Institute of Science
Rehovot, Israel 76100

## Abstract

Given a sequence of numbers which can be mapped into an $m \times n$ array, sorting rows and columns is shown to yield an overall sorted sequence. This unusually simple procedure is proven to require $O(\log_2 m)$ iterations by analysing the data movement in the array under successive row and column sorts. An efficient bubble sort network suitable for VLSI implementation, with near-optimal $area-time^2$ performance is a direct application of the row-column sorting technique. The ease in implementation for practical VLSI chips is also demonstrated.

## I. Introduction

The problem of sorting numbers on a two-dimensional array has been studied by various researchers ([1], [2], [10]) and more recently by Leighton[3], Lang et al.[8] and Tseng et al.[12]. This problem involves routing of each data item to a distinct position of the array predetermined by some indexing scheme. Three different schemes have been considered by Thompson & Kung[1] : row major, shuffled row major, and snake-like row major (see Figure 1). The corresponding column-major forms can be considered equivalent to the schemes above. Most of the above referenced algorithms are based on two dimensional adaptations of very efficient sorting algorithms for linear sequences like bitonic sort[9] in [1] and [2], and odd-even merge sort[9] in [10] and these are essentially recursive in nature. Even though they perform optimally within a constant factor of the lower bound of O(n) for a $n \times n$ array, these algorithms spend most of the time in routing data to appropriate processors. Consequently, the complexity of the execution time is dominated by the number of unit routing steps in a SIMD model. Thompson and Kung[1] derived a 4(n-1) lower bound from an initial configuration where elements on the opposite corners of an $n \times n$ array have to be exchanged.

However, mere figures of time complexity appear to be superficial, when we consider the complexity of the control structure for the complicated data routing during the successive stages of recursion. One hardly needs to overemphasize the cost of the communication overheads in a VLSI implementation. In this context one can be more enthusiastic about the algorithm in [12], which in spite of being recursive in nature gets away with minimal control and achieves a bound within O(log n)[a] of the optimal. In the realm of sorting a two-dimensional array of numbers, a seemingly "nice" way would be to sort rows and columns (since it involves sorting on smaller problems of approximately $\sqrt{n}$ size) and "hope" that somehow a combination of these two operations will terminate in a sorted sequence. Unfortunately, such a procedure doesn't seem to work when implemented in a straight-forward manner and indeed Leighton[3] observes that ".. if the matrix were square, we would essentially be sorting rows and columns which is well known to leave entries arbitrarily away from the correct sorted position". This was in obvious reference to the row-major indexing scheme. Paradoxically things fall into place when one sorts the rows in a snake-like row-major form without increasing the complexity of the procedure. This simple algorithm, which we call **shear-sort** , will be formally introduced in the next section. Section III will provide the analysis of the algorithm. In section IV we discuss very efficient and simple VLSI implementation of the algorithm using only bubble-sort network and in section V we discuss a method to optimize the algorithm by simple manipulations.

Row-major

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Snake-like row-major

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 8 | 7 | 6 | 5 |
| 9 | 10 | 11 | 12 |
| 16 | 15 | 14 | 13 |

Shuffled row-major

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 6 |
| 3 | 4 | 7 | 8 |
| 9 | 10 | 13 | 14 |
| 11 | 12 | 15 | 16 |

Figure 1. Indexing schemes.

## II. Row-column sort

Let Q = [ $q_{i,j}$ ] be an $m \times n$ matrix onto which we have mapped a linear integer sequence S. Sorting the sequence S is then sorting the elements of Q in some predetermined indexing scheme. We suggest an iterative algorithm in which every iteration consist of the two basic operations:

(1) Row-sort - Sort independently all the row vectors of Q such that adjacent rows are sorted in opposite directions (alternate rows in the same direction). In a normal snake-like row-major indexing scheme, sort the first row from left to right (increasing). At the end of this step, $q_{i,j} \leq Q_{i,j+1}$ for all i = 1, 3, 5,..2p+1 and $q_{i,j} \leq q_{i,j+1}$ for all i = 2, 4, 6,..2p.

(2) Column-sort - Sort independently, in an ascending order from top to bottom all column vectors of Q. After this step $q_{i,j} \leq q_{i+1,j}$ for all j = 1, 2, ..n.

The shear-sort algorithm is defined as a repetitive application of steps 1 and 2 until one of the following conditions is satisfied :

(a) all the columns are sorted, i.e. no element has moved in the present column-sort **after a row-sort**, or

(b) no element has moved in the present row-sort **after a column sort**.

A step by step application of the algorithm is shown in Figure 2.

[a] Throughout this paper log will be assumed to be to the base 2 unless otherwise mentioned.

0190-3918/86/0000/0903 $01.00 © 1986 IEEE

| | | | |
|---|---|---|---|
| 1 | 4 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 5 | 8 | 12 | 16 |

Fig 2 a Rows and columns are sorted in row-major form but the array as a whole is unsorted.

| | | | |
|---|---|---|---|
| 1 | 4 | 9 | 13 |
| 14 | 10 | 6 | 2 |
| 3 | 7 | 11 | 15 |
| 16 | 12 | 8 | 5 |

Fig 2 b After the first rowsort in snake-like row-major indexing scheme.

| | | | |
|---|---|---|---|
| 1 | 4 | 6 | 2 |
| 3 | 7 | 8 | 5 |
| 14 | 10 | 9 | 13 |
| 16 | 12 | 11 | 15 |

Fig 2 c End of iteration 1

| | | | |
|---|---|---|---|
| 1 | 2 | 4 | 6 |
| 8 | 7 | 5 | 3 |
| 9 | 10 | 13 | 14 |
| 16 | 15 | 12 | 11 |

Fig 2 d Rowsort of iteration 2

| | | | |
|---|---|---|---|
| 1 | 2 | 4 | 3 |
| 8 | 7 | 5 | 6 |
| 9 | 10 | 12 | 11 |
| 16 | 15 | 13 | 14 |

Fig 2 e At the end of iteration 2 all elements are in their final sorted rows.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 8 | 7 | 6 | 5 |
| 9 | 10 | 11 | 12 |
| 16 | 15 | 14 | 13 |

Fig 2 f A snake-like row-major sorted sequence.

The reader may note that the algorithm will terminate into a sorted snake-like row-major sequence from the definition of such an indexing scheme. It is trivial to observe that, if the rows and columns are sorted in the directions corresponding to the algorithm, the array is sorted. Thus conditions (a) and (b) are equivalent and are necessary and sufficient conditions.

To give the reader an insight into the 'mechanism' of this sorting technique and throw some light on how such a simple procedure results in a sorted sequence we will obtain a very loose upper bound using an informal analysis. In the next section we will derive a tight upper bound using an indirect technique.

Consider the n smallest elements in Q and assume they are randomly distributed over the rows and columns of the array. The first column-sort will move these elements to the first row if initially they all happen to be in different columns. However, if all n smallest elements are in the same column (only possible for $m \geq n$), by virtue of alternating sorting direction on rows, a row-sort will have the effect of moving the elements on odd rows to the leftmost column of the array and the remaining half to the rightmost column. *This phenomenon is analogous to normal shear of a column and hence the name of this algorithm* . Clearly, at the beginning of the second iteration the n smallest elements have moved up into the upper half of Q. The second row-sort will then pair the elements on the two leftmost columns for the odd rows and on the two rightmost columns for the even rows. Following the same reasoning, it is not difficult to see that the column sort of the p iteration will move the n smallest elements of Q to the band defined from rows 1.. $\frac{n}{2^p}$. Without loss of generality we can assume n to be a power of 2 and conclude that the n smallest elements of Q will move to their sorted positions in at most log n iterations.

It follows from the above discussion that for m < n, the n smallest elements move into their sorted position in at most log m iterations. It is evident that once the first row is in place, it will continue to do so throughout the remaining iterations. Therefore we now face a problem of sorting the reduced array $Q_{-1}$ of dimension $m-1 \times n$. Each time a row is in its place, we reduce the problem to a smaller array on which the smallest elements are brought to the 'first' row in $\lceil log(m-k) \rceil$ iterations where k is the number of previously discarded 'first' rows. The total number of iterations to sort the array is thus

$$\sum_{k=0}^{k=m-1} \lceil \log(m-k) \rceil$$

which is bounded from above by mlog m.

In this simple analysis we assumed that after the 'first' row of array $Q_{-k}$ is in place all the remaining elements are randomly distributed in the reduced array $Q_{-k-1}$. This is not the case as we will show in the next section which gives a much better bound of O(log m) iterations.

## III. Analysis of the algorithm

We shall prove that shear-sort converges in O(log m) iterations, for an $m \times n$ array, by showing that the elements go within a specified distance of its final sorted row in every successive iteration. It will be seen that an arbitrary element (and hence all elements) goes within $\frac{m}{2^p}$ of its final sorted row after p iterations, from which the O(log m ) bound follows immediately. We will use here an application of the [0-1]principle (Knuth[6]) (for a direct combinatorial proof see Scherson & Sen[13]). For the purpose of applying the [0,1] principle, we will visualize our sorting algorithm as a sorting network of logm + 1 stages, where in each stage we sort all the rows in a row-major snake-like form followed by sorting all the columns (Figure 3).

Consider the simple case of a $2 \times n$ array containing arbitrary number of 0's and 1's.

(a) After the rows are sorted, the 0's in the first row will be packed to the left, while 0's in the second row will be pushed to the right. Let us denote the number of 0's in the first and second rows by $n_1$ and $n_2$, respectively (Figure 4).

(b) Depending on the value of $n_1 + n_2$, sorting the columns (in this case a simple compare exchange), will result in one of the following

Case 1 ( $n_1 + n_2 < n$ ) : the bottom row will contain only 1's and the top row will be a mixture of 0's and 1's.

Case 2 ( $n_1 + n_2 = n$ ) : the top row will contain only 0's and the bottom row will have only 1's.

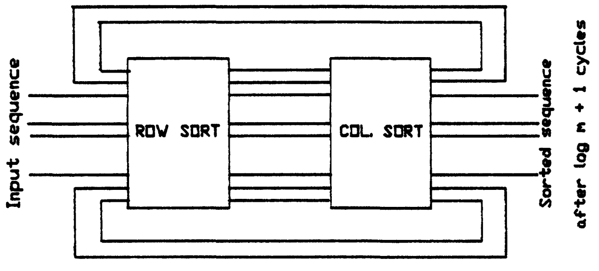


Figure 3. The algorithm as a sorting network.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Figure 4a. The top row is sorted from left to right and the bottom row from right to left(non-decreasing)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 4b. Afterthe column sort, one of the two rows(second in this case) contains only 0's or 1's. In a more favorable case, both rows are clean.

Case 3 : ( $n_1 + n_2 > n$) : the top row will contain only 0's and the bottom row will contain an unordered sequence of 0's and 1's.
Out of these cases, Case 2 is the most favorable, since the rows are already sorted and in the other cases we have a sorted row (first in case 1 and second in case 2). Also, the rows are mutually ordered i.e., all the elements of the top row are less than or equal to the elements of the bottom row. In the next row-sort the remaining row is ordered and sorting is complete. Henceforth we shall refer to a row as being 'clean', if it contains identical elements (only 0's or only 1's). Analogously, a row consisting of both 0's and 1's will be called 'dirty'.

The above discussion makes it clear that in a pair of adjacent rows, after the first iteration at least one of the rows is 'cleaned', i.e. it consists of only 0's or 1's. We will now extend the above phenomenon to an $m \times n$ array. Without loss of generality assume m to be a power of 2. After the row-sort of the first iteration we have 0's squeezed to the left on the odd- numbered rows and packed to the right on the even-numbered rows. Consider that a column sort consists of the following two stages
(1) For an element $q_{i,j}$, do a compare exchange with the element $q_{i+1,j}$, for i = 1,3 ..,m-1. For i = 2,4 ..,m, compare exchange $q_{i,j}$ with $q_{i-1,j}$. (This is equivalent to sorting m/2 pairs of adjacent rows independently.)
(2) Sort the columns normally i.e. across the entire columns.

It should be clear that decomposition of a column sort in the above manner is not going to affect the ordering resulting from a normal column sort. The extra (hypothetical) step makes the analysis much simpler. From the previous discussion, we know that following step 1 there is at least one clean row among every pair of odd-even rows. We may now consider an entire 'clean' row as a single entity during the column sort since a 'clean' row continues to be clean throughout the subsequent iterations. This follows from the observation that, when a 'clean' row of 0's is compared with a 'dirty' row across the columns, the clean row bubbles to the top row remaining 'clean'. Similarly a 'clean' row of 1's sinks to the bottom. Clearly, two clean rows, when compared across the columns cannot 'dirty' themselves. It may be easier to visualize this process of column sort being a bubble sort, in which all elements of a row vector are simultaneously compared to the corresponding elements in their adjacent row. The reader should note that the actual algorithm used for sorting columns does not necessarily have to be a bubble-sort since the ordering obtained is independent of the algorithm used. Row-sorts do not affect the clean rows.

| |
|---|
| 0 0 0 0 0 . . . . . . . . . . . 1 1 1 1 |
| 1 1 1 1 1 1 . . . . . . . . . . . 0 0 0 |
| 0 0 0 0 0 . . . . . . . . . . . 1 1 1 1 |
| 1 1 . . . . . . . . . . . 0 0 0 0 0 0 0 |
| |
| |
| |
| 1 1 1 1 1 1 1 1 . . . . . . . . . . 0 |
| 0 0 0 . . . . . . . . . . . 1 1 1 1 1 1 |
| |
| |
| |
| 1 1 1 1 . . . . . . . . . . . 0 0 0 0 0 |
| 0 0 0 0 . . . . . . . . . . . 1 1 1 1 1 |
| 1 1 1 . . . . . . . . . . . 0 0 0 0 0 0 |

Fig 5 a We can visualize the column sorting as
1. sorting on pairs of rows across all columns which 'cleans' atleast half the number of 'dirty' rows followed by
2. sorting on columns across the entire array. Note that 'cleaned' rows in previous step continues to remain 'clean' thereafter.

| |
|---|
| 0 0 0 0 0 . . . . . . . . . . . 0 0 0 0 |
| 0 0 0 0 . . . . . . . . . . . 0 0 0 0 0 |
| 0 0 0 0 0 . . . . . . . . . . . 0 0 0 0 |
| 0 0 0 0 0 0 . . . . . . . . . . . 0 0 0 |
| |
| |
| |
| 1 1 1 0 0 . . . . . . . . . . . 1 0 0 1 |
| 0 1 0 1 1 . . . . . . . . . . . 0 0 1 1 |
| |
| |
| |
| 1 1 1 1 1 . . . . . . . . . . . 1 1 1 1 |
| 1 1 1 1 . . . . . . . . . . . 1 1 1 1 1 |
| 1 1 1 1 1 1 . . . . . . . . . . . 1 1 1 |

Fig 5 b The 'clean' rows of 0's bubble up at the top of the array and the 'clean' rows of 1's settle down. Note that the 'dirty' rows are contiguous somewhere between the bands of 'clean' rows.

It follows, that after the column-sort step of iteration 1, we have at least half (m/2) rows 'clean', with the clean rows of 0's at the top and 'clean' rows of 1's at the bottom. The 'dirty' rows will occupy a band of at most m/2 rows in the middle of the array (Figure 5). Actually we may find less than m/2 'dirty' rows since two 'dirty' rows may mutually 'clean' each other during the column sort. In the course of the second iteration, half of the m/2 'dirty' rows will become clean by applying the same argument to the contiguous band of dirty rows while the clean rows continue to be clean. By a repetitive application of this phenomenon of cleaning half the number of dirty row with every successive iteration, we may conclude that there can be at most one dirty row after log m iterations. An additional row-sort orders this dirty row which actually separates the clean rows of 0's from the clean rows of 1's in the sorted array. Thus, for an initial array consisting of an arbitrary number of 0's and 1's **the shear-sort algorithm terminates successfully in logm +1 iterations.**

The classification of rows as 'clean' and 'dirty' in constructing the complexity analysis was extremely helpful since it reduced the burden of keeping track of every element in the array to only m rows. At this point the implications in a generalized array consisting of elements which may not be only 0's or 1's, is not clear. Nevertheless, we can state the following important result:

**Theorem 1** : An $m \times n$ array of elements can be sorted using shear-sort in time proportional to $\lceil \log m \rceil$+ 1 [m*(time for row-sort) + n*(time for column sort)].

The proof follows immediately from the previous discussion and the [0,1] principle. The tightness of this bound should be obvious from the the informal discussion at the end of the previous section. In the next section we will present an efficient implementation of this algorithm on a mesh-connected processor array.

The **average-case** performance analysis can also be simplified using the [0-1] principle. An important corollary following from the discussion of clean and dirty rows can be summarized as follows:

**Corollary 1** : A 0/1 array consisting of 'd' dirty rows initially, can be sorted using $\lceil logd \rceil$ iterations of shear sort.

Theorem 1 can be stated as a special case, since all m rows can be dirty initially. Consider O(m) 0's uniformly distributed in the $m \times n$ array, the other elements being 1's. Because of the uniform distribution, we may conclude, that on the average, m/2 rows will be dirty. From Corollary 1, the average number of iterations is logm - 2 or O(log m). This shows that the algorithm is optimal within minor variations of the basic shear-sort paradigm.

## IV. VLSI Implementation of shear-sort

In spite of the terrible performance of a normal single processor bubble sort, efforts have been directed towards obtaining efficient VLSI implementations ([5], [6]) because of the inherent simplicity of the algorithm. For this purpose a parallel version of bubble sort viz. odd-even transposition sort([6]) has been adopted. By using crossing sequence techniques, several researchers have shown that the optimal $AT^2$ bound for sorting n elements is O( $n^2$ ) in word model and O( $n^2 \log^2 n$ ) in bit model ([3],[5],[6]). The normal N/2 processor bubble sort where each processor performs one compare-exchange operation during each of the N iterations behaves horribly ( O( $n^3$ ) ) with respect to the $AT^2$ measure. This remains unchanged even by using completely pipelined bit-parallel comparison exchange modules to sort more than one problem instance. The pipelined scheme consists of O ( $n^2$ ) comparators which reduces the effective area by a concurrency factor(n) - the time remaining unchanged ([5]).

**Theorem 2** : The $AT^2$ performance of shear-sort implemented with a bubble-sort network is O($n^4 \log^3 n$) for $n^2$ elements.

Figure 6c shows the implementation of the shear-sort using a pipelined scheme where each of the n rows(columns) are pipelined through this sorting network. The 'Transpose/ Detranspose ' network aligns the array properly for the next column (row) sort. Following Leighton's[3] argument, the Transpose/Detranspose network needs n non-unit length wires and hence occupies $O(n^2)$ area where the transposition is performed in n parallel stages by hardwiring the rows to the corresponding columns (and vice versa- see Figure 6b). The bubble-sort network consists of $n^2$ comparators and thus the total area of the network is O( $n^2 \log n$ ). We will need 2n word steps to sort all n rows (columns). Each comparator is capable of performing a compare-exchange operation of two O(log n) bit numbers

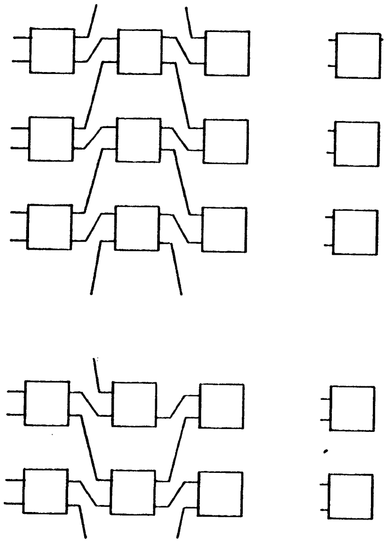


Figure 6a. An N/2 by N comparators Bubble-sorter
Each box represents a compare-exchange module

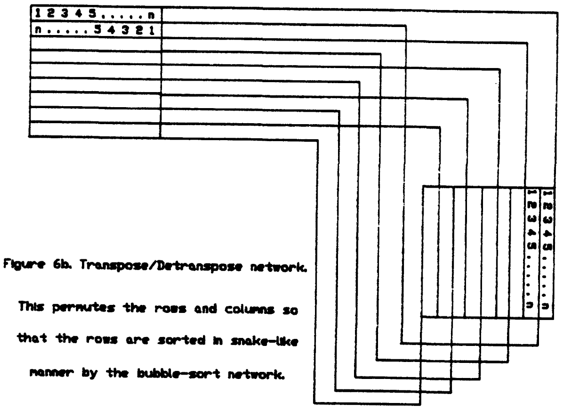


Figure 6b. Transpose/Detranspose network.
This permutes the rows and columns so that the rows are sorted in snake-like manner by the bubble-sort network.

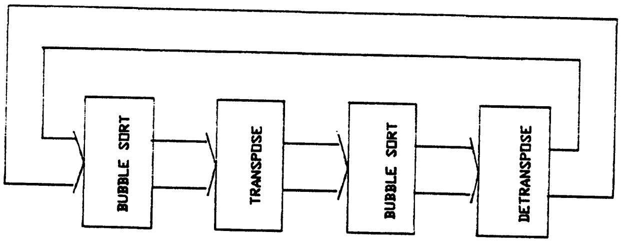


Figure 6c. Block diagram of Shear-sort using Bubble-sort network.
log n + 1 passes through this network will sort any sequence

in O(1) time. As observed previously the Transpose/Detranspose network also needs O(n) time for each iteration. Since we need log n iterations the $AT^2$ performance for this scheme will be

$$O(n^2 logn) \times O((n \log n)^2) = O(n^4 \log^3 n)$$

This is only O( $\log^3 n$ ) away from the lower bound. A similar result can be obtained for the bit model by using bit-serial compare exchange modules. Each compare-exchange module can perform a compare exchange operation every O(log n) time units and can fit into an O(1) by O(log n) unit rectangle. Thus the bubble-sort circuit occupies an area of O( $n^2 \log n$ ) units. The Transpose/Detranspose circuit consists of n non-unit length wires which occupy an area of O ( $n^2$ ) units. Each of these wires routs O(n logn) bits of data and thus takes O(n logn) units of time to complete the operation. The total time is O( $n\log^2 n$ ) and so the $AT^2$ measure for this scheme is O( $n^4\log^5 n$ ). This is only a factor of $\log^3 n$ away from the optimal which is O( $n^4\log^2 n$ ) for O( $n^2$ ), O(log n) bit numbers ([3]).

As noted by Thompson[5], this network needs very little in the way of control as no complicated operations are involved and may be more attractive than its $AT^2$ performance indicates (being a couple of log n factors away from the optimal). There is hardly any need to overemphasize that this scheme of sorting which exploits the powerful property of the algorithm has made bubble-sort comparable to some more sophisticated VLSI sorting networks as far as $area-time^2$ trade-off is concerned.

Figure 7 shows a more regular network for accomplishing the required compare exchanges, though it needs slightly more area (a factor of logn more than the previous approach). We have managed to incorporate the comparators for carrying out row and column sorts within the mesh- connected network, which is very similar to an ILLIAC IV like machine. The horizontal and the vertical compare-exchange modules are used during row-sort and column sort stages respectively. A ROW/COLUMN control line decides which set of the comparators is being used. A ROWDIR control line which controls the direction of compare exchange for the horizontal comparators, achieves alternating sorting directions in rows. A single row/column of elements 'oscillates' between two adjacent row(column) n/2 times, each of which carries out the required compare-exchanges needed for two iterations of odd-even transposition sort. The ROWDIR is changed during every clock cycle of the row-sort stage, so that adjacent rows are sorted in opposite directions. The overlapped comparators in the figure actually represent single comparators multiplexed between the row and column sorts. Each cell is drawn with two inputs, but only one of them is selected according to the row or column sort stages. After O(logm +1) such iterations the array is sorted in a snake-like ordering.

A few comments may be expedient at this point to emphasize the ease in implementation and the expandability of this scheme which are of major concern to any VLSI chip designer. The simplicity in the control structure, which is the result of purely iterative nature of the algorithm, and the regularity of the layout due to use of only nearest- neighbor type operations, make it ideally suitable for systolic implementation. Each chip containing a subset $k \times k$ of the $n \times n$ components will be connected to four similar chips in the four (North, East, South, West) directions, thus presenting no additional complications for chip interconnections. To account for the pin limitations in a chip, and packaging a maximum number of components in a single chip, we can use bit serial communications between the nearest neighbors. This would not change the asymptotic performance since a compare-exchange operation is of the same complexity as a bit-serial communication time of a 'b' bit words ( O(b) time). For example, in a 80 pin package, we need at most 10 pins for the global control signals leaving us with 70 pins for the inter-chip communication. Rounding this to the nearest power of two, i.e. 64, we can integrate a 16 X 16 mesh-connected compare-exchange modules in a single chip (for a $k \times k$ array, we need 4k data pins).

While this scheme is theoretically inferior to Lang et al.[8] by a factor of logn, it may prove to be more feasible for reasonable sized arrays by sacrificing a little time-performance in the bargain.

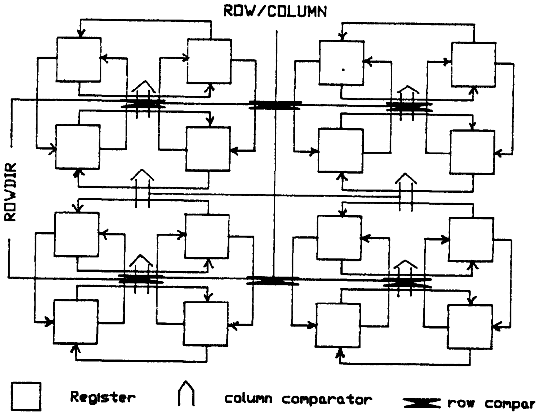


Figure 7. A regular VLSI layout for Shear-sort.

## V. Optimization of the shear sort

A further reduction in execution time for shear-sort is possible for a 'brick-wall' type sorting using a simple manipulation in the dimension of the array of $n^2$ elements. Recall from section III, that the band of 'dirty' elements keeps on shrinking by a factor of 2 in every iteration. This is the basis for an O(log m) (for an $m \times n$ array) iteration convergence. Thus, during the successive stages of the algorithm, it is enough to sort the elements in a column that fall within the 'dirty' band, instead of sorting an entire column during the column-sort stage. Another way of looking at it is that, since all the elements move within $\frac{m}{2^p}$ rows after p iterations (discussed more in [13]), we need to perform only $\frac{m}{2^p}$ steps of the brick-wall sort after p iterations.

For an $m \times n$ array, this means we perform m, m/2, m/4 .. iterations of brick-wall sort in successive column-sort stages. However, no such optimization seems possible for the the row-sort stage since the algorithm is non-adaptive so that we keep on performing all the 'n' compare-exchange steps throughout the course of execution. This yields a complexity of n(logm +1) + 2(m - 1) compare exchange steps. For $n^2$ elements, this can be expressed as

$$\frac{n^2}{m}(logm + 1) + 2(m - 1)$$

By choosing m = $\frac{n\sqrt{logn}}{\sqrt{2}}$, we get a complexity of

$$O(\ 2\sqrt{2}n\sqrt{logn} + \frac{nloglogn}{\sqrt{2logn}}\ )$$

The second term can be approximated to o(n) since $\frac{loglogn}{\sqrt{logn}} = o(1)$, thereby yielding the following result:

**Theorem 3:** A rectangular array of $n^2$ elements can be sorted using shear-sort in time proportional to O( $n\sqrt{logn}$ ).

The behavior of this function is very close to linear since $\sqrt{logn}$ is less than 5 for well over tens of millions i.e. more than the practical size of any sorting chip. In fact it outperforms most of the well-known algorithms for number of elements below a couple of tens of thousands which is not rare in practical situations. The cautious reader may have noticed that an extra compare exchange step is needed during the column sort stage depending on the boundary of the dirty band. If the dirty band starts from an even row during all the iterations of column sort the complexity function will be affected by an additional log m steps, which is negligible.

This improvement can be very easily incorporated into the VLSI implementation shown in Figure 7. The number of iterations of odd-even transposition sort during row and column sort stages can be easily controlled by the number of 'oscillations' between adjacent rows(columns) of comparators.

## VI. Conclusions

The feasibility of sorting a rectangular array of elements by sorting rows and columns was demonstrated. The algorithm was shown to execute in at most O(log m) iterations and in section III we traced the data movement during each iteration. Recall that we showed that all elements moved within O( $\frac{m}{2^p}$ ) rows of their final destination row in O(p) iterations. Also taking advantage of this phenomenon it is possible to optimize a simple algorithm like bubble sort. Sorting a row(column) is actually sorting $\sqrt{n}$ elements in a n element sequence which may be expensive. However this basic operation may be optimized by using a sorting network as was demonstrated in section IV. The complexity of this algorithm is more appropriately expressed as O ( $\sqrt{n} \times k \times \log n$ ) for an 'n' element sequence organized as a square array, where k is the time for sorting $\sqrt{n}$ elements and this is O( $\sqrt{n} \log n$ ) for a single processor sort which was used as a basis for the multiprocessor implementation. For the sorting network allowing pipelining this turns out to be O( $(\sqrt{n} + k) logn$ ) which at the best will give us a time of O( $\sqrt{n} \log n$ ).

This algorithm can be also executed effectively in any MIMD type machine which allows independent access to rows and columns in a shared memory model. A multiprocessor architecture which can implement the algorithm elegantly because of its orthogonal access to the memory banks and originally conceived as a high performance graphics system which can draw very fast vectors of any orientation ([11]) is discussed in Scherson & Sen[13]. In a similar architecture, Tseng et al.[12] propose a combination of bitonic sort and a single processor O(nlogn) sort, which can achieve identical complexity figures when implemented on mesh-connected processors. That process also sorts rows and columns in successive stages though it requires a more complicated control structure being recursive in nature. The direction of sorting in rows change dynamically in successive levels of recursion, as also does the number of independent sequences in the columns.

In an effort to optimize the algorithm further, by trying to reduce the cost of sorting entire rows and columns by partial sorting an interesting variation is mapping a two dimensional odd-even transposition sort on this row-major snake-like indexing scheme. Each iteration will consist of performing compare-exchange on elements ( $x_{2i-1,j}$ , $x_{2i,j}$ ) on all rows independently followed by the same procedure on all the columns ( $x_{i,2j-1}$ , $x_{i,2j}$ ) and then repeating the same with the elements ( $x_{2i,j}, x_{2i+1,j}$ ) in all rows and elements ( $x_{i,2j}, x_{i,2j+1}$ ) in all columns. It is not difficult to see that such an algorithm will result in a sorted sequence. It was speculated in [13] that the upper bound for the number of iterations may be $\sqrt{n}$ giving an optimal and an even simpler algorithm. However it turns out to be totally inefficient running into O(n) iterations like a normal bubble-sort which makes us hopeful about the optimality of shear sort within similar variations.

### References

[1] C.D. Thompson & H.T. Kung , "Sorting on a Mesh-Connected Parallel Computer," Communications of the ACM, vol 20, no. 4, April 1977.

[2] D. Nassimi & S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers, vol C-27, no 1, January 1979.

[3] T. Leighton , "Tight Bounds on the Complexity of Parallel Sorting," IEEE Transactions on Computers, vol C-34, no. 4, April 1985.

[4] D. Gale & R.M. Karp ," A Phenomenon in the Theory of Sorting," Journal of Computer and System Sciences, no. 6 , 1972.

[5] C.D. Thompson, "The VLSI Complexity of Sorting ," IEEE Transaction on Computers, vol C-32, no. 12, December 1983.

[6] D.E. Knuth, "The Art of Computer Programming," vol. 3, Addison-Wesley, 1973.

[7] J.D. Ullman, "The Computational Aspects of VLSI ," Computer Science Press, 1984.

[8] Lang Hans-Werner et al., "Systolic sorting on a Mesh Connected Network," IEEE Transactions on Computers, vol C-34, no. 7, July 1985.

[9] K. Batcher, "Sorting Networks and their applications," in *Proc.* AFIPS Spring Joint Comput. Conf, vol 32, 1968.

[10] M. Kumar & D.S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," IEEE Transaction on Computers, vol C-32, March 1983.

[11] I.D. Scherson, " A Parallel Processing Architecture for Image Generation and Processing, " Preliminary Report, E.C.E. Report No. 84-20, U.C.S.B., August 1984.

[12] P.S. Tseng, K. Hwang and V.K. Prasanna Kumar, " A VLSI based Multiprocessor Architecture for implementing parallel algorithms, " International Conference on Parallel Processing, Aug. 1985.

[13] I.D. Scherson and Sandeep Sen, " A Characterization of a Parallel Row-column sorting technique for rectangular arrays, " ECE Technical Report No. 85-14, U.C.S.B., August 1985.

# Trace-Driven Simulations of Parallel and Distributed Algorithms in Multiprocessors†

**Michel Dubois, Fayé A. Briggs*, Indira Patil* and Meera Balakrishnan**

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0781
(213) 743-8080

*Department of Electrical and Computer Engineering
Rice University, P. O. Box 1892
Houston, Texas 77251
(713) 527-8101 ext. 3595

## Abstract

A methodology to derive trace-driven simulations on a uniprocessor for evaluating the performance of parallel algorithms in multiprocessors is described. Based on the desired degrees of accuracy and speed of the evaluation, we interleave the independent traces, which are obtained by actually running each of the cooperating processes on a uniprocessor and capturing global events. Since only global events are captured, data reduction is very high. Using this trace, we can extract parameters which are applied to a variety of progressively refined analytical and hybrid models to study the impact of algorithmic and architectural features on performance. Finally, design spaces of architectures and algorithms can be explored through successive refinements of the simulations.

## 1. Introduction

Multiprocessor systems are emerging in the commercial world as cost-effective systems for satisfying throughput demands of the mini and supermini markets [ARC85]. It is foreseen that such systems can deliver not only throughput, but also speed-up. Many languages have been developed to take advantage of multiprocessor systems [AND83, BAR83, TED84]. They introduce the notion of a process to the users of these machines. The execution of a program can be implemented by a set of concurrent and cooperating processes running simultaneously on different processors. Speed-up can thus be obtained for a given program and not only for a batch of programs.

Many results have been derived on the computation complexity of parallel algorithms [HEL78, KRU83, GAN84]. However, the interaction between software and hardware is so complex in MIMD machines that simple complexity studies are often inadequate. Some attempts at modeling some algorithms on specific multiprocessors are presented in [DUB82a-b, VRS84, VRS85]. However, concurrency control mechanisms in the algorithm, such as scheduling and synchronization, and data allocation effects are most of the time neglected for reasons of analytical tractability. Moreover, it is not clear how the parameters used in the models can be estimated. Simulation technique is often the method of choice to verify analytical approximations. Of course, the most accurate evaluation method is measurement on an actual prototype[GEH82].

Some major simulators have been implemented to predict the performance of algorithms on multiprocessors. The first one, developed at Lawrence Livermore Laboratory[AXE84], uses a trace-driven approach but traces every instruction with the detailed model of the S-1 multiprocessor. Therefore, the simulation is complex and must contain hardware timing information which is specific to the S-1. The NYU simulator is also an instruction level simulator[ABR84], written in CDC Cyber assembly language. The Cedar simulator[ABU85], written to take advantage of Parafrase[KUC84], allows the user to select statements whose execution may be simulated. A parallel algorithm simulator, called PSIMUL, was written at IBM[SOK86]. This is also an instruction level simulator, but with no timing facility. The trace is produced by interleaving blocks of records of the individual traces of each process, where the size of each block is determined by the time slice interval on the host machine.

In this paper, we present a different methodology for developing trace-driven simulations of the execution of algorithms in multiprocessor systems. It is not specific to any particular architecture and hence can be used to evaluate different machines. The only limitation we impose is on the computational model which is described in the next section. The complexity of the trace-driven simulations as presented here is intermediate between the complexity of analytical models and detailed, cycle-by-cycle simulations. In our methodology, every instruction of the algorithm is executed. However, only global events are traced. Depending on the accuracy desired, the interleaving of the individual traces is derived from virtual or real time components. The trace is then applied to the performance prediction. In our technique, we extract values for algorithmic and architectural features from the trace and apply them to analytical or hybrid simulation models to predict the performance. Of course the detailed simulation with real timing is also accomodated and is necessary for the validation of the more approximate models.

The paper is structured in two parts. In the first part, the computational model and the basic multiprocessor environment we assume is described. In the second part, the architecture for the simulator is analyzed and different levels of complexity of simulations are identified through a careful analysis of the tasks involved in the trace-driven simulation of multiprocessor programs.

## 2. Computational Model and General Hypotheses

Parallel and distributed algorithms can be classified broadly in two classes, with respect to the complexity of analyzing their performance: *data-independent* and *data-dependent* algorithms. In data-independent algorithms the

† This research is supported by an NSF Research Initiation Grant No DMC-8505328 and the USC Faculty Innovation Fund

0190-3918/86/0000/0909 $01.00 © 1986 IEEE

performance is independent of the *values* of the data. These algorithms include many numerical algorithms. The performance of data-dependent algorithms depends on the *values* of the data. Such algorithms which include most symbolic algorithms are more difficult to analyze than data-independent algorithms because of the large number of configurations to consider. Compounding this difficulty is the possibility that the data-dependent algorithm may be non-stationary; that is, the behaviour of the algorithm execution varies with time.

The parallel programming model we will use is based on practical concurrent programming language constructs [AND83]. The model allows the user to specify the concurrency and cooperation between the processes of the program. This technique is different from the compiler generated parallelism which have been shown to be effective for numerical algorithms[KUC84], but performs somewhat poorly on nonnumerical algorithms[LEE85]. The poor performance may be due to the fact that most nonnumeric algorithms are data-dependent and often non-stationary. Therefore, the programming model adopted here is all the more applicable to nonnumeric algorithms.

It is assumed that each processor has a large instruction cache or memory so that accesses to program code and local variables are done locally and do not generate global traffic. This assumption is justified by the fact that the program size for the algorithms considered here is usually very small compared to the total number of data accessed. If an algorithm does not fit this description, then the modeling technique must be refined to include instruction fetches.

In the type of machine envisioned in this study, the operating system is minimum. The multiprocessor is used as an attached multiprocessor. Basic operating system mechanisms supported by the methodology described here are flexible enough to support most, if not all, possible algorithms. These mechanisms are for process scheduling, sychronization, and communication[LIN81,DUB82b]. In addition to these mechanisms, data partitioning and allocation, and process granularity may also affect the performance of the algorithm[JON80, GAJ85]. The effects of these mechanisms can be derived from the global trace events by scanning the event attributes in order to extract statistics for the multiprocessor. The computation homogeneity or heterogeneity of the set of concurrent processes is also an important feature of the algorithm. Analyzing heterogeneous computations analytically is often impractical.

In a multiprocessor system, scheduling may be static or dynamic. Both scheduling mechanisms are therefore supported by the methodology. In the static scheduling policy, the binding between processors and processes is determined at compile time. Static scheduling has low runtime overhead but is less flexible and often results in poor load balancing. In the dynamic scheduling policy the time of binding of processors to processes is delayed until runtime. A process can be executed by any processor that becomes available. Dynamic scheduling results in more runtime overhead but also in better load balancing.

In the attached processors under study, scheduling support is minimum. Therefore, the system relies on self-scheduling by the application program as performed in the HEP multiprocessor via a local or global ready list[GAJ85, SMI81]. In this case, the processes participating in the algorithms execute code to fork other processes and to access the job queues. In the static scheduling, each process is forked on a specific processor, determined at compile time and the process runs to completion on the designated processor. Each processor has a run list. When a process forks a new process on a different processor, it puts a descriptor for the process in the queue of the target processor and executes a static_fork instruction. The effect of this instruction is simply to wake up the target processor in case it is idling. The target processor automatically reads the top of its run list, loads the descriptor and starts execution.

In the more complex, dynamic scheduling case, a run list is shared by several processors. The dynamic_fork procedure is somewhat similar to the static_fork procedure but, the job descriptor is put in the shared run list if an attempt to bind the new process to a processor fails because there is no free processor.

In both static and dynamic scheduling, a processor that terminates a process attempts to get a new descriptor from the private or shared list before entering the idle state. The termination of a process is indicated by the execution of a *terminate* primitive.

Our methodology is applicable to most practical multiprocessor architectures. These include shared-memory multiprocessors and distributed systems with or without cache memories. For multiprocessors with cache memories, the impact of software or hardware-enforced coherence on algorithm performance can be evaluated since only global events may cause *cross-interrogates* [DUB82c].

The basic set of language constructs we have used to support multiprocessing are the process creation and control primitives: *fork/join, suspend, activate, create, terminate and resume.* In addition, we provide simple synchronization primitives such as *spin-lock, suspend-lock* and *unlock*. Knowing that each synchronization primitive has its own advantages and disadvantages, we also provide other data-level primitives such as *fetch-and-add* [GOT83], *full/empty bit* [SMI81]. For distributed systems, message-based primitives such as *send/receive* [SHA84] are used.

## 3. Modeling Methodology

The methodology uses a hybrid and hierarchical approach to modeling by incorporating trace-driven simulations and analytical models[SCH78, KUM80]. In a multiprocessor, the events to trace are accesses to shared data or message-passing through the network. However, the methodology allows for the tracing of *any* selected set of events.

### 3.1. Interleaving of the traces

In order to evaluate the performance of arbitrary parallel and distributed algorithms on proposed or existing multiprocessors, the traces must be generated through the interpretation of the multiprocessor algorithm. This is especially important for algorithms whose behavior is highly sensitive to data values. Therefore the traces are derived *while* the multitasked algorithm is executed.

The events of importance in a multiprocessor system are accesses to shared data (and, in particular, synchronization), message-passing operations, and the creation and purging of processes. In between the execution of such events, processors execute locally and do not affect each other. One problem is to interleave the traces of events generated by each processor. This interleaving is needed because the simulation technique described here is a technique for a uniprocessor system.

This section analyzes three levels of accuracy in trace interleaving: interleaving based on rank, on a virtual time approximation or on physical time.

### 3.1.1 Rank Interleaving

A first approach is to interleave the P traces according to the rank or position of each event in each individual processor trace. Assume that the P processors are ordered in an arbitrary sequence and numbered from 0 to P-1. In the interleaved trace, global event $k$ in processor j occupies position $kP+j$. This interleaving techniques is said to be ***based on rank***. This approach is acceptable for simple multitasked algorithms where the computation is homogeneous, and the real elapsed time between two successive events is approximately the same for all processors at any given time. This is the case for the simple PDE example given in section 5 of this paper. It is an unacceptable approximation for more complex algorithms such as quicksort, in which the execution behavior of the processors is not homogeneous all the time.

### 3.1.2 Virtual-Time Interleaving

To improve the reliability of the simulation, the traces are interleaved according to the rank of each event in its trace and according to a timing estimate such as the instruction count between two events generated by a process. We call this interleaving an interleaving based on a ***virtual time approximation*** [LAM78, JEF83]. The reason is that the timing estimate is not a physical time. In this technique, the trace for each processor is a set of records of the type

$$< pn\ ,\ e\ ,\ \delta t\ ,\ parameters\ >.$$

$\delta t$ is an estimate of the time complexity of the work (in number of instructions, or in number of instruction cycles) performed by the processor since the last global event from the processor. In our facility, we derive this number of instruction cycles from a mechanism first developed at AT&T Bell Laboratories[WEI84] and modified at Rice University[JUM85]. $pn$ is the number of the processor causing the event, $e$ is the event type and *parameters* are the parameters of the event. Examples of event types include fetch, store, test_and_set, block, fork and terminate. Parameters may be, for example, a shared-memory address, a process descriptor, or a run list pointer. Table 1 indicates the various trace events with a definition of their parameters. The virtual time approximation defines a partial ordering of the global events produced by the processors as follows. For processor j there are a total of $n_j$ events numbered $E_{j,1}$, $E_{j,2}$, ..., $E_{n,n_j}$ and the time between events $E_{j,i}$ and $E_{j,i+1}$ is $\delta t_{j,i} = \delta t + t_b$, where $\delta t$ is the virtual time increment given in the trace record and $t_b$ is the virtual time that the processor has been blocked since the last global event. Then, the virtual time for the $k$th event of processor j is denoted by $T_{j,k} = \sum_{i=0}^{k-1} \delta t_{j,i}$. By definition, the partial ordering is such that $E_{j,m} \leq E_{l,n}$ if and only if $T_{j,m} \leq T_{l,n}$. A partial ordering of events is not sufficient for the simulation because events having the same virtual time can only be scheduled one at a time in the multiprocessor simulator implemented on a uniprocessor. In order to define a total ordering of events, a selection algorithm is used to select among trace events having the same virtual-time stamp.

The interleaving based on the virtual time approximation is attractive because it is independent of actual multiprocessor organization, data allocation strategies, and technological parameters. It only depends on the algorithm and on the uniprocessor architecture. If it is valid, the interleaved trace may be reused multiple times to study various system configurations and technology trade-offs. This aspect has been a major advantage of trace-driven simulations for the analysis of uniprocessor systems. In the analysis of multiprocessor systems this advantage may be lost in some cases because of a phenomena called ***trace-shifting*** and explained below.

### 3.1.3 Trace-Shifting Phenomena

In the virtual time approximation, the effect of the multiprocessor environment is not included in the algorithm to interleave the traces. There are many ways in which the system under study may affect the interleaving of the traces. For example, in a cache-based system a cache miss in a processor delays the occurence of the next global event in this processor, and therefore affects the interleaving of events. If this effect is symmetric with respect to all the processors (i.e. if the same type of delay is encountered by all processors at some random times), then the virtual time approximation is valid. When the effect is not symmetric over a long period of the simulation, a long-term phenomena called ***trace-shifting*** may occur in which the traces get out of phase as compared to the order of execution on the real machine. For example, in the cache-based system, some processor may be in a phase in which misses occur in bursts because it is accessing a new data structure; in a distributed global memory system, such as the Cm*[GEH82], trace shifting may occur if the data is allocated asymmetrically, even if all the processors are performing the same work. For example, if all data are stored in the memory of processor 0, processor 0 will access memory faster and therefore, in the real system, it will execute faster than the other processors. In the simulation, however, all processors would execute at the same speed. Therefore, the trace of processor 0 has shifted with respect to the P-1 other traces. Trace shifting may result in considerable errors in some cases. For data-dependent algorithms and dynamic scheduling, it may result in a totally different execution sequence as compared to the execution on the real machine.

### 3.1.4 Physical-Time Interleaving

In general, the only way to avoid trace shifting is to include physical (real) timings in the trace-interleaving algorithm. We call this type of interleaving an interleaving based on ***physical time***. First, the "$\delta t$" provided in the trace record of the individual processor traces must be a physical time estimate as accurate as possible. Second, a penalty must be added to account for data access delays such as cache miss penalty or remote memory access penalty. The penalty depends on various factors including the multiprocessor configuration, data allocation and technology parameters. A fixed penalty for each access type may also result in trace shifting, because of conflicts for shared-resource accesses. If some processors are systematically conflicting while accesses from other processors are conflict-free, the traces will shift in the case where a fixed penalty is added for each access. The penalty should in this case be variable and include conflict delays. A full fledged simulation of the multiprocessor may be needed to determine accurately the total access delays through the interconnection network and through the memory system.

In some cases, when the algorithm interpretation is not affected by the shifting, the problem of trace shifting can be compensated for during the simulation. That is, traces can be realigned periodically: a total penalty is accumulated for each processor during the simulation; when the difference between the penalties becomes too large, the traces are synchronized by subtracting their total penalty from the time stamp of their next event.

### 3.1.5 On-the-fly and Off-line Tracing

Finally, tracing may be done *on-the-fly* or *off-line*. If it is done *on the fly* then the interpretation of the algorithm, interleaving of the traces, simulation and collection of statistics, are done in one single run. No trace file is derived, saved and reused. This mode of operation is desirable if the interleaving must be based on physical time, or if the overhead of handling large trace files (on tape or on disk) is large compared to the overhead of rederiving the trace each time. Note that the trace-driven approach is still applicable in this case even if the trace cannot be reused. Indeed, partitioning the simulation into two parts (the derivation of the trace and the simulation ) removes all local events from the global event list and results in increased efficiency. *Off-line* tracing is done in two phases. In the first phase, the interleaved trace is derived and stored in a trace file. The trace file may then be reused several times for multiple simulations of different system configurations, data allocation and/or technological parameters. The off-line interleaving of traces only makes sense when the trace is re-usable, i.e., when the interleaving is based on rank or on the virtual time approximation.

### 3.2. Description of Simulator Architecture

The simulator is divided into three types of modules (see Figure 1): the trace-generator, trace-scheduler, and MP-simulator. The function of a *trace-generator* module is to generate the trace records as they are requested by the scheduler module. For each algorithm and each partitioning strategy, one must write a set of trace generator modules. The *trace-scheduler* is the driver of the whole simulator: it determines the next process that will generate a trace record according to the trace interleaving strategy and a *selection algorithm,* it requests the next trace record from the selected process, and submits the record to the MP simulator. This last module may be a detailed simulation or a program to gather statistics of the dynamic behavior of the algorithm on the architecture.

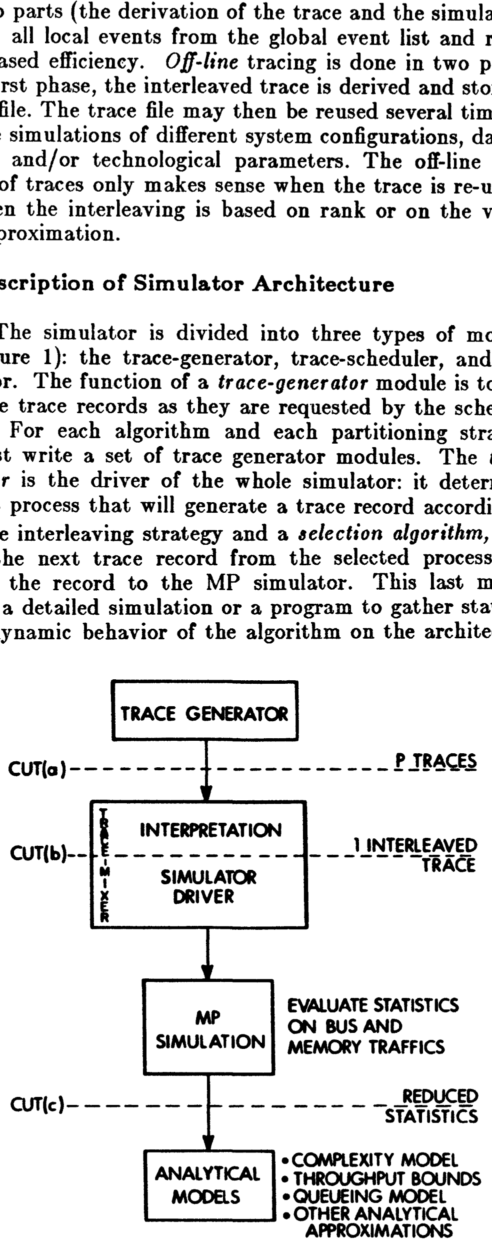


**Figure 1.** Activity flowchart of trace-driven simulation.

These statistics are then used as parameters in simple analytical models such as complexity, throughput or queueing models.

An activity flowchart for the overall performance analysis is also outlined in Figure 1. Several "cuts" are possible in the flowchart. A cut means that the simulation is run in two independent parts, and that intermediate results from the first part must be stored. A cut at (a) requires storing one or several files on tapes or disks, for each processor; a cut at (b) requires storing one single thread of trace records, a cut at (c) reduces the information that has to be stored to statistics such as mean traffic values, variance of memory service time, histogram of execution time between two successive global events.

Cuts at (a) and (b) result from off-line tracing while a cut at (c) results from on-the-fly tracing. Depending on the interleaving strategy, on the algorithm being traced, and on the overhead of file manipulation, a cut at (a) or (b) may be desireable in order to reuse the trace.

### 3.2.1 Trace-Generator Module

A trace-generator module of the simulator executes the process and produces each global event from that process. The process is written as a procedure specifying the task to be performed augmented by statements which log global-trace event records as these events are generated by the process. For each "forked" process there exists a *trace generator.* A trace generator is completely specified by a process and by a descriptor defining the data set to which the process is applied.

The implementation of a trace generator is based on the coroutine principle. Few basic functions are provided to facilitate the "forking", activation and suspension of the trace generators. The first function is called *fork* (proc, $p_1, p_2, \ldots, p_n$). This function creates and activates a process of type "proc" with the associated parameters $p_1, p_2, \ldots, p_n$, and returns a unique process identification number, *pid* . Subsequently, the process can start execution when the trace-scheduler performs a *transfer* (*pid*) to the process. The trace generator suspends itself and transfers control to the scheduler by executing a *suspend* (). This is essentially a coroutine call to the trace scheduler. In the trace generator, a trace event is logged into the event list of the scheduler by *log_event* (pn,event_type, time_incr,parameters). The "time_incr" attribute is the virtual or physical time increment from the last event logged by that processor. The next time the trace scheduler schedules the trace generator for event production it again executes a *transfer* (*pid*). These and other primitives were packaged into a Concurrent C preprocessor[MAD86].

### 3.2.2 Trace-Scheduler Module

The trace scheduler is designed to invoke the trace generators. Its first function is therefore the interpretation of the multitasked algorithm. The second function is to produce a single thread of totally ordered events from the partially ordered set resulting from the interpretation.

The trace scheduler maintains four types of lists: the event list, ready list, run list and the blocking lists. The run list, which could be local or shared, contains the set of forked processes waiting for a processor. The blocking lists support blocking synchronization primitives such as suspend-locks and synchronous message primitives. The event list contains at every time as many nodes as there are active processors in the multiprocessor. A node of the event list includes a processor_number field, an event descriptor field, a time stamp field, and other parameters. The time stamp is computed on the basis of the time increment provided in the trace record and the system time at the arrival of the event. Nodes in the list are ordered according to increasing time stamp values. The scheduler selects the processor whose node is at the top of the list as the producer of the next global event. If several events at the head of the event list have equal time stamps, then these events are moved to the ready list where a ***selection algorithm*** is invoked to schedule the events in the ready list in turn. Possible selection algorithms are round-robin, priority, or random; the choice of an algorithm should reflect the interconnection network arbitration protocol.

At initialization of the scheduler, the first event, "start", and the process identification corresponding to the algorithm to be run is placed in the event-list. While the ready list is not empty one event at a time is selected for event processing based on the event type in the multiprocessor simulator. The processor which issued that event is then resumed to generate the next trace event, which is logged into the event list. When the ready list is exhausted, the scheduler starts a new cycle by moving events with equal time stamps to the ready list for scheduling. Note that if the application has been correctly interpreted, the algorithm terminates when the event list is empty.

The possible event types for the scheduler are given in Table 1. In the following, the implementation of some of these primitives in the trace scheduler are discussed. On a read, write, test_and_set or a reset event type, the scheduler simply submits the request to the MP simulator and then invokes the next trace element. Programs for spin_locks and suspend_locks are given in Figure 2. A suspend_lock is more difficult to implement and requires the additional primitive ***block*** (x) to indicate to the trace-scheduler that the execution of the process is to be suspended. The scheduler puts the processor in a blocking list associated with x, and thereby suspends it from invoking its trace generator. On executing an ***unblock*** (x) in a trace generator, the scheduler randomly selects one of the processors blocked on x for reactivation by placing it in the event list.

**Table 1**

**Possible Trace events and their parameters**

| **Events** | **Parameters** |
|---|---|
| READ, WRITE,<br>TEST_AND_SET, RESET | x |
| BLOCK | x |
| FORK | p_id |
| TERMINATE | pn |
| START_PROCESS | p_id |
| STOP_PROCESS | p_id |
| SYNC_SEND, SYNC_RECEIVE | m, p_id |
| ASYNC_SEND, ASYNC_RECEIVE | m, p_id |
| SYNC_BROADCAST | m |
| ASYNC_BROADCAST | m |
| READF, WRITEE | x |
| FETCH_AND_ADD | x, d |

x: shared memory address; p_id: process identification; pn: processor number; m: message; d: increment.

To illustrate the use of these functions in implementing a trace generator, we consider the simple example of a parallel and dynamic *quicksort* performed on an array A[1....n]. A description of this algorithm is given in [DEM82]. In this paper we present a skeleton of the algorithm in Figure 3 and illustrate a segment of the implementation of the trace scheduler with dynamic forks in Figure 4. For simplicity, we have excluded all the necessary declarations and calls to the MP simulator. The quicksort routine picks up a "descriptor" from the run list via a spin-lock and splits up the array into two subarrays about a pivot elememt A[v] [BAA78]. A new quicksort process is forked to sort one of the subarrays while the current process sorts the other subarray. An array with a number of elements less than "threshold" is sorted by ***insertion sort.*** During the forking of the new process, a "descriptor" is also placed on the run list via a spin-lock.

```
/**********************************/
/* Microcode for spin_lock(x)     */
/* and suspend_lock(x)            */
/**********************************/
spin_lock(x)
{
   repeat y := test_and_set(x);
     until y = 0;
}
suspend_lock(x)
{
   y := test_and_set(x);
   if (y = 1) block(x);
}
/**********************************/
/* Trace generators for           */
/*spin_lock(x) and suspend_lock(x)*/
/**********************************/
gen_spin_lock(x)
{
      y := x;
      if (y = 0) x := 1;
      log_event(pn,"tas",2,x);
      suspend();
      while (y = 1)
      {
         y := x;
         if (y = 0) x := 1;
         log_event(pn,"tas",2,x);
         suspend();
      }
}
gen_suspend_lock(x)
{
    y := x;
    if (y = 0) x := 1;
    log_event(pn,"tas",2,x);
    suspend();
    if (y = 1)
    {
       log_event(pn,"block",b_time,x);
       suspend();
    }
}
```

**Figure 2.** Trace generators for spin_lock and suspend_lock.

The trace generator for the quicksort will include two statements wherever a reference to global data A[*] is made in the algorithm. These are *log_event* (pn,event_type,time_incr,parameters) and ***suspend*** (). Note that the trace generator for spin-lock, shown in Figure 2, will replace the spin-lock in the quicksort process. Complete analysis of the quicksort in multiprocessors with distributed global memory system can be found in [BAL86, PAT86].

### 3.2.3 Multiprocessor Simulator Module

The complexity of the multiprocessor simulator depends on the class of multiprocessor and the level of details of the simulation. We illustrate these complexities by considering single-bus multiprocessor architectures. The simulator collects statistics on the bus and memory traffic generated by the global events transmitted by the trace scheduler. The statistics collection for distributed systems and for systems with distributed global memory merely consists in the updating of traffic counters. The simulator for cache-based machines is by far the more complex. It must keep track of the cache contents and implement cache replacement and coherence algorithms. The cross-interrogate and traffic statistics may then be used in analytical models to compute the degradation on system performance during the execution of the algorithm.

```
quicksort(A,l,u)
{
pn = getmyPE();
dequeue(global_run_list) ;
while (( u-l) > threshold)
{
   v = split(A,l,u,pn);
   forkprocess(quicksort,A,l,v-1,pn) ;
   l = v + 1 ;
}
insort(l,u) ;
terminate(pn) ;
}

forkprocess(proc,A,l,u,pn)
{
pd = fork(proc,A,l,u) ;
enqueue(global_run_list) ;
log_event(pn,"fork",fork_time,pd) ;
suspend();
}

terminate(pn)
{
         log_event(pn,"term",delta_t) ;
         suspend()
}

dequeue(global_run_list)
{
 spin_lock(global_run_list) ;
 dummy_dequeue_to_ensure_serial_access
         to_global_run_list() ;
 unlock(global_run_list) ;
}

enqueue(global_run_list)
{
 spin_lock(global_run_list) ;
 dummy_enqueue_to_ensure_serial_access
         to_global_run_list() ;
 unlock(global_run_list) ;
}
```

**Figure 3.** Dynamic Quicksort program skeleton.

## 4. Hybrid Simulation Models

The traffic estimates obtained through enumeration of bus access or shared memory access events in the simulation trace can be used in analytical models. The analytical model can be as simple as throughput bounds or queueing models. Many models have been developed to analyze the performance of multiprocessors once the traffic and latency values are known[HOO77,DUB82a,MUD84].

In general, the more precise an analytical model is, the more system parameters are needed. Also, the model may become quite complex to exploit. We feel that a great deal of information on the performance of multitasked algorithms can be obtained by simple throughput bounds.

The models in which the parameters are estimated as averages over the entire run of the application is only valid when the computation is approximately *stationary* in time. By stationary, we mean that its execution behavior with respect to causing the global events remains constant during the overall processing. If the behavior varies (such is the case for the sorting algorithm, which is non-stationary because the decomposition varies during the execution), then the models presented above must be applied successively to the different *phases* of the algorithms. During each phase, it is considered that the characteristics of the computation remain constant.

```
trace_scheduler()
{
Initialize ;
pd = fork(quicksort,A,l,u) ;
log_event(NULL,"start",0,pd) ;
while (event_list NOT NULL )
{
   EVENTS_TO_READY_LIST ;
   rdy_ptr = get_next_ready_event ;
   while (rdy_ptr NOT NULL)
         {
         pd_old = bind[rdy_ptr -> pn];
         CASE(rdy_ptr -> type)
"start":pn = get_PE_pool() ;
         pd = rdy_ptr-> parameter ;
         bind[pn] = pd ;
         Q_global_run_list(rdy_ptr) ;
         transfer(pd) ;
         break ;
"fork": Q_global_run_list(rdy_ptr) ;
         pn = get_PE_pool();
         if(pn NOT NULL)
         {
           pd = rdy_ptr -> parameter ;
           bind[pn] = pd ;
           transfer(pd)   ;
         }
           transfer(pd_old) ;
         break ;
"term": transfer(pd_old)    ;
         if (global_run_list NOT NULL)
         {
            pn = rdy_ptr -> pn ;
            rdy_ptr = D_Q_global_run_list ;
            pd = rdy_ptr -> parameter  ;
            bind[rdy_ptr ->pn] = pd ;
            transfer(pd) ;
         }
         else
         {
            bind[rdy_ptr ->pn] = NULL ;
            ret_PE_pool(rdy_ptr ->pn) ;
         }
         break ;
"block":x = rdy_ptr ->parameter ;
         block(rdy_ptr,x) ;
         break ;
"unblock":x = rdy_ptr ->parameter ;
         rdy_ptr = random_select(x);
         pd = search(rdy_ptr ->pn) ;
         transfer(pd) ;
         transfer(pd_old) ;
         break ;
         }
         M_P_Simulator(rdy_ptr) ;
         rdy_ptr = get_next_ready_event ;
         }
   }
}
```

**Figure 4.** Skeleton of trace scheduler.

### 4.1 Example

Below a grid relaxation algorithm for solving numerically partial differential equations is described and executed for the hybrid simulation. This example is an illustration of the methodology for a simple algorithm on a complex machine (i.e. a cache-based system). A complete analysis for different cache and coherence mechanisms can be found in [DUB85].

Grid relaxation is an iterative technique to solve partial differential equations (PDEs) numerically. The traditional example chosen to compare algorithms and implementations for PDE solvers is Laplace's equation on a square domain of $R^2$, and is referred to as the *model problem* [YOU71]. The problem is first discretized on a square grid. In the Jacobi iteration, the computation consists in repeatitively computing the average of the four N-, E-, W-, and S-neighbors at each grid point. Therefore, the (K+1)th iterate is computed as

$$x_{i,j}^{K+1} = \frac{1}{4}\left[x_{i+1,j}^{K} + x_{i-1,j}^{K} + x_{i,j+1}^{K} + x_{i,j-1}^{K}\right]$$

Two copies of the grid, U and V, are maintained. In each iteration, a copy is updated by using the iterate values of the other copy. Then the two copies are interchanged. This is an example of a synchronized algorithm. This algorithm can also be implemented as an asynchronous algorithm in which the latest iterate values are used [KUN76]. Let $N^2$ be the size of the square grid. A partition can be described in terms of $P$ blocks, where $P$ is the number of processors. $P=p^2$, where $p$ is the number of blocks on each side. The boundary conditions add a total of $4N$ fixed iterate values to each grid, so that the total number of values for each grid is actually $N^2+4N$.

This algorithm is an example of a data-independent algorithm with homogenenous decomposition. A simple interleaving based on rank was applied for its simulation.

The trace was then used to study the performance of a bus-oriented multiprocessor system caches. For the *write-through* system, the memory is updated on each write, and no space is allocated in the cache on a write miss. Each write cycle on the bus is read by all the processor nodes and caches are invalidated when they contain a copy of the modified block[PAT82].

There are two types of events causing accesses to the bus: read misses and writes. Let $M$ be the total number of misses and $W$ the total number of writes in all the caches for one iteration. Note that $W=N^2$. If $C_{rb}$ and $C_w$ are the number of bus cycles required per cache block read and word write respectively, the total bus traffic due to global data accesses per Jacobi sweep is the sum of the miss traffic and the write traffic and can be written as

$$Traf\ (p,q)=MC_{rb}+N^2C_w \qquad (4.1)$$

This traffic places an upper bound on the execution speed because of the limited bus bandwidth.

Let $T_{rb}$ and $T_w$ be the processor wait time for a read miss and for a write in the absence of conflicts, respectively. Also, let $m_i$ be the total number of misses per iteration in processor $i$. A lower bound on the iteration time is obtained by neglecting all the bus conflicts.

$$T_{ite}\,(p,q)=t_0+\frac{N^2t_1}{P}+T_{rb}\ \underset{0\leq i<P}{Max}[m_i]+\frac{N^2}{P}T_w \qquad (4.2)$$

In (4.2), $t_1$ is the time to update one iterate when all operands are in cache, $t_0$ is a fixed time, independent of the decomposition. The maximum of "$m_i$" is used because the process with the worst case computation time determines the duration of the iteration in a synchronized algorithm[KUN76].

From the trace-driven simulator we can evaluate $M$ and $m_i$ for different values of $P$ and various cache configurations. The values obtained through simulation can then be substituted in (4.1) and (4.2).

## 5. Conclusion

Understanding the behavior and the performance trade-offs of multitasked algorithms in multiprocessor machines is a very difficult task. Many authors have tried to develop stochastic models. But such models are too general to be applicable to specific algorithms. Performance issues such as the impact of synchronization techniques, scheduling methods, and partitioning and allocation strategies cannot easily be mapped onto simple analytical models. This is specially true for data dependent algorithms, heterogeneous decompositions, or dynamic scheduling. Full-fledged cycle-by-cycle simulation of the multiprocessor is very tedious, especially if the goal of the analysis is to understand the performance of the multiprocessor for very large configurations of multiprocessors and very large problem sizes.

In this paper, we have presented an intermediate approach, namely a trace-driven, approximate simulation of the problem on the multiprocessor. The simulation yields results such as the total number of events of given types. These event counts are then introduced in throughput bounds or approximate analytical models to evaluate the approximation to the desired performance measure of the algorithm. The approach presented in this paper provides a general methodology, with which simulations and analytical models may be refined and validated progressively, from the complexity models to the cycle-by-cycle simulation.

The methodology has been applied to the grid relaxation and static and dynamic quicksort problems. These examples were presented here to illustrate the methodology. Extensive results can be found in the referenced technical reports.

## 6. References

[ABR84] S. Abraham, A. Gottlieb and C. Kruskal, "Simulating Shared-Memory Parallel Computers," *Ultracomputer Note #70,* Courant Institute, NYU, April 1984.

[ABU85] W. Abu-Sufah and A.Y. Kwok, "Performance Prediction Tools for Cedar: A Multiprocessor Supercomputer," *Proc. of 12th Ann. Intl. Symposium on Computer Architecture,* June 1985, pp. 406-413.

[AND83] G.R. Andrews, *et al.,* "Concepts and Notations for Concurrent Programming," *Computing Surveys,* Vol.15, No. 1, March 1983.

[ARC85] Special session on commercial cache-based multiprocessors, *Proceedings of the 12th International Symposium on Computer Architecture,* June 1985.

[AXE84] T.S. Axerold, P. Dubois and P. Elgroth, "A Simulator for MIMD Performance Prediction-- Application to the S-1 MkIIa Multiprocessor," *Parallel Computing,* North-Holland, Vol. 1, 1984, pp. 237-274.

[BAA78] S. Baase, *Computer Algorithms: Introduction to Design and Analysis,* Addison-Wesley, Reading, MA, 1978.

[BAL86] M. Balakrishnan and M. Dubois,"Performance Analysis of Quicksort in a Distributed Global Memory System," submitted to *Conpar'86.* Also available as a USC-CRI Technical Report.

[BRI83] F.A. Briggs and M. Dubois, "Effectiveness of Private Caches in Multiprocessor Systems with Parallel-Pipelined Memories," *IEEE Transactions on Computers,* Vol. C-32, No. 1, January 1983.

[CEZ83] R. Cezzar, and D. Klappholz, "Process Management Overhead in a Speed-up Oriented MIMD System," *Proceedings of the Parallel Processing Conference,* August 1983.

[COV85] R.G. Covington and J.R. Jump, *CSIM User's Guide,* Tech. Report No. TR8501, Dept. of Elec. and Comp. Eng., Rice University, Houston, Texas, Oct. 1985.

[DEM82] J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Transactions on Computers,* Vol. C-31, No. 4, April 1982.

[DUB82a] M. Dubois and F.A. Briggs, "An Approximate Analytical Model for Asynchronous Processes in Multiprocessor Systems," *Proceedings of the Parallel Processing Conference,* August 1982.

[DUB82b] M. Dubois and F.A. Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," *IEEE Transactions on Software Engineering,* July 1982. pp419-431.

[DUB82c] M. Dubois and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers,* Vol. C-31, No 11, November 1982.

[DUB85] M. Dubois, "Throughput Analysis of Cache-based Multiprocessor With Multiple Buses," *USC Technical Report* CRI-85-27, December 1985.

[GAJ85] D.D. Gajski, and J.K. Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer Magazine,* June 1985.

[GAN84] D.B. Gannon, and J. Van Rosendale, "On the Impact of Communication Complexity on the Design of parallel Numerical Algorithms," *IEEE Transactions on Computers,* Vol. C-33, No 12, Dec. 1984.

[GEH82] E.F. Gehringer, *et al.,* "The Cm* Testbed," *IEEE Computer,* October 1982.

[GOT83] A. Gottlieb, *et al.,* "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Transactions on Programming Languages and Systems,* Vol. 5, No. 2, April 1983.

[HEL78] D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra, *SIAM Review,* Vol. 20, No. 4, October 1978.

[HOO77] C.M. Hoogendorn, "A General Model for Memory Interference in Multiprocessors," *IEEE Transactions on Computers,* C-26, October 1977.

[JEF83] D. Jefferson, "Virtual Time," *Proceedings of the 1983 International Conference on Parallel Processing,* August 1983.

[JON80] A.K. Jones, *et al.,* "Experience Using Multiprocessor Systems -- A Status Report," *Computing Surveys,* Vol. 12, No. 2, June 1980.

[JUM85] J.R. Jump, J.B. Sinclair, F.A. Briggs, R.G. Covington, V. Balasundaram, R.V. Kothari and S. Madala, *Workload Characterization of CSIM-based Simulations,* Tech. Report Number TR8513, Rice University, Nov. 1985.

[KRU83] C.P. Kruskal, "Searching, Merging and Sorting in Parallel Computations," *IEEE Transactions on Computers,* Vol. C-32, No. 10, Oct. 1983, pp. 942-946.

[KUC84] D.J. Kuck, A. Sameh, R. Cytron, A. Veidenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, C. Beckman, J. Davis, C.P. Kruskal, "The Effects of Program Restructuring, algorithm change, and Architecture Choice on Program Performance," *Proceedings of the Parallel Processing Conference,* August 1984.

[KUM80] B. Kumar and E.S. Davidson, "Computer System Design Using a Hierarchical Approach to Performance Evaluation," *Communication of the ACM,* Vol. 23, No. 9, September 1980, pp. 511-521.

[KUN76] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results,* J.F. Traub Ed., New York: Academic Press, 1976.

[LAM78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communication of the ACM,* July 1978, Vol. 21, No. 7.

[LEE85] G. Lee, C.P. Kruskal and D.J. Kuck, "An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors," *IEEE Transactions on Computers,* Vol. C-34, No 10, October 1985.

[LIN81] B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering,* Vol. SE-7, No. 2, March 1981, pp. 174-188.

[MAD86] S. Madala, "Concurrent C User's Guide," *Dept. of Electrical and Computer Eng., Rice Univ.* Houston, Texas, 1986.

[MUD84] T.N. Mudge, and H.B. Al-Sadoun, "Memory Interference Models with Variable Connection Time," *IEEE Transactions on Computers,* Vol. C-33, No. 11, November 1984.

[PAP84] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of 11th Intl. Symposium on Computer Architecure,* 1984, pp348-354.

[PAT82] J.H. Patel, "Analysis of Multiprocessors with Private Cache Memories" *IEEE Trans. on Computers,* Vol. C-31, No. 4, April 1982, pp.296-304.

[PAT86] I. Patil and F.A. Briggs, "Dynamic Quicksort In Multiprocessors," *Tech. Report, TR8606,* Dept. of Elec. and Computer. Eng, Rice Univ., Houston,TX,Aug 1986.

[ROB79] J.T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms," *IEEE Transactions on Software Engineering,* Vol. SE-5, Jan. 1979.

[SCH78] H. Schwetman, "Hybrid Simulation Models of Computer Systems," *Communication of the ACM,* Vol. 21, No. 9, September 1978, pp. 718-723.

[SHA84] S.M. Shatz, "Communication Mechanisms for Programming Distributed Systems," *IEEE Computer,* June 1984.

[SMI81] B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real Time Signal Processing IV,* Vol. 298, Aug. 1981.

[SOK86] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, G. F. Pfister, "PSIMUL - A System for Parallel the Execution of Parallel Programs," *IBM T. J. Watson Research Center,* Yorktown Heights, New York.

[TED84] M. Tedd *et al. Ada for Multi-microprocessors,"* Cambridge University Press: Cambridge, England.

[VRS84] D. Vrsalovic, *et al.,* "Performance Prediction for Multiprocessor Systems," *Proceedings of the 1984 Parallel Processing Conference.*

[VRS85] D. Vrsalovic, E.F. Gehringer, Z.Z. Segall and D.P. Sieworek, "The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems," *Proc. of 12th Ann. Intl. Symposium on Computer Architecture,* June 1985, pp. 396-405.

[WEI84] P.J. Weinberger, "Cheap Dynamic Instruction Counting," *AT&T Bell Laboratories Technical Journal,* Vol. 63, No. 8, Part 2, Oct. 1984,pp. 1815-1826.

[YOU71] D. Young, *Iterative Solution of Large Linear Systems,* Academic Press: New York, 1971.

# DELTA TRANSFORMATIONS TO SIMPLIFY VLSI PROCESSOR ARRAYS FOR SERIAL DYNAMIC PROGRAMMING†

*Richard J. Lipton*
*Daniel Lopresti*

Department of Computer Science
Princeton University
Princeton, NJ 08544

## Abstract

We introduce a technique for transforming certain dynamic programming algorithms into ones which are in some sense equivalent, but require asymptotically less space and time under a logarithmic cost criterion. When mapped into silicon, these transformed algorithms result in VLSI processor arrays which are significantly smaller and faster. We illustrate the discussion with a general case of serial dynamic programming and briefly describe other, nonserial, applications in pattern matching.

## Introduction

VLSI processor arrays are often application-driven; the structure of the algorithm and the domain of the problem dictate the complexity of the hardware needed. Algorithmic structure has been used to simplify communication requirements. We show how structure in the problem can be exploited to simplify the computation itself.

Throughout the following complexity analysis we shall use the logarithmic cost criterion [1], an appropriate measure for algorithms to be implemented in VLSI. The cost of storing a non-negative integer $i$ under the logarithmic criterion is

$$l(i) = \begin{cases} 1 & if\, i = 0 \\ \lceil log_2(i) \rceil & if\, i > 0 \end{cases}$$

and the cost of evaluating a primitive function $f$ (e.g. addition or comparison) is the sum of the costs of its operands

$$l(f(x_1, x_2)) = l(x_1) + l(x_2)$$

### The Serial Dynamic Programming Problem

The general case of a problem solvable by serial dynamic programming can be expressed as

$$\min_X f(X) = \min_X \sum_{i \in T} f_i(X^i)$$

where

$$X = \{x_1, x_2, \dots, x_n\}$$

is a set of discrete-valued variables, $S_i$ is the definition set of $x_i$ ($|S_i| = \sigma_i$), and

$$T = \{1, 2, \dots, n-1\}$$

are the time instances of the problem. We wish to minimize the objective function $f$ over all schedules $X$ subject to the serializing constraint

$$X^i = \{x_i, x_{i+1}\}$$

The $f_i(X^i)$ are components of the objective function [2].

It may be helpful to observe that this formulation is fully equivalent to the problem of finding a shortest (i.e. lowest cost) path through a multistage graph with edge weights determined by

$$c_{i,j,k} = f_i(x_{i,j}, x_{i+1,k}) \quad 1 \le i \le n-1, 1 \le j \le \sigma_i, 1 \le k \le \sigma_{i+1}$$

where $x_{i,j}$ is the $j^{th}$ value $x_i$ assumes in some standard ordering. We interpret $c_{i,j,k}$ as the cost of traversing the edge from the $j^{th}$ vertex in the $i^{th}$ stage to the $k^{th}$ vertex in the $i+1^{st}$ stage.

### The Problem Restricted

For the development which follows, we make two assumptions which restrict this most general problem. The first, strictly for analytical convenience, is that

$$\sigma_i = 1 \quad i = 1, n \quad and \quad \sigma_i = m \quad 2 \le i \le n-1$$

so that we consider only graphs which have $n$ stages, with one vertex (or "node" or "state") in the first and last stages and $m$ vertices in each of the intermediate stages. Such a graph is shown in figure 1.

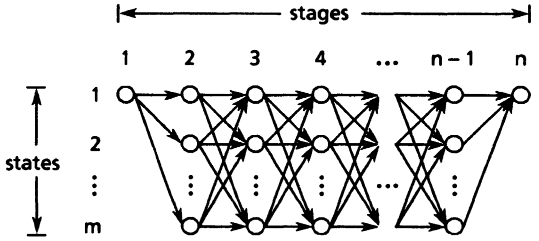


*figure 1.* a multistage graph

The second assumption is a necessary condition for the transformations we present. It is that

$$c_{i,j,k} \in N \quad and \quad 0 \le c_{i,j,k} \le C$$

for all $c_{i,j,k}$. That is, we assume the edge weights can be represented as elements chosen from a finite set of non-negative integers, and we know in advance their maximum value. This is a highly restrictive assumption, but one which reflects real-world conditions. Since programs and processor

† This research was supported in part by DARPA under contract N00014-82-K-0549.

0190-3918/86/0000/0917 $01.00 © 1986 IEEE

arrays have limited resources, we often know (or at least must require) bounds on the edge weights.

### A Dynamic Programming Solution

The shortest path through a multistage graph can be found using a backward recurrence with initial conditions

$$r_{n-1,j} = c_{n-1,j,1} \qquad 1 \le j \le \sigma_{n-1}$$

and iterative step

$$r_{i,j} = \min_{1 \le k \le \sigma_{i+1}} \left[ r_{i+1,k} + c_{i,j,k} \right] \qquad 1 \le i \le n-2,\, 1 \le j \le \sigma_i$$

where $r_{i,j}$ is the length of the shortest path from the $j^{\text{th}}$ vertex in the $i^{\text{th}}$ stage to the lone vertex in the $n^{\text{th}}$ stage. We shall refer to this recurrence as the *standard serial dynamic programming algorithm*.

### Space and Time Complexity

Since the edge weights of the graph lie between 0 and $C$, the cost of a path from a vertex in the $i^{\text{th}}$ stage to the vertex in the last stage can range from 0 to at most $(n-i)C$. We remark that after the optimal costs for the $i+1^{\text{st}}$ stage are used to determine those in the $i^{\text{th}}$ stage they can be forgotten. Thus, the storage required is approximately

$$ml((n-3)C) + ml((n-2)C)$$

bits, which means the space complexity is $O(m\log n)$.

For the time complexity, we note that an addition in the $i^{\text{th}}$ stage requires time $l((n-i-1)C) + l(C)$ and a comparison requires $2l((n-i)C)$. Since determining the cost of the optimal path for each of the $m$ nodes in the second through the $n-2^{\text{nd}}$ stages requires $m$ additions and $m-1$ minimizations, the intermediate stages contribute

$$\sum_{i=2}^{n-2} \left[ m^2[l(n-i-1)C + l(C)] + m(m-1)[2l((n-i)C)] \right]$$

which is $O(m^2 n\log n)$. Determining the optimal path from the vertex in the first stage requires $m$ additions and $m-1$ minimizations, and so contributes

$$m[l(n-2)C + l(C)] + (m-1)[2l((n-1)C)]$$

which is $O(m\log n)$. Hence the time complexity of this algorithm is $O(m^2 n\log n)$.

### Mapping the Recurrence into Silicon

Li and Wah [3] present a VLSI processor array for serial dynamic programming which is quite general, at the expense of failing to exploit all of the potential parallelism. The architecture itself is based on the observation that the inner loop step of the algorithm is equivalent to an inner-product in the closed semiring (N, *min*, +, ∞, 0). Hence, the problem can be solved as a sequence of matrix-vector multiplications, where the additive step is minimization and the multiplicative step is addition. The array is depicted in figure 2.

To determine the shortest paths for vertices in the $n-2^{\text{nd}}$ stage, the multiplicand vector is shifted through the array while the cost matrix for that stage is applied and the product vector accumulates. For the next stage, the product vector is shifted while the multiplicand vector remains stationary. This alternation continues; on the last pass the cost vector corresponding to the first stage is input and the final value of the shortest path is accumulated in the first processor.

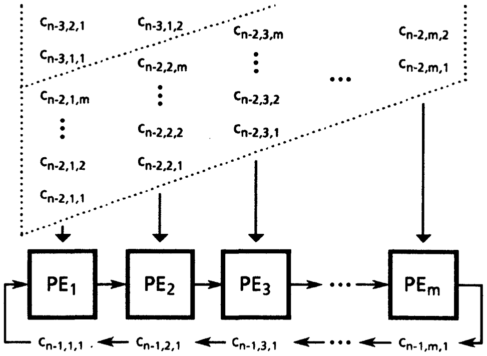


*figure 2.* VLSI array for serial dynamic programming

This parallel architecture requires $m$ processors. The first must be able to store values which can be as large as $(n-1)C$, the others as large as $(n-2)C$. Hence, each processor requires area $O(\log n)$, so the area of the entire array is $O(m\log n)$.

The time for the array to complete the computation is determined by that of the first processor, since it performs both the first and last operations. In processing the $i^{\text{th}}$ stage, it must perform $m$ additions which require time $l((n-i-1)C) + l(C)$ and $(m-1)$ comparisons which take $l((n-i)C)$. Thus, the time required is

$$\sum_{i=1}^{n-2} \left[ m[l((n-i-1)C) + l(C)] + (m-1)l((n-i)C) \right]$$

which yields a time complexity of $O(mn\log n)$.

### The Delta Transformation

We now proceed with the main result of this paper, the development of which we illustrate with the serial dynamic programming problem. The knowledge that the edge weights in the multistage graph are known to have a fixed range is powerful; it permits us to transform the standard algorithm into one which requires asymptotically less area and time under the logarithmic cost criterion.

### The Residue Reduction

We first observe that the bound on edge weights constrains the maximum difference between terms which appear in a minimization.

**Lemma 1:** the difference between any two terms which appear in a minimization in the standard serial dynamic programming algorithm is at most $2C$. That is,

$$\max_{i,j,k_1,k_2} \left[ (r_{i+1,k_1} + c_{i,j,k_1}) - (r_{i+1,k_2} + c_{i,j,k_2}) \right] \le 2C$$

(for proofs of this and following lemmas and theorems, see [4]).

This fact can be exploited to make determining the minimum simpler. In particular, it is only necessary to examine low order bits.

**Lemma 2:** given three non-negative integers $x_1$, $x_2$, and $\delta$ such that

$$x_1 - \delta \le x_2 \le x_1 + \delta$$

choose any $\Delta$ greater than or equal to $2\delta+1$ and let $y_1 = x_1 \mathrm{mod}\Delta$ and $y_2 = x_2 \mathrm{mod}\Delta$. If we define

$$min_\delta^\Delta[y_1, y_2] = \begin{cases} min[y_1, y_2] & if\, max[y_1, y_2] - min[y_1, y_2] \le \delta \\ max[y_1, y_2] & otherwise \end{cases}$$

then

$$min_\delta^\Delta[y_1, y_2] = min[x_1, x_2] mod\Delta$$

Now we replace the standard dynamic programming recurrence with the following

$$r_{n-1,j}^\Delta = c_{n-1,j,1} \qquad 1 \le j \le \sigma_{n-1}$$

and

$$r_{i,j}^\Delta = \underset{i,j,k}{min_{2C}^\Delta} \left[ r_{i+1,k}^\Delta +_\Delta c_{i,j,k} \right] \qquad \begin{matrix} 1 \le i \le n-2, 1 \le j \le \sigma_i, \\ 1 \le k \le \sigma_{i+1} \end{matrix}$$

(where $+_\Delta$ is addition modulo $\Delta$) for any $\Delta$ greater than or equal to $4C+1$. Whence we arrive at the main result of this section.

**Theorem 1:** for $r^\Delta{}_{i,j}$ defined above

$$r_{i,j}^\Delta = r_{i,j} mod\Delta \qquad 1 \le i \le n, 1 \le j \le \sigma_i$$

That is, the new recurrence performs precisely the same computation as the old, only using values modulo $\Delta$. We call $r^\Delta{}_{i,j}$ so defined a *residue reduction*; it is a distillation of the original algorithm.

### The Path Augmentation

The residue-reduced algorithm does not give us the true cost of the shortest path through the graph, although it does determine its remainder modulo $\Delta$ correctly. We observe, however, that the costs for the $n-1$st stage (the initial conditions) are determined precisely as before. Consider picking an $r^\Delta{}_{n-1,j}$ as an initial sample and inductively walking a path through the residue values to determine the true cost, $r_{1,1}$. To this end, we state

**Lemma 3:** the difference between the costs of the shortest paths from vertices in any two consecutive stages is at most $C$. More specifically,

$$r_{i,j} - C \le r_{i+1,k} \le r_{i,j} + C$$

Now we make a second general observation; it concerns determining the true value of a non-negative integer given only the remainder of that integer modulo some value and the full value of another integer which is known to be close in magnitude.

**Lemma 4:** given three non-negative integers $x_1$, $x_2$, and $\delta$ such that

$$x_1 - \delta \le x_2 \le x_1 + \delta$$

choose any $\Delta$ greater than or equal to $2\delta+1$ and let $y_1 = x_1 \mathrm{mod}\Delta$ and $\mu = \mathrm{min}^\Delta{}_\delta[y_1, x_2\mathrm{mod}\Delta]$. If we define

$$aug_\delta^\Delta[y_1, x_2] = \begin{cases} x_2 - (x_2 mod\Delta -_\Delta y_1) & if\, \mu = y_1 \\ x_2 + (y_1 -_\Delta x_2 mod\Delta) & if\, \mu = x_2 mod\Delta \end{cases}$$

then

$$aug_\delta^\Delta[y_1, x_2] = x_1$$

By Lemma 3 we can augment the reduced recurrence along any path from the last stage to the first. If we want to retain the general interconnection scheme of the original matrix-vector multiplication array, we should make the augmenting values available in an end processor. A logical choice is the path along the top of the recurrence, corresponding to $r_{1,j}$, as these values are always available in the first processing element.

Now, to the residue recurrence given earlier we add

$$a_{n-1} = r_{n-1,1}^\Delta$$

and

$$a_i = aug_{2C}^\Delta[r_{i,1}^\Delta, a_{i+1}] \qquad 1 \le i \le n-2$$

We then assert

**Theorem 2:** for the recurrence defined above and any $\Delta$ greater than or equal to $4C+1$

$$a_i = r_{i,1} \qquad 1 \le i \le n-1$$

That is, this new recurrence correctly determines the values $a_i = r_{i,1}$ and in particular the cost of the shortest path through the graph, $a_1 = r_{1,1}$. We call the combination of a residue reduction and a path augmentation a *delta transformation.*

### Space and Time Complexity of the Delta Algorithm

The residue-reduced recurrence $r^\Delta{}_{i,j}$ requires storage of $l(\Delta)$ bits for each value. As before, the space complexity is determined by that required to compute one stage of costs. There are $m$ vertices in a stage, so the space complexity is $O(m\log\Delta)$ which is $O(m)$ since $\Delta$ is a constant. For the augmentation path, we observe that a value at the $i^{\text{th}}$ stage can be as large as $(n-i)C$. Since only two augmentation values need be retained at any one time, the storage required is at most $l((n-1)C)+l((n-2)C)$ bits. Thus, the space complexity of the entire delta-transformed algorithm is $O(m+\log n)$. This is asymptotically less than the $O(m\log n)$ required by the standard dynamic programming algorithm.

The time required for a $\min^{\Delta_\delta}$ at any stage of the computation is $O(\log(\Delta))$, which is to say constant. Hence, the complexity contributed by the reduced recurrence is $O(m^2n)$. The time required for an $\text{aug}^{\Delta_\delta}$ at the $i^{\text{th}}$ stage is proportional to $l((n-i)C)$, hence the augmentation path contributes $O(n\log n)$ to the time complexity. Thus, the complexity for the entire algorithm is $O(m^2n+n\log n)$, which is asymptotically faster than the $O(m^2n\log n)$ required by the original.

### Mapping the Delta Algorithm into Silicon

We can again use Li and Wah's matrix multiplication scheme, with a slight addition. Recall that the value $r_{i,1}$ is available at $\text{PE}_1$ after the $i^{\text{th}}$ stage has been processed; on odd numbered iterations it is the stationary value, on even numbered iterations it is the first value to shift in. Hence, we add a processor external to the array on the left side, as depicted in figure 3. This processor is initialized with the value $c_{n-1,1,1}$. Then, every $m$ clock cycles, it alternates between sampling $\text{PE}_1$'s stored value and the first value of the next pass to shift in.

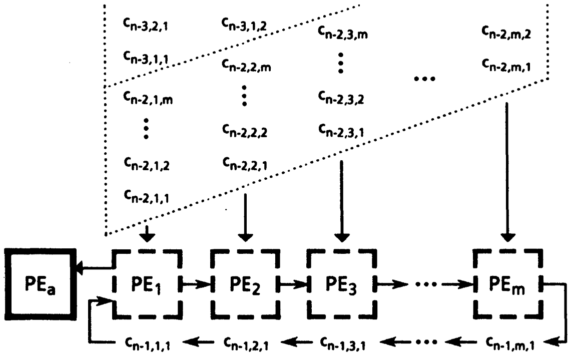


*figure 3.* VLSI array for the delta-transformed algorithm

This scheme requires a total of $m+1$ processors. Each of the $m$ computing the residue recurrence store values which can be at most $\Delta$; hence each requires area $O(\log\Delta)$, which is a constant, so their combined area is $O(m)$. The augmenting processor must manipulate values as large as $(n-1)C$, hence it requires area $O(\log n)$. Thus, the area of the entire array is $O(m+\log n)$ as opposed to the $O(m\log n)$ that the original required.

The reduced processors perform inner product steps on data values of constant size, which require constant time. Since each processor handles $O(mn)$ of these steps, their time complexity is $O(mn)$. In the same time that these processors compute an entire stage, the augmenting processor must perform operations which require time at most $O(\log n)$. Since there are $O(n)$ stages, the augmenting processor requires a total time $O(n\log n)$. Hence the time complexity of the entire array is $O(n\max[m, \log n])$ as opposed to the $O(mn\log n)$ the original array required.

### Conclusions

We have demonstrated a technique to simplify VLSI processor arrays for a general case of serial dynamic programming. Extensions to other dynamic programming algorithms, both serial and nonserial, are straightforward; the necessary and sufficient condition for application of a delta transformation is that all values to be minimized (or maximized) are close in magnitude. Unfortunately, the transformations as presented here will not yield space/time savings for every problem [4]. They can be applied, however, in many important instances, including approximate string matching, two-dimensional image comparison, and dynamic time warping of speech. For example, we have designed and fabricated a linear systolic array for comparing DNA sequences (essentially strings over a small alphabet) [5]. Using a delta transformation, we were able to place 30 processors on a single nMOS chip which runs at six megahertz, whereas if we had implemented the unmodified algorithm, we would have been fortunate to fit four processors running at a much lower speed.

This paper is a condensed version of [4].

### References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1976.

[2] Umberto Bertelè and Francesco Brioschi, *Nonserial Dynamic Programming*, New York: Academic Press, 1972.

[3] Guo-jie Li and Benjamin W. Wah, "Systolic Processing for Dynamic Programming Problems," *Proceedings of the 1985 Conference on Parallel Processing*, 1985.

[4] Richard J. Lipton and Daniel Lopresti, "Using Residue Arithmetic to Simplify VLSI Processor Arrays for Dynamic Programming," TR CS-022, Department of Computer Science, Princeton University, January, 1986.

[5] Richard J. Lipton and Daniel Lopresti, "A Systolic Array for Rapid String Comparison," *1985 Chapel Hill Conference on Very Large Scale Integration*, Henry Fuchs, editor, Rockville, MD: Computer Science Press, 1985, pp. 363-376.

# Parallel External Merging

by

Eliezer Dekel and Istvan Ozsvath

University of Texas at Dallas

## Abstract

The problem of merging two large ordered sets A, and B is considered. An efficient parallel algorithm is developed. The shared memory model of computation is used.

Key words: Parallel algorithm, SIMD, parallel merge, external sorting.

## 1. Introduction

The merging and sorting problems have received considerable attention in the literature [1] - [5], [7] - [18]. While the merge operation serves as a basic operation in many algorithms [10] it is usually treated in the context of sorting. With the growing use of databases, the problem of merging becomes more important. Much effort has been devoted to solving this problem on parallel processors [9], [12], [13], [17]. In this paper we present an optimal parallel algorithm for merging two sets of records.

Classically the merge becomes external when the sets are too large to be contained in the memory of the processor. In parallel models input is considered to be external if it cannot fit in the internal memory of the PEs. For this particular problem, the two sets can be stored on a disk. Disks with multiple read/write heads can be regarded as a EREW P-RAMs. We associate with each read/write head a PE and consider the disk medium as the shared memory. The PEs can communicate with each other in 0(1) time by leaving messages in predetermined memory locations.

## 2. Parallel External Merge Algorithm.

Two sorted files A and B with m and n records respectively are to be merged. The result of the merge is a third sorted file C, with n+m records. The serial merge of the two input files, (given correct buffering and overlapped I/O and computation) can be performed in O(m + n) time. In this paper we present a solution to the external merge problem when a fixed number, p, of PEs are available.

Ideally with p PE's, where p < m+n, the files can be merged in O((n+m)/p) time. Our algorithm performs the merge within this optimal time. Conceptually the algorithm has three stages. In the first stage we identify pairs of subfiles from files A and B that could be independently merged. In the second stage, we merge all these subfiles in parallel. Each of these pairs is merged serially by one of the p available PE's. In the last stage, the results of the individual merges are concatenated to produce the result file C.

Let the key of record q in file X be $K_q^X$. Record i in file A has insertion point j in file B if $K_j^B \le K_i^A \le K_{j+1}^B$. If $K_j^A \le K_1^B$ (the key of the first record in B) then it has insertion point 0. If $K_n^B \le K_i^A$ then record i has insertion point n+1 (n is the index of the last record in B).

We first partition each of our input files into p subfiles: subfile $A_i$ with at most $\lceil m/p \rceil$, $1 \le i \le p$ and subfile $B_j$ with at most $\lceil n/p \rceil$, $1 \le j \le p$ records. Define the partition records to be the records with indexes: 1, $\lceil m/p \rceil$, $2*\lceil m/p \rceil$, ... ,$(p-1)*\lceil m/p \rceil$, m in A and, $\lceil n/p \rceil$, $2*\lceil n/p \rceil$, ... , $(p-1)*\lceil n/p \rceil$, n in file B. Next for each of the 2p+2 partition records, we read into memory its key and mark whether it came from file A or file B. Observe that the p+1 keys from each file are sorted. We now merge the two sets of keys, using a bitonic merge [11] to form a sorted sequence. Let the sorted sequence be $S = \{K_h^X, 1 \le h \le 2p+2\}$, where X can be either A or B. Let $J_h$, $1 \le h \le 2p+2$ be the original index of $K_h^X$ in file X. Having created this sequence, it is easy to see that there is no need to search the whole file in order to find an insertion point for some record i. Let $K_h^A$ be a key for partition record i in A, i. e. $J_h = i$. Let $K_f^B$ be the last key from file B in S such that $K_f^B \leq K_h^A$. Also let $K_r^B$ be the first key from file B in S such that $K_h^A \leq K_r^B$. Clearly, when looking for the insertion point of record i of A in file B we can limit our search in file B to records between the records indexes $J_f$ to $J_r$.

By performing the bitonic merge on the keys of the partition records we reduced both the number of disk accesses and also the number of read conflicts. When looking for an insertion point for record i (j) of A (B) in B (A) we have to search a subfile with at most $\lceil n/p \rceil$ ($\lceil m/p \rceil$) records. Since the search range is clearly partitioned, no record has to be accessed more then once. Clearly the number of disk accesses is reduced. Although we did not eliminate read conflicts, we definitely reduced the probability of one occurring.

Although we showed how to find insertion points in file B corresponding to the records in A, the same method can be used to find insertion points in file A corresponding to the records in file B. To identify the partition records in A that are candidates for creating read conflicts, we mark all instances in S where there are j > 1 keys from file A in between two given keys from B. Let us consider one instance of j > 1 keys from A falling in between two consecutive keys of partition records in B. More formally we look at the following subsequence of S: $K_h^B$, $K_{h+1}^A, \ldots, K_{h+j}^A$, $K_{h+j+1}^B$. As explained earlier, this translates to a search for insertion points in B for records $J_{h+1}$, $J_{h+2}$, ..., $J_{h+j}$ of file A. The search for the j points has to be conducted on records $J_h$ to $J_{h+j+1}$ in file B. Since in the search step we assign one PE for each partition record in A, we can expect j PEs to be assigned to search in the interval $J_h$ to $J_{h+j+1}$ in B. These j PEs can partition the interval to j subfiles in B. Each of the new subfiles can have at most $\lceil \lceil n/p \rceil / j \rceil$ records. The new j partition points from B can next be merged with the j keys from A. Now, there are two possible cases. Either there is one j point from A between two consecutive points from B, or there are several. In the first case, the PE can conduct the search independently; namely, there is no danger of a read conflict. In the latter case the process can be repeated until a situation with one PE searching a partition is arrived at (i. e. no read conflicts occur). In the worst case the search without read conflict problem will be trivially solved when a partition with only one record is created.

The introduction of the first level of bitonic merge has a cost of O(log p) when using p PEs. The search step, still with read conflicts, can be performed in O(log (n/p)) time on file B. On file A the search step takes O(log (m/p)). To eliminate read conflict we continue to partition intervals and perform bitonic merges until we can ensure searches without read conflicts. This process can be done in $O((\log p)*(\log_p (n/p)))$ for file B and in $O((\log p)*(\log_p (m/p)))$for file A. The complexity of the partition stage of our new algorithm is readily seen to be O(log m + log n) as before.

In order to simplify the discussion above we assumed that a PE "knows" whether it is the only one to be searching the interval. To perform the Bitonic merge effectively each PE should also have the boundaries of the search interval and the number of PEs sharing that interval. This information can be obtained in O(log p) time. A detailed discussion of this step is given in Dekel and Ozsvath [7].

## 3. Conclusions.

In this paper we describe a parallel merge algorithm. We show how to eliminate the possibility of read conflicts without changing the complexity of the algorithm. We chose to develop an algorithm for an important operation, file merging. The need for algorithms like our merge algorithm will increase dramatically as more parallel computers are in use. Indeed, such computers are likely to become quite common in the near future. These computers will come with a fixed number of processors and will be expected to perform operations on varied sizes of data.

## References

[1] G. Baudet and D. Stevenson,"Optimal sorting for parallel computers," IEEE Trans. Comput. , Vol c-27, No 1,(Jan 1978) pp. 84-87.

[2] M. A. Banuccelli, E. Lodi and L. Pagli,"External sorting in VLSI," IEEE Trans. Comput. , Vol c-33, No 10,(Oct 1984) pp. 931-934.

[3] D. Bitton, D. J. DeWitt, D. K. Hsaio and Jaishankar Menon, "A taxonomy of parallel sorting," Computing Surveys , Vol 16, Nu. 3 (Sept 1984) pp. 265-318.

[4] K-M Chung, F. Luccio and C. K. Wong,"On the complexity of sorting in magnetic bubble memory systems," IEEE Trans. Comput. , Vol c-29, No 7, (Jul 1980), pp. 553-563.

[5] E. Dekel, "Parallel sort with few processors," University of Texas at Dallas, Technical Report CS-204.

[6] E. Dekel and S. Sahni, "Binary trees and parallel scheduling algorithms," , IEEE Trans. Comput. , Vol. C-32, No. 3 (March 1983) pp 307-315.

[7] E. Dekel and I. Ozsvath, "Parallel external merging," University of Texas at Dallas, Technical Report CS-207.

[8] S. Even, "Parallelism in tape sorting," Comm. ACM , 17,4 (Apr 1974), pp. 202-204.

[9] F. Gavril,"Merging with parallel processors," Comm. ACM , 18,10 (Oct 1975), pp. 588-591.

[10] Horowitz an Sahni, "Fundamentals of data structures," Computer Science Press, Rockville MD, 1976.

[11] D. E. Knuth, The Art of Computer Programming, Vol 3, Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[12] C. P. Kruskal, "Searching merging and sorting in parallel computation," IEEE Trans. Comput. , Vol c-32, No 10,(Oct 1983) pp. 942-946.

[13] D. T. Lee, H. Chang and C. K. Wang, "An on-chip compare /steer bubble sorter," IEEE Trans. Comput. , Vol c-30, No 6, (Jun 1981) pp. 396-405.

[14] D. Nath, S. N. Maheshwari and P. C. P. Bhatt,"Efficient VLSI networks for parallel processing based on orthogonal trees," IEEE Trans. Comput. , Vol c-32, No 6,(Jun 1983) pp. 569-581.

[15] C. D. Thompson,"The VLSI complexity of sorting," IEEE Trans. Comput. , Vol c-32, No 12,(Dec 1983) pp. 1171-1184.

[16] S. Todd,"Algorithms and hardware for a merge sort using multiple processors," IBM J. Res. Develop. , Vol 22, No 5, (Sep 1978) pp. 509-517.

[17] L. G. Valiant,"Parallelism in comparison problems," SIAM J Comput. , 4 (3) (1977) pp. 348-355.

[18] H. Yasuura, N. Takagi and S. Yajima, "The parallel enumeration sorting scheme for VLSI," IEEE Trans. Comput. , Vol c-31, No 12, (Dec 1982) pp. 1192-1201.

# Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem

*Robert A. Wagner*
*Yijie Han*

Department of Computer Science
Duke University
Durham, NC 27706

## Abstract

The data dependent prefix problem is to compute all the $n$ initial products $x_1 \bigcirc x_2 \bigcirc \cdots \bigcirc x_k$, $1 \le k \le n$, where the order is specified by a linked list. A parallel algorithm for the data dependent prefix problem is presented. This algorithm has time complexity $O(\frac{n}{p}+\log n \log\frac{n}{p})$ using $p$ processors on the exclusive-read exclusive-write computation model. A bucket sorting algorithm is also developed to be used as a component of the prefix algorithm. This bucket sorting algorithm sorts $n$ numbers in the range $\{1,2,...,m\}$ using $p$ processors in $O(\lceil \frac{\log m}{\log(\frac{n}{p}+\log p)} \rceil (\frac{n}{p}+\log p))$ time.

Index words -- Prefix, bucket sorting, parallel algorithms, graph algorithms, optimal algorithms.

## 1. Introduction

Parallel algorithms have been studied in many different areas. Some serial algorithms are easily parallelized while some other serial algorithms seem difficult to parallelize. The study of parallel algorithms reveals many properties of the problems which were previously unknown. In the serial case, time complexity is usually the most important complexity measure for algorithms. In the parallel case, the processor complexity also comes into play. To measure how much parallelism of the problem has been exploited by an algorithm the parameter of *speedup* is defined as $\frac{T_1}{T_p}$, where $T_i$ is the time complexity of the algorithm using $i$ processors and $T_1$ is the time complexity of the fastest known sequential algorithm for the same problem. It is not difficult to see $\frac{T_1}{T_p} \le p$. When $T_p = \Theta(\frac{T_1}{p})$ we say the parallel algorithm achieves linear speedup with $p$ processors. In the parallel environment algorithms with maximum speedup and algorithms with optimal performance (linear speedup) are both sought. As the number of processors grows the utilization of the processors tends to decrease and linear speedup algorithms become difficult to obtain. The contribution of the research presented in this paper is a new parallel algorithm for the data dependent prefix computation. This algorithm achieves linear speedup allowing asymptotically more processors than previously known algorithms. The time complexity of our algorithm improves the best previous result. A bucket sorting algorithm used as a component of the prefix algorithm is also presented. This sorting algorithm shows better performance than known algorithms.

The shared memory computation model is assumed in this paper. In a shared memory model both memory cells and processors are linearly ordered and each memory cell can be addressed by any processor. Three shared memory models are distinguished according to the ways they resolve read or write conflicts[10]. The Concurrent-Read Concurrent-Write model(CRCW), which is the strongest model among these three, allows simultaneous read and write of one memory cell by several processors. The content resulting after a cell is written by more than one processor can be determined in a variety of ways. In the Concurrent-Read Exclusive-Write(CREW) model several processors can simultaneously read one memory cell, but simultaneous write is prohibited. The weakest model among three, the Exclusive-Read Exclusive-Write(EREW) model, prohibits both simultaneous read and write. Our algorithms are EREW and CREW algorithms.

Let $\bigcirc$ denote an associative operation. Given $x_1, x_2, \cdots, x_n$, the (data independent) prefix problem is to compute $x_1 \bigcirc x_2 \bigcirc \cdots \bigcirc x_k$, $1 \le k \le n$. The data dependent prefix problem is to compute $x_1 \bigcirc x_2 \bigcirc \cdots \bigcirc x_k$, $1 \le k \le n$, for a given linked list containing $x_1, x_2, ..., x_n$, with $x_i$ following $x_{i-1}$. For both cases, the product computation problem is to compute $x_1 \bigcirc x_2 \bigcirc \cdots \bigcirc x_n$.

The prefix problem is a fundamental problem in parallel computation. It finds applications in many problems such as processor allocation and reallocation, data alignment, data compaction, sorting and polynomial evaluation. The data dependent prefix problem is especially important since linked lists have been used widely as a data structure in many algorithms and prefix computation is probably the most basic and fundamental computation for linked list manipulation. One example is to use the prefix algorithm to number the data items in a linked list so that each item knows how far it is from the first element of the linked list (compute list rank for each element). This numbering operation is extremely useful when one intends to apply certain parallel operations to the linked list.

Studies have been conducted on the parallel prefix problem[2][5][7][8][9]. These studies assume that the input data is data independent in the sense $x_i$ is in memory cell $i$. For the data dependent case, where input data forms a

* Work reported herein was partially supported by the National Science Foundation under grant MCS - 8202610.

0190-3918/86/0000/0924 $01.00 © 1986 IEEE

linked list and a map specifies which element follows which, optimal algorithms are also pursued. The best previous result is due to Kruskal, Rudolph and Snir[6] which achieves time complexity $O(\frac{\log n}{\log(2n/p)}\frac{n}{p})$* using $p$ processors on the EREW model. In this paper we show that time complexity $O(\frac{n}{p}+\log n \log\frac{n}{p})$** can be obtained on the EREW model. Our result achieves linear speedup whenever $p=O(\frac{n}{\log n \log\log n})$. This shows improvement over the best previous result[6] which achieves linear speedup only when $p<n^{\epsilon}$, $0\leq\epsilon<1$. When our algorithm uses the maximum number of processors for which it can achieve linear speedup its time complexity becomes $O(\log n \log\log n)$, a polynomial in $\log n$. In contrast the best previous algorithm[6] can achieve a time complexity of $O(n^{1-\epsilon})$ using the maximum number of processors for which the algorithm obtains linear speedup. Our algorithm is the first result which shows that linear speedup can be achieved for the data dependent prefix problem with polylogarithmic time complexity.

Parallel sorting problem has been studied extensively. See [1] for a survey of this area. For our purpose, a sorting algorithm is required for sorting $n$ numbers in the range $\{1,2,...,\log n\}$ with $p$ processors in time $O(\frac{n}{p}+\log p)$. Parallel sorting algorithms mentioned in [1] do not meet our requirement. Parallel comparison sorting algorithms are not useful here because they will take at least $\Omega(\frac{n\log n}{p})$ time with $p$ processors. Hirschberg's bucket sorting algorithm [1][4] also implies a time bound of $O(\frac{n\log n}{p})$. Kruskal, Rudolph and Snir claimed a sorting algorithm which sorts $n$ numbers in the range $\{1,2,...,n\}$. The time complexity of their algorithm is $O(\frac{\log n}{\log(2n/p)}(\frac{n}{p}+\log p))$ with $p$ processors. Application of their sorting algorithm to our problem would imply a time bound of $O(\frac{\log n}{\log(2n/p)}(\frac{n}{p}+\log p))$ which is also too slow. We have developed our own bucket sorting algorithm. Our algorithm sorts $n$ numbers in the range $\{1,2,...,m\}$ with $p$ processors. The time complexity of our sorting algorithm is $O(\lceil\frac{\log m}{\log(\frac{n}{p}+\log p)}\rceil(\frac{n}{p}+\log p))$. Compared to the results of Hirschberg and Kruskal et. al., our result is more flexible in the sense that it adjusts itself better to the range of the numbers and the number of processors used.

## 2. Basic Fact and Definition

Product circuits, which are directed acyclic graphs, are also widely used as a parallel computation model. Some results may be obtained relatively easily using this model. We allow indegree of any node in the graph to be no more than 2. A node with indegree 2 represents a product of its two inputs. The nodes with indegree 0 are the input nodes. The depth of a circuit is the longest directed path in the graph. Fig. 1 shows two circuits for product computation. For the prefix computation the following fact is important to us.

**Fact:**

> Any circuit for product computation with the property that every node of the circuit represents a product of $\bigcirc_{k=i}^{j} x_k, 1\leq i\leq j\leq n$, can be used to construct a circuit for the prefix problem, and the depth of the prefix circuit is within a constant factor of the depth of the original product circuit.

For the product computation, the circuits can be viewed as trees, which we will call product trees, with sons pointing to fathers. For the class of product trees satisfying the condition stated in the above fact we can define, for each non-input node, its left son as the node which feeds $\bigcirc_{k=i}^{m} x_k$ to it and right son which feeds $\bigcirc_{k=m+1}^{j} x_k$ to it.

The fact comes from the following observation. To compute the prefix, each node representing $\bigcirc_{k=1}^{j} x_k, 1<j\leq n$, passes its product to its father. Each of these fathers then passes the product received to its right son. Each interior node getting a product from its father will transfer it to its left son and then compute a new product by combining the product passed from its father with the product accumulated by its left son during the process of computing the product. This new product will be passed to the interior node's right son. Fig. 2 shows this process.

This fact essentially shows that, for the serial and parallel (data independent) case, the prefix problem is no harder than the product problem. This fact is noted by Schwartz[9] and many other researchers and is used to construct parallel (data independent) prefix algorithms. This fact also applies to the data dependent prefix problem as it was used by Kruskal et. al. to construct their algorithm. Our algorithm also uses this fact, i.e. we are going to construct a parallel product algorithm with the product tree which the algorithm implicitly builds satisfying the condition stated in the fact. Thus our algorithm can be easily transformed to compute the prefix. This is accomplished by, when the product computation finishes, reversing the computation and popping out the product stored during the process of product computation.

Now we give definitions for some concepts related to linked list.

**Definition:**

> A linked list with $n$ elements is a data structure whose $n$ elements are associated with $n$ consecutive memory locations, where these $n$ memory locations are linked one after another by arcs. The last arc in the list points to a special symbol *nil* denoting the end of the list. An arc $(a,b)$ indicates that the element associated with location $a$ is followed by the element associated with location $b$. This arc is represented by value $b$ in location $a$. $a$ is the tail of the arc and $b$ is the head of the arc.

* Logarithms in this paper are to the base 2.

** We note here that $O(\frac{n}{p}+\log p)=O(\frac{n}{p}+\log n)$ and $O(\frac{n}{p}+\log p\log\frac{n}{p})=O(\frac{n}{p}+\log n\log\frac{n}{p})$, for $p\leq n$.

## 3. Previous Results for the Prefix Problem

It is advantageous here to mention some of the previous work on the prefix problem, since our algorithm uses some of the previous known algorithms as components and our work represents a continuation of the effort of these researchers.

The following well-known algorithm can be used to solve the prefix problem.

```
PREFIX_INDEPENDENT ($X[0..n-1]$);
   begin
      for $k:=0$ to $\lceil \log n \rceil -1$ do
         forall $i: 2^k \leq i < n$ do
            $X[i]:=X[i-2^k] \bigcirc X[i]$;
   end.
```

Algorithm PREFIX_INDEPENDENT has a corresponding version for the data dependent case, which is expressed by Wyllie[11]:

```
PREFIX_DEPENDENT($X[0..n-1]$,$NEXTX[0..n-1]$,$HEAD$);
  forall $i: 0 \leq i < n$ do
     begin
       $temp:=HEAD$;
       if $NEXTX[i] \neq nil$ then $NEXTX[NEXTX[i]]:=i$
       else $HEAD:=i$;
       $NEXTX[temp]:=nil$;
       for $k:=1$ to $\lceil \log n \rceil$ do
          if $NEXTX[i] \neq nil$ then
             begin
                $X[i]:=X[i] \bigcirc X[NEXTX[i]]$;
                $NEXTX[i]:=NEXTX[NEXTX[i]]$;
             end
     end.
```

If $n$ processors are available both algorithms will take $O(\log n)$ time. However, in no situation will they achieve linear speedup. This occurs because the total number of operations $N$ involved in these algorithms is $O(n \log n)$. With $p$ processors the best one can hope for is $O(\frac{n \log n}{p})$.

Kruskal et. al. [6] discussed an improved version of PREFIX_INDEPENDENT which achieves linear speedup. The elements are partitioned into $p$ blocks where block $i$ is $X[i \cdot \frac{n}{p} .. (i+1)\cdot\frac{n}{p}-1]$, $0 \leq i < p$. Processor $P_i$ is assigned to block $i$ to calculate the prefix (and therefore the product $p_i$) of block $i$. Then PREFIX_INDEPENDENT can be used to obtain the prefix $p_i^*$ for the $p$ products. Finally $p_i^*$ is added to each element of block $i+1$ by processor $P_i$.

However, no technique similar to the one mentioned above is known for algorithm PREFIX_DEPENDENT. A handy improvement can be made to achieve $O(\frac{n \log p}{p})$ time with $p$ processors. Fig. 3 shows that a linked list can be transformed into a linked tree of $p$ branches, each with length no more than $\frac{n}{p}+1$. This can be done in $O(\frac{n}{p}\log p)$ time with $p$ processors by repeatedly assigning $NEXTX[i]:=NEXTX[NEXTX[i]]$. This is equivalent to executing PREFIX_DEPENDENT for $\lceil \log p \rceil$ loops. Thereafter each processor works on one branch. The timing will be $O(\frac{n \log p}{p})$.

Kruskal, Rudolph and Snir's result shows that with $p$ processors $O(\frac{\log n}{\log(2n/p)}\frac{n}{p})$ time can be achieved for the data dependent prefix computation[6]. The hard part of the problem is to process arcs, in an order that guarantees that the arcs being simultaneously processed are not chained together. By dividing the $n$ elements into $\frac{n}{p}$ blocks, they apply $p$ processors to visit the $\frac{n}{p}$ blocks one at a time. During this phase, only arcs leaving a block are processed. Since no processor will look at the head of the arcs leaving the block which the processors are visiting, it is ensured that arcs simultaneously processed are not chained together. After this all the inter-block arcs are treated so the whole problem is divided into $\frac{n}{p}$ equal size subproblems. By distributing processors evenly to all the subproblems and applying recursion all the arcs will be processed. This is one pass of the so-called Pair-Off. After pair-off each remaining element must have both of its neighbors eliminated, thus the number of remaining elements is at most $\frac{2n}{3}$. Elements are compacted and another pass of pair-off is applied. The number of elements left after each pass will form a geometric series and consequently, time complexity of $O(\frac{\log n}{\log(2n/p)}\frac{n}{p})$ is obtained.

## 4. A Parallel Bucket Sorting Algorithm

In this section a parallel bucket sorting algorithm is presented. This algorithm sorts $n$ numbers in the range $\{1,2,...,m\}$ using $p$ processors. The sorting requires time $O(\lceil \frac{\log m}{\log(\frac{n}{p}+\log p)} \rceil (\frac{n}{p}+\log p))$.

To sort $n$ elements in the range $\{1,2,...,m\}$ with $p$ processors, we divide the elements into $\lceil \frac{n}{p} \rceil$ blocks. These blocks will be visited by the $p$ processors sequentially. Suppose element $i$ valued $v_i$ is visited by processor $P_k$, the element will be dropped into the $((v_i-1)\cdot p+k)$-th bucket. After all the elements are dropped into buckets, packing the elements will yield the sorted sequence. The details of the algorithm are shown below.

```
BUCKETSORT($A[1..n]$) /* $1 \leq A[i] \leq m$, $1 \leq i \leq n$. */

  processors: $P_i$, $1 \leq i \leq p$;
  array:     $S[1..m \cdot p]$,  /* Buckets */
             $C[1..n]$,        /* Counter for each element to
                                calculate its rank.
                              */
             $G[1..m]$;        /* $G[i]$ is the total number
                                of occurrences of elements
                                valued $i$.
                              */
```

```
forall $P_i : 1 \leq i \leq p$
  begin
  /* Initialization. Takes $O(m)$ time.*/
    $S[1..m \cdot p] := 0$;

  /* Throwing elements into buckets.
     Takes $O(\frac{n}{p})$ time.
  */
    for $k := 1$ to $\lceil \frac{n}{p} \rceil$ step 1
      begin
        $C[(k-1) \cdot p + i] := S[(A[(k-1) \cdot p + i] - 1) \cdot p + i]$;
        $S[(A[(k-1) \cdot p + i] - 1) \cdot p + i] :=$
           $S[(A[(k-1) \cdot p + i] - 1) \cdot p + i] + 1$;
      end

  /* $G[i]$ is going to be returned to the calling
     procedure. Takes $O(m + \log p)$ time.
  */
    forall $k : 1 \leq k \leq m$ do $G[k] := \sum_{i=(k-1) \cdot p+1}^{k \cdot p} S[i]$;

  /* Packing. Takes $O(\frac{n}{p} + m + \log p)$ time. */
    PREFIX_INDEPENDENT($S[1..m \cdot p]$);
    for $k := 1$ to $\lceil \frac{n}{p} \rceil$ step 1
      begin
        $C[(k-1) \cdot p + i] := C[(k-1) \cdot p + i]$
           $+ S[(A[(k-1) \cdot p + i] - 1) \cdot p + i - 1]$;
      end
 end
```

For $n$ elements ranging from 1 to $m$, BUCKETSORT takes $O(\frac{n}{p} + m + \log p)$ time units with $p$ processors. When $m > \frac{n}{p} + \log p$, we can use the idea of radix sorting. The $\log m$ bits representing the numbers are divided into $\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil$ blocks. Starting from the least significant block of bits, BUCKETSORT can be applied. After applying BUCKETSORT $\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil$ times, once for each block of bits, the sorting is accomplished. The total time will be $O(\lceil \frac{\log m}{\log(\frac{n}{p} + \log p)} \rceil (\frac{n}{p} + \log p))$ since the $m$ in the BUCKETSORT is $\frac{n}{p} + \log p$ now.

We note here that this sorting algorithm can also be used to pack data. Assigning 1 to each marked element and 2 to each unmarked element, execution of this sorting algorithm will pack the marked elements to the beginning of the array and unmarked elements to the end of the array.

## 5. The Main Algorithm

Our algorithm for the data dependent prefix problem is presented in this section. This algorithm uses the observation stated before, i.e. an appropriate product computation algorithm can be used to construct an algorithm for prefix computation. Thus we are focusing on constructing a suitable algorithm for product computation. The idea of Kruskal et. al. that appropriate pairs of elements are combined such that one pass of combining (or 'pair-off') will reduce the number of elements by at least one third is followed here. However, instead of using $p$ processors to cut $n$ elements into $\frac{n}{p}$ blocks we observed that the $n$ arcs in the linked list can be divided into $O(\log n)$ groups and all the arcs in one group can be eliminated simultaneously by one 'pair-off' pass. The details of this idea are expressed below.

We divide arcs into groups by the following rule: arc $(a, b)$ is in group $2k - a_k$ where

$k = \min \{ i \mid \text{the } i\text{-th bit of } a \ XOR \ b \text{ is } 1 \}$,

*XOR* is the exclusive-or operation, bits are counted from the least significant bit starting at 1, $a_k$ is the $k$-th bit of $a$. We prove the following lemma.

**Lemma:**

a) Each arc belongs to one and only one group.

b) For any group $G$, if two different arcs $(a, b) \epsilon G$ and $(c, d) \epsilon G$, then $a, b, c, d$ are all distinct.

**Proof:**

Part a) is obvious from the way we form groups. Now we prove part b).

Assume $G$ is an arc group indexed by $2i - a_i$. $a \neq c$ since no two arcs can originate from one element, and $b \neq d$ since no two arcs can have the same target. Since both arcs are in $G$, $a_i = c_i$. Now by the definition of grouping, $b_i = d_i = (a_i \ XOR \ 1)$. Hence $a$, $b$, $c$, $d$ are all distinct.

Q.E.D.

By the above lemma, we know pairs of elements connected by arcs in a group can be combined simultaneously without interfering with each other. Since at most $\lfloor \log n \rfloor + 1$ bits are used to represent the elements in the linked list, at most $2 \lfloor \log n \rfloor + 2$ groups are needed. To determine, for arc $(a, b)$, which group it belongs to, we compute

$c := a \ XOR \ b$;
$c := (((c-1) \ XOR \ c) + 1)/2$;

This will set all the bits of $c$ to zeros except the lowest bit valued 1. Now we use value $c$ to index into a table $T$ to determine which bit of $c$ is 1. Table $T$ can be built by the following simple procedure.

```
TABLE($n, p$)
  begin
    for $j := 0$ to $\log n$ step 1
      $T[2^j] := j + 1$;
  end.
```

It is possible that several processors attempt to read the same entry of the above table simultaneously. We can use this table on the CREW model anyway. The time for table building will not dominate the overall time complexity.

The procedure for determine the arcs' group membership is as follows.

```
MEMBER($X[1..n]$, $NEXTX[1..n]$)
  forall $P_i : 1 \leq i \leq p$
    begin
      for $k := 1$ to $\frac{n}{p}$ step 1
        begin
          $M[i] := X[(i-1)\frac{n}{p}+k]$ XOR $NEXTX[(i-1)\frac{n}{p}+k]$;
          $M[i] := (((M[i]-1)$ XOR $M[i])+1)/2$;
          $A[(i-1)\frac{n}{p}+k] := 2 \cdot T[M[i]]$;
          if ($M[i]$ AND $X[(i-1)\frac{n}{p}+k]) > 0$ then
              $A[(i-1)\frac{n}{p}+k] := A[(i-1)\frac{n}{p}+k]-1$;
        end
    end.
```

After the group of each arc is calculated, the bucket sorting algorithm is used to move arcs so that arcs in the same group are gathered together. Now, one group at a time, all the arcs are visited. This will take no more than $O(\sum_{i=1}^{2\log n+2} \lceil \frac{|G(i)|}{p} \rceil) = O(\frac{n}{p}+\log n)$ time, where $|G(i)|$ is the number of arcs in group $i$. Specifically, when the head of an arc visited by a processor is already deleted due to the previous processing of that arc's predecessor then the processor does nothing, otherwise it combines the head and the tail of the arc, marks the tail deleted, and updates the arc to point to the successor of its successor. After one pass of this 'pair-off', any remaining element must have both its predecessor and successor marked deleted. So at most $\frac{2n}{3}$ elements can survive. The remaining elements are packed and after $O(\log\frac{n}{p})$ passes, there are no more than $p$ elements left. Finally the remaining elements are combined by algorithm PREFIX_DEPENDENT. The following gives the details of our algorithm for the CREW model.

```
PRODUCT_PREFIX($X[1..n]$, $NEXTX[1..n]$, $HEAD$)
  begin
    TABLE($n$, $p$);
    COMBINE($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
  end

COMBINE($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
  forall $P_i : 1 \leq i \leq p$
    begin
      $DELETED[1..n] := false$;
      if $n \leq p$ then
  PREFIX_DEPENDENT($X[1..n]$, $NEXTX[1..n]$, $HEAD$);
      else
        begin
          /* Calculating group membership. */
          MEMBER($X[1..n]$, $NEXTX[1..n]$);

          /* Bring elements of the same group together.*/
          /* BUCKETSORT returns each element's rank
            in array $C$.
          */
          BUCKETSORT($A[1..n]$);
          for $k := 1$ to $\frac{n}{p}$ step 1
            $B[C[(i-1)\frac{n}{p}+k]] := (i-1) \cdot \frac{n}{p}+k$;

          /* Visiting arcs one group at a time. */
          /* Array $G$ is returned by BUCKETSORT.
            $G[k]$ is the total number of elements
            in group $k$.
          */
          $j := 1$;
          for $k := 1$ to $2\log n+2$ step 1
            begin
              while $G[k] \geq p$ do
                begin
                  PAIR-OFF($B[j..j+p]$);
                  $j := j+p$;
                  $G[k] := G[k]-p$;
                end
              if $G[k] \neq 0$ then PAIR-OFF($B[j..j+G[k]]$);
              $j := j+G[k]$;
            end

        /* Now pack. PACK is a simplified version
          of BUCKETSORT. PACK returns the size
          of the packed array.
        */
        $n' := $PACK($A[1..n]$);
        COMBINE($X[1..n']$, $NEXTX[1..n']$, $HEAD$);
  end

PAIR-OFF($B[a..b]$)
  begin
    if $b-a < p$ then Disable $P_i : i > b-a+1$;
    if NOT $DELETED[B[a+i-1]]$ then
      begin
        $X[B[a+i-1]] := X[B[a+i-1]] \bigcirc$
                    $X[NEXTX[B[a+i-1]]]$;
        $DELETED[NEXTX[B[a+i-1]]] := true$;
        $NEXTX[B[a+i-1]] :=$
                $NEXTX[NEXTX[B[a+i-1]]]$;
      end
    Enable $P_i : 1 \leq i \leq p$;
  end
```

Each execution of procedure MEMBER takes $O(\frac{n}{p})$ time and each execution of procedure BUCKETSORT takes $O(\frac{n}{p}+\log n)$ time, for the $n$ numbers range from 1 to $2\log n+2$. As mentioned before, visiting groups of arcs takes $O(\frac{n}{p}+\log n)$ time, and packing can be easily verified to take $O(\frac{n}{p}+\log p)$ time. Thus one pass of the combining process, i.e. an execution of procedure COMBINE, takes $O(\frac{n}{p}+\log n)$ time. After executing COMBINE no more than $\frac{2n}{3}$ elements are left, and hence the following recursive formula for time complexity $T_p$ can be established.

$$T_p(n)=O(\frac{n}{p}+\log n)+T_p(\frac{2n}{3}).$$

Since $O(\log\frac{n}{p})$ passes of COMBINE are used and the remaining less than $p$ elements can be combined in $O(\log p)$ time units, we have $T_p(n)=O(\frac{n}{p}+\log n\log\frac{n}{p})$. All the operations in the algorithm except the ones for calculating the group membership of the arcs are EREW operations.

For the EREW model, one way to determine the group membership for an arc $(a,b)$ is to continuously bisect the bits of $a$ $XOR$ $b$ and ask if the lower half of the bits are 0's. In this binary splitting fashion, we can calculate the group membership for all the arcs in time $O(\frac{n}{p}\log\log n)$. The $\log\log n$ factor can be eliminated by using $p$ copies of table $T$, one copy for each processor. This table $TE$, which is of size $n \cdot p$, has only $(\log n+1)\cdot p$ entries which could possibly be indexed into. These entries are $TE[i][j]$, $1 \leq i \leq p$, $j=2^k$, $0 \leq k \leq \log n$. Table $TE[1..p][1..n]$ can be built by copying relevant entries of table $T$. Distribute processors such that $P_i$, $(k-1)\cdot\frac{p}{\log n+1}+1 \leq i \leq k\cdot\frac{p}{\log n+1}$, $1 \leq k \leq \log n+1$, are assigned to entries $TE[1..p][2^{k-1}]$. These processors will make $p$ copies of the $T[2^{k-1}]$ and put them in $TE[1..p][2^{k-1}]$. The total copying operation will take $O(\log n)$ time. The procedure for determine arcs' group membership is obtained by merely replacing the statement

$$A[(i-1)\frac{n}{p}+k]:=2\cdot T[M[i]]$$

in procedure MEMBER by

$$A[(i-1)\frac{n}{p}+k]:=2\cdot TE[i][M[i]].$$

The table storage technique due to Fredman, Komlós and Szemerédi[3] can be used here to reduce the size of the index table to $O(p\cdot\log n)$ while retaining constant time for table lookup, i.e. $O(\log n)$ space is enough for one copy of the index table. Although the sequential time complexity of the algorithm[3] for building a single copy of the index table is $O(\log^4 n)$, $O(\frac{n}{p}+\log n\log\frac{n}{p})$ time is more than enough when $p$ processors are employed to build a single copy of the index table. Thus, we can first create a single copy of the index table with the help of $p$ processors, then duplicate the table to $p$ copies. The time complexity of $O(\frac{n}{p}+\log n\log\frac{n}{p})$ still holds. The algorithm just obtained is an EREW algorithm.

As mentioned before, for the prefix computation we execute our algorithm first and then reverse the computation and pop out products stored at each node of the product tree.

## 6. Conclusion

Parallel algorithms allowing a maximum number of processors to achieve linear speedup for the data independent prefix problem have already been obtained, even on weaker models such as ultracomputers[9]. This is not the case in the data dependent case. We conclude this paper by raising the question: is $O(n/p+\log n)$ time bound achievable for the data dependent prefix problem on the EREW model?

## References

[1]. D. Bitton, D.J. DeWitt, D.K. Hsiao and J. Memon. A taxonomy of parallel sorting. ACM Computing Survey, Vol. 16, No. 3, Sept. 1984, pp. 287-318.

[2]. F. Fich. New bounds for parallel prefix circuits. Proc. of the 15th Annual ACM symposium on Theory of computing, May 1983, pp. 100-109.

[3]. M.L. Fredman, J. Komlós, E. Szemerédi. Storing sparse table with $O(1)$ worst case access time. J. ACM, Vol. 31, No. 3, July 1984, pp. 538-544.

[4]. D.S. Hirschberg. Fast parallel sorting algorithms. Comm. ACM, Vol. 21, No. 8, Aug. 1978, pp. 657-661.

[5]. P.M. Kogge and H.S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. Vol. C-22, Aug. 1973, pp. 786-792.

[6]. C.K. Kruskal, L. Rudolph, M. Snir. The power of parallel prefix. IEEE Tran. Computers, Vol. C-34, No. 10, Oct. 1985, pp. 965-968.

[7]. R.E. Ladner and M.J. Fischer. Parallel prefix computation. J. ACM, Oct. 1980, pp. 831-838.

[8]. J. Reif. Probabilistic parallel prefix computation. Proc. of 1984 International Conf. on Parallel Processing, Aug. 1984, pp. 493-443.

[9]. J.T. Schwartz. Ultracomputers, ACM Transactions on Programming Languages and Systems, Dec. 1980, pp. 484-521.

[10]. M. Snir. On parallel searching. SIAM J. Comput. Vol. 14, No. 3, Aug. 1985, pp. 688-708.

[11]. J.C. Wyllie. The complexity of parallel computation, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

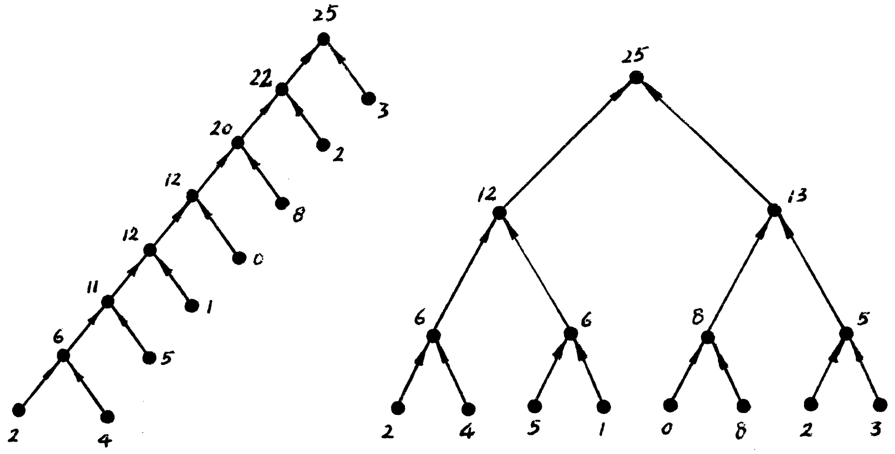


Fig. 1. Two product trees. ○ is addition here.

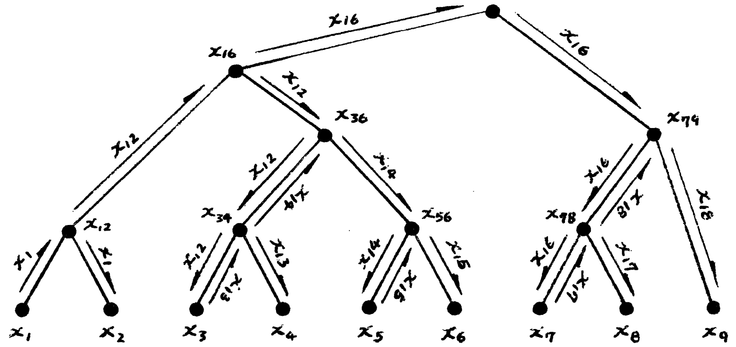


Fig. 2. Use a product tree to construct a prefix algorithm. $x_{ij}$ denotes $\bigcirc_{k=i}^{j} x_k$.

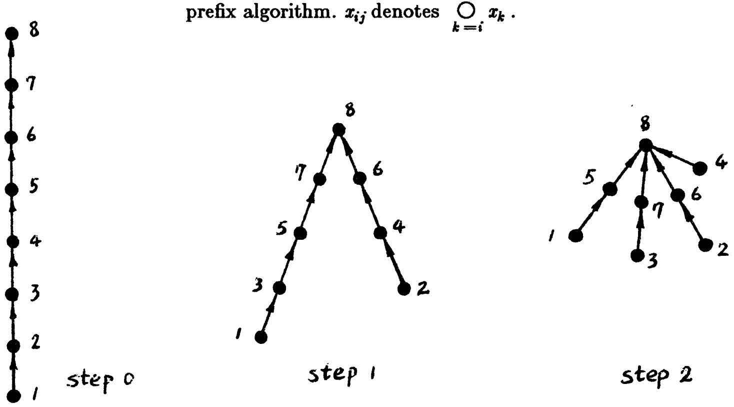


Fig. 3. A linked list of $n$ items can be transformed into a tree of maximum depth $\frac{n}{p}+1$ in $O(\frac{n}{p}\log p)$ time, using $p$ processors.

# A PARALLEL RANGE SEARCH ALGORITHM USING MULTIPLE ATTRIBUTE TREE

*Nageswara S.V. Rao* and *S.S. Iyengar*

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA

*R.L. Kashyap*

Department of Electrical Engineering
Purdue University
West Lafayette, IN 47907,USA

*ABSTRACT*

The problem of range search arises in many applications in areas such as information retrieval, database, robotics and computational geometry. There are a good number of sequential algorithms for this problem based on data structures such as k-d tree, quad tree, range tree, d-fold tree, super B-tree, overlapping k-ranges, non-overlapping k-ranges, etc. In this paper we present a parallel algorithm for a Single Instruction Multiple Data (SIMD) computing system with $p$ processing elements. We make use of the linearized multiple attribute tree as the underlying data structure. Our algorithm has the complexity of $O(kN/p)$, $p \leq N$ where $k$ is the dimensionality and $N$ is the number of points of the data space.

## 0. INTRODUCTION

The problem of range search arises in many application areas such as information retrieval, database, robotics, and computational geometry. There have been many sequential algorithms for range search problem using the data structures such as k-d tree, quad tree,- range tree, overlapping k-ranges, nonoverlapping k-ranges, d-fold tree, super B-tree, k-fold tree, and multiple attribute tree [1,3,5-7].

In this paper, we present a parallel algorithm for the range search problem. We make use of the Multiple Attribute Tree (MAT) as the underlying data structure. Our model of computation is a *Single Instruction Multiple Data* (SIMD) computing system, consisting of $p$ processing elements. The system has a single shared memory that supports simultaneous reads. The time complexity of our algorithm is given by $O(kN/p)$, $p \leq N$.

The organization of this paper is as follows: In Section 1, we discuss the MAT data structure and the corresponding directory. In Section 2 the basic range search algorithm is discussed, and its complexity is estimated. Firstly an $O(k)$ algorithm is obtained using a processor array of size $N$ in a straight forward manner. Then, an $O(kN/p)$, $p \leq N$ algorithm using a processor array of size $p$ and an augmented directory.

## 1. THE MULTIPLE ATTRIBUTE TREE

The MAT data structure was first introduced and analyzed by Kashyap et al [2]. The MAT is shown to outperform the inverted file structure for partial match and complete match queries in the cases when the directory resides on the main and secondary memories in [4]. An exhaustive treatment on various structural aspects of MAT can be found in [5].

The k-dimensional MAT on $k$ attributes $A_2,A_3,...,A_k$ for a set of records is defined as a tree of depth $k$, with the following properties [4]:

i) it has a root at level 0,

ii) each child of the root is a (k-1)-dimensional MAT, on (k-1) attributes, $A_1,A_2,...,A_k$, for the subset of the records that have the same $A_1$ value. This value is the value of the root of the corresponding (k-1)-dimensional MAT, and

iii) the child nodes of the root are in the ascending order of their values. This set of child nodes is called the *filial-set.* □

From the above definition we note that there is a root node at level 0, and it does not have any value. Every node of level $i$,$i$=1,2,...,$k$, corresponds to a value of the attribute $A_i$. Fig 1(b) shows the MAT data structure for the sorted set of records of Fig.1(a). The attributes $A_1,A_2,...,A_k$ form the hierarchy of of levels 1 through $k$. The nodes of every filial-set are ordered according to their values. Another important property is that each record or point is represented by a unique path from the root to the corresponding terminal node. The total number of nodes in the MAT is at most $kN$ for a given data set containing $N$ records or points.

The MAT is *linearized* and stored as a directory. We make use of the *breadth-first* linearization in which the nodes are stored in the order they are encountered when MAT is traversed in a breadth-first manner. The directory is an array of $M$ ($\leq kN$) directory elements and each directory element has the following fields:

```
directory-element =   record
             value:   1..M;
       first-child:   1..M;
        last-child:   1..M;
                      end;
```

where the fields corresponding to a node $T$, at level $j$, and numbered $n$ are defined as follows:

value: the value of the node $T$;
first-child: node number of the first child of the child set of $T$;
last-child: node number of the last child of the child set of $T$;

The idea of breadth-first linearization is illustrated in Fig.2 and the corresponding directory is shown in Fig.3. In the first-child field of leaf nodes contain the pointers to the corresponding records. The time complexity of constructing this directory using a uniprocessor system is $O(N \; logN)$

0190-3918/86/0000/0931 $01.00 © 1986 IEEE

| $A_1$ | $A_2$ | $A_3$ | $A_4$ | Record pointer |
|---|---|---|---|---|
| 1 | 1 | 3 | 3 | 1 |
| 1 | 2 | 1 | 2 | 2 |
| 1 | 1 | 4 | 1 | 3 |
| 1 | 1 | 2 | 6 | 4 |
| 2 | 3 | 5 | 7 | 5 |
| 1 | 1 | 3 | 5 | 6 |
| 1 | 2 | 5 | 6 | 7 |
| 2 | 3 | 5 | 1 | 8 |

FIG. 1(a) INPUT DATA FILE

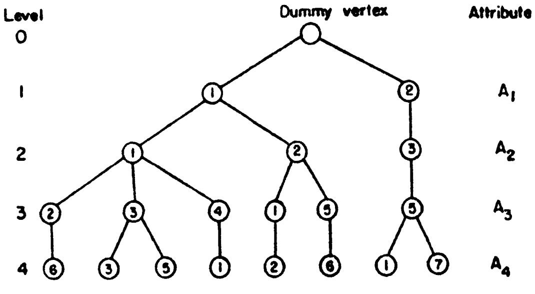


FIG. 1(b) MAT representation for data of FIG.1(a)

(including the time required to sort the records on all the attributes) [5]. Here, we are concerned only with answering the range query, assuming that the directory is available in memory. This directory is utilized in the design of parallel range search algorithms in the next section.

## 2. PARALLEL RANGE SEARCH ALGORITHM

A range query is given by $Q = \bigcap_{i=1}^{k} q_i$, where $q_i$ specifies the range $[l_i, h_i]$ at the level $i$. In geometric terms a range query specifies a *rectilinearly oriented hyper rectangle* in $k$ dimensional space. Answering a range query calls for the retrieval of the points enclosed by the specified hyper rectangle, and these points form a 'sub' MAT, called the *query* MAT (QMAT), on the original MAT. These nodes of the QMAT are called the *qualified* nodes of MAT for the query $Q$. Answering a range query involves identifying the nodes of the QMAT on the MAT. Let $R$ be the number of records contained in the hyper rectangle. The number of the nodes of the QMAT is no more than $kR$.

An algorithm for range query proceeds by descending down the MAT level by level, starting from the first level. At each level $i$, the child sets of qualified nodes of previous level are searched for the containment in the range $[l_i, h_i]$. The algorithms is as follows:

```
algorithm RANGE-SEARCH(level,qnodes);
begin
    tempset:= $\phi$;
    for each $n \in$ qnodes do
        add to tempset all the child nodes of $n$ that lie in
        the range $[l_{level}, h_{level}]$;
    if level < $k$
    then RANGE-SEARCH(level+1,tempset)
    else return the pointers given by the elements of tempset;
end
```

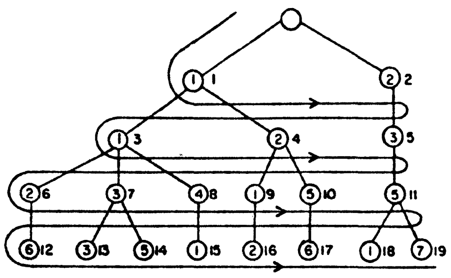


FIG. 2 Breadth - first linearization

In the algorithm RANGE-SEARCH, at a level $i$, all the nodes that satisfy the partial query $\bigcap_{j=1}^{i} q_j$ are collected as tempset. In the next level $i$+1, the child nodes of these nodes are tested for inclusion in the range $[l_{i+1}, h_{i+1}]$. At the final level, the pointers to the information about the records that satisfy the given query are retrieved.

The MAT data structure has some builtin parallelism with respect to answering a range query. The searching for the qualified nodes can be simultaneously carried out on the sub MATs of the same level. However, there seems to be a stringent sequentiality in the way the attributes are processed one after the other. Again, speaking in geometric terms, processing of each attribute reduces the dimensionality of the search space by one. From the above discussion, we conclude that $O(k)$ is lower bound on the steps involved in answering a range query on the MAT-based approach.

The model of computation is in the form of an array of processor elements $PE_j$, $j$=1,2,...$p$, and operates in *Single Instruction and Multiple Data* (SIMD) mode. More specifically, each processor element executes the same algorithm in synchronism with all the other processor elements. We use single shared memory in which the breadth-first MAT directory and other variables are stored. As will be shown later there will be no conflicts in writing into the memory. But, simultaneous read operations are supported. The range query is represented by low-limit[$i$] and high-limit[$i$], $i$=1,2,...,$k$, which give the lower and upper limits of the range for the attribute $A_i$.

Firstly, consider the processor array $PE_i$, $i$=1,2,...$N$ where $N$ is the number of records in input set. The array base[$i$], $i$=1,2,...,($k$+1) is stored in the memory, where any node of level $i$ lies in between the entries indexed by base[$i$], and base[$i$+1]-1 in the directory. We use the array enable[$i$], $i$=1,2,...$N$ to selectively enable and disable the processor elements of the processor array. The algorithm consists of $k$ steps and in step $i$, $i$=1,2,...,$k$ the level $i$ is processed to obtain the qualified nodes of the level $i$. Each enabled processing element acts on a child node of a qualified node at previous level, and checks if the child node satisfies the range constraint of the current level. The processor elements corresponding to the qualified nodes of current level are enabled in the next step. For the final level, the points that satisfy the range query are obtained. The algorithm executed by the processor element $PE_j$, $j$=1,2,...$N$ for the step $i$, $i$=1,2,...$k$ is as follows:

```
algorithm $PE_j$;
begin
1.   if enable[$j$]
2.   then
     begin
3.       enable[$j$] = false;
4.       $n$ = base[$i$] + $j$;
5.       if low-limit[$i$] ≤ value[$k$] ≤ high-limit[$i$]
6.       then
         begin
7.           for $l$ = first-child[$n$] to last-child[$n$] do
8.               enable[$l$ - base[$i$]] = true;
         end;
     end;
end;
```

In the above algorithm the qualified records have to be returned in the final level by suitably modifying the lines 7 and 8. This parallel algorithm implements the algorithm RANGE-QUERY and it is very easily seen that this correctly answers the range query. Since the algorithm is executed in $k$ synchronous steps, it is evident that the time complexity of this algorithm is $O(k)$.

In applications involving large amounts of data, the value of $N$ could be very large and the assumption of the array of $N$ processors is not very pragmatic. We now present an algorithm that utilizes an array $PE_i$, $i=1,2,...,p$, $p \leq N$ processor elements. The basic idea of 'processing the MAT nodes level by level' is still followed; at any level the processor array is invoked required number of times along breadth of the MAT. In this case the directory is augmented with two more fields - enable and level. The former is used to selectively enable and disable the processor elements and latter field gives the level of the node. The global variable current-level gives the level-number that is currently being processed, and it is incremented after each level is processed. The array index[$i$], $i=1,2,...,p$ gives the current index (into the directory) to be used by the processor element $PE_i$. Initially, the enable field is made true for all the nodes of level 1 and also the entries of the array index corresponding to the first level nodes are filled up. For the final level, the qualified records are retrieved. The algorithm executed by the processor element $PE_j$, $j=1,2,...,p$ is as follows:

```
algorithm $PE_j$;
  begin
1.    $l$ = index[$j$];
2.    if ((current-level = level[$l$]) and (enable[$l$]))
3.    then
      begin
4.        if (low-limit[current-level] ≤ value[$l$]
                             ≤ high-limit[current-level])
5.        then
          begin
6.            for $m$ = first-child[$l$] to last-child[$l$] do
7.                enable[$m$] = true;
          end
8.        index[$j$] = $l$+$p$;
9.        enable[$l$] = false;
      end;
  end;
```

The if statement in line 2 makes sure that all processors of the SIMD array process the nodes of the current level. After the first $p$ nodes of a level are processed, the next $p$ nodes are processed by the use of the array index as in line 8. The lines 6 and 7 are to be modified to retrieve the qualified records in the final level. It is straight forward to see that the range query is processed correctly.

**THEOREM**: The time complexity of the parallel range search algorithm, on an SIMD system of $p$ processors is $O(kN/p)$.

**PROOF:** For the execution of the algorithm, at any level $i$, the maximum number of times the processor array is invoked is given by:

$$\lceil \text{(number of nodes of level } i/p \rceil$$

The total number of times the processor array is invoked is

$$\sum_{i=1}^{k} (\lceil \text{(number of nodes of level } i\text{)}/p \rceil)$$

$$= (\sum_{i=1}^{k} \lceil \text{ (number of nodes of level } i \rceil)/p = O(kN/p+k)$$

Hence, the time complexity of this algorithm is $O(kN/p)$. □

We note that for the case $p=N$ this algorithm has the same complexity as the earlier one. But, earlier is superior as it uses less memory space. We also note that the maximum number of times the processor array is invoked is given by $k(N/p+1)$. This upperbound corresponds to the wosrt-case structure of the MAT. In an average-case the number of times that the processor array is invoked will be less than this bound. The authors are presently working on this aspect.

## 3. CONCLUSIONS

We have developed an $O(kN/p)$ algorithm for range search problem using the linearized MAT data structure and SIMD computing system. Since the attributes are processed one after the other, the MAT based parallel algorithm has a lower bound complexity of $\Omega(k)$ on the processor array containig no more than $N$ processor elements.

## 4. REFERENCES

[1] BENTLEY, J.L., and FRIEDMAN, J.H., "Data structures for range searching",**ACM Computing Surveys**, 11, 4(Dec. 1979), 397-409.

[2] KASHYAP, R.L., SUBAS, S.K.C., and YAO, S.B., "Analysis of multiattribute tree organization", **IEEE Trans. Software Engineering**, SE-2, 6(1977), 451-467.

[3] MEHLHORN, K., *Data Structures and Algorithms 3: Multidimensional Tree Structures and Computational Geometry*, EATCS monograph on Theor. Computer Sci., Springer-verlag, 1984.

[4] NAGESWARA RAO, S.V., IYENGAR, S.S., and VENI MADHAVAN, C.E., "A comparative study of multiple attribute tree and inverted file structures for large bibliographic files", **J. Inf. Process. and Mgmt.** 21, 5(1985), 433-442.

[5] NAGESWARA RAO, S.V., and IYENGAR, S.S., The Multiple Attribute Tree Structure, Tech. Report TR85-030,Dept. of Computer Sci., Louisiana State Uni., Baton Rouge,June 1985.

[6] OVERMARS, M.H., *The Design of Dynamic Data Structures*, Lect. Notes in Comput. Sci. 156, Springer-verlag, 1984.

[7] WILLARD, D.E., "New data structure for orthogonal queries", **SIAM J. Computing**, 14, 1(Feb. 1985), 232-253.

# DESIGN AND EVALUATION OF A PARALLEL SORT UTILITY

**Micah Beck**
**Dina Bitton**

Computer Science Department
Cornell University
Ithaca, NY 14853

**W. Kevin Wilkinson**

Bell Communications Research
Morristown, NJ 07960

**Abstract.** A fundamental measure of processing power in a database management system is the performance of the sort utility it provides. When sorting a large data file on a serial computer, performance is limited by factors involving processor speed, memory capacity, and I/O bandwidth. In this paper, we investigate the feasibility and efficiency of a parallel sort-merge algorithm through implementation on the *JASMIN* prototype, a backend multiprocessor built around a fast packet bus. We describe the design and implementation of a parallel sort utility. We then present and analyze the results of measurements corresponding to a range of file sizes and processor configurations. Our results show that using current, off-the shelf technology coupled with a streamlined distributed operating system, three and five microprocessor configurations provide a very cost-effective sort of large files. The three processor configuration sorts a 100 megabyte file in one hour, which compares well with commercial sort packages available on high-performance mainframes.

## 1. Introduction

Sorting of large data files is one of the most important and frequently performed operations in database management. Output files generated by a report are usually sorted with respect to an attribute or a combination of attributes that are of interest to the users of the report. In mass-storage devices, files are often maintained sorted with respect to a key attribute in order to facilitate subsequent searching and processing. Database management systems also pre-sort files in order to eliminate duplicate records [5, 14] or to process complex join and aggregate queries [14]. Thus, one of the fundamental measures of processing power in a transaction management system is the performance of the sort utility it provides [1].

In most systems, sort performance is not satisfactory. For large files, a sort may require hours of processing time during which it saturates the CPU and I/O devices. As a consequence, even in high-performance transaction management systems, a large sort will often be deferred to after-hours processing in order not to interfere with the execution of short interactive transactions.

What limits the performance of a file sorting operation, and how can this performance be enhanced? A theoretical machine with a 100 megabyte memory and a 50 nanosecond Compare instruction could sort one million 100 byte records in 1.5 minutes, including 30 seconds to read the file from a disk with a 3 million-byte per second transfer rate and 30 seconds to write it back to the same disk[a] [1]. In practice, file sizes exceed main memory capacity by several orders of magnitude, comparisons and record moves are slower, and the effective I/O bandwidth is well below the maximum bandwidth of the disk channel. In addition, general purpose operating systems impose processor overhead and limit the utilization of other resources. The designer of a sort utility must strive to minimize record moves in memory, overlap I/O latency with computation, and reduce disk seek times. Even then, the efficiency of file sorting on a serial computer with conventional mass-storage remains severely limited. Parallel computation combined with high I/O transfer rates is a natural approach to overcoming these limits. Indeed, a number of recent database machine designs use parallel sorting as a fundamental building block for query processing [8, 9, 15, 16].

In this paper, we investigate the feasibility of parallel file sorting on a backend multiprocessor machine. We describe the design of a sort utility that exploits parallel computation and high I/O bandwidth, and its implementation on the *JASMIN* multiprocessor. We consider two schemes which differ in the starting location of the unsorted data (Figure 1), and may lead to significant differences in performance. The first scheme corresponds to a *Backend Sort*, where a file initially located on the host is downloaded to a backend multiprocessor for sorting. The file is distributed to the backend processors, which sort it and then return the sorted file to the host. The second scheme, a *Distributed Sort*, assumes that the source file is initially distributed across a number of disks attached to backend processors. The processors sort the file in parallel, then send it to the host or write it to a backend disk. Basically, the parallel sorting algorithm used in both schemes is the same, but the design goal of overlaping computation with the distribution of the data in one case (*Backend Sort*), is replaced with overlapping computation with disk access in the other (*Distributed Sort*).

The remainder of this paper is organized as follows. In Section 2, we describe the hardware and software architecture of the *JASMIN* system. In Section 3, we describe the parallel external sort algorithm, and the implementation choices that we have made – in particular the process structure and the buffering scheme. In Section 4, we present the performance of our algorithm as implemented on *JASMIN*. We analyze the execution time, processor utilization, and network traffic as functions of the file size and the multiprocessor configuration. Finally, Section 5 contains our conclusions.

Support for this work was provided by the National Science Foundation, under grant DCR-8410889, a Hewlett-Packard faculty development fellowship, and Bell Communications Research.

[a] With an efficient algorithm, sorting $n$ records in main memory requires $O(nlogn)$ record comparisons and exchanges. For $n$ = 1,000,000, a typical proportionality of 1.5 and 20 instructions for a comparison and an exchange, this amounts to 600M instructions.

0190-3918/86/0000/0934 $01.00 © 1986 IEEE

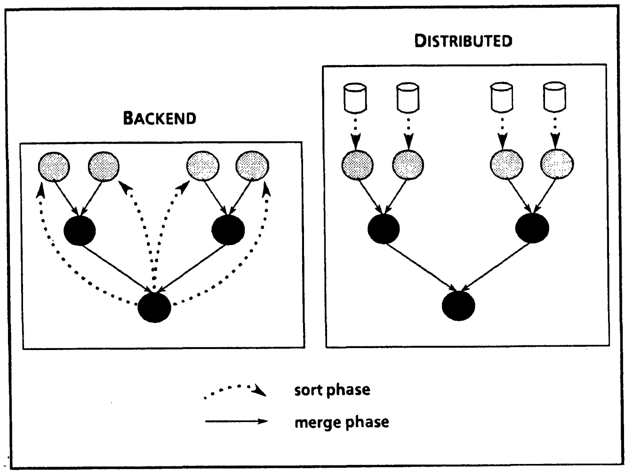


Figure 1: Backend and Distributed Sorting

## 2. A Backend Multiprocessor

Our parallel external sort algorithm was implemented on a prototype AT&T Bell Laboratories multiprocessor [2]. This multiprocessor may be configured with up to 12 microcomputers communicating through a fast packet bus. It also has an umbilical connection to a DEC VAX host running UNIX[b] System V. The backend processors run the *JASMIN* distributed operating system kernel, which is under development at Bell Communications Research. *JASMIN* is designed to efficiently support distributed applications. In this section, we briefly describe the hardware characteristics of the multiprocessor and process management and communication in *JASMIN*. The reader is referred to [12] for a complete description of the operating system and to [7] for a description of a distributed database management system which runs on *JASMIN*.

### 2.1. The S/Net

The multiprocessor used in *JASMIN* is built around the S/Net, a fast (80 Mbit/s) packet bus developed at AT&T Bell Laboratories. The multiprocessor consists of a set of Multibus-based microcomputers, each with an S/Net Processor Interface Board. The S/Net itself consists of a set of Buffer Interface Boards, tied together on a single backplane, each connected to a Processor Interface Board. A designated Buffer Interface Board is connected to the VAX host. The VAX downloads the microcomputers via the S/Net and provides some support, such as file access and debug output, for the execution environment. The microcomputers consist of a 10 MHz Motorola 68000 CPU and 1 Megabyte of main memory. Some of the microcomputers also have an SMD disk controller and a Fujitsu Eagle local disk.

While the total bandwidth of the S/Net is 80 Mbit/s, no single processor can achieve that speed. The maximum kernel level data transfer rate that a single processor can attain on the S/Net is approximately 300 Kbyte/sec, or 3% of the network's total capacity. Operating system overhead further reduces the effective data transfer rate between processes to 100 Kbyte/sec.

### 2.2. The *JASMIN* Operating System

The *JASMIN* distributed operating system is modeled after DEMOS [3]. The basic object implemented by the system is an interprocess communication capability called a *path*, over which small, fixed length messages can be sent. Holding a path enables a process to send a message to the creator of the path. Additionally, bulk data transfers between processes may be accomplished by attaching a buffer to a path in the creator's address space and using kernel primitives to move data into or out of the buffer. The system allows any process to create paths and to pass them to other processes via messages.

A set of processes which share text and data address space, but have separate execution stacks, are said to form a *load module*. Since memory is shared by these processes, a load module cannot span processors. While processes in the same load module share text and data space, their execution stacks inhabit different memory segments, they hold different sets of paths, and they receive messages from separate queues. In other words, paths and messages are exchanged between processes, not load modules. The load module notion is a convenience to permit cooperating processes to share memory.

*JASMIN* process scheduling is particularly simple. When created, each process is assigned a static priority level. Within each priority level, scheduling is strictly non-preemptive. A running process will execute until it performs a blocking system call, or until a higher priority process becomes ready-to-run. This guarantees mutual exclusion among a set of processes operating on shared data structures within a load module, as long as these processes are run at the same priority level.

## 3. A Parallel File Sorting Algorithm

Large files cannot usually be sorted in main memory, and so file sorting requires an *external sort algorithm*. Most external sort algorithms are based on iterative merging. They partition the file into subfiles that separately fit in memory, internally sort the individual subfiles, and then iteratively merge them into a single sorted file. During each iteration, every record is read and written once to mass-storage. These serial algorithms can be parallelized in a number of ways [4, 6, 10].

### 3.1. The Algorithm

We have implemented a parallel N-way merge-sort algorithm adapted from [4]. The algorithm assumes a logical tree connection between the processors (Figure 2). Each of the leaf processors has a disk attached to it, and thus can perform an external sort, independently of the host and the other processors in the backend. The parallel algorithm has two phases. We will refer to the first phase as the *sort phase*, and to the second as the *merge phase*. During the *sort phase*, each leaf processor creates fixed-length sorted subfiles from its partition of the file; during the *merge phase*, these subfiles are merged in parallel. Parallelism in merging is exploited both through pipelining merge steps between levels of the tree, and through concurrent merging performed by processors on the same level of the tree. The two phases of the algorithm are described in more detail below.

During the sort phase, fixed-length sorted subfiles are created by the leaves as in a serial sort. Partitions of the file are sorted in memory using Quicksort, and then written to disk. Available memory is divided between a subfile and an array of pointers to records in that subfile. During Quicksort, data records are not moved in memory, only pointers are

[b] UNIX is a trademark of AT&T Bell Laboratories.

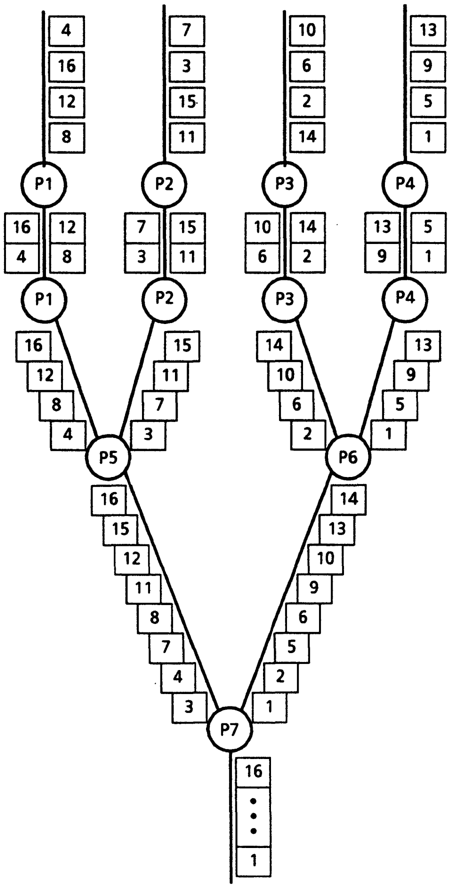


**Figure 2: A Parallel Merge-Sort**

changed.

After the sort phase completes, each leaf merges its subfiles, producing a single sorted partition of the file. As it is produced, the partition is transferred to the leafs parent in the merge tree. The parent processor merges the partitions it receives from its children and sends the result one level up in the tree. The complete sorted file is produced at the root of the merge tree. In our implementation, merge streams are transferred in blocks containing many records. While pipelining of successive merge phases *could* be at the record granularity, communication overhead makes such a design prohibitively inefficient.

It is interesting to note that, given a fixed topology for the merge tree, most of the steps in our sort algorithm are linear in terms of the number of comparisons performed. The partitioning of $n$ records into $n/b$ fixed-size sorted blocks of $b$ records each takes $O(nlog(b))$ comparisons. Similarly, merging $d$ sorted streams of $n/d$ records takes $O(nlog(d))$ comparisons. Thus, at non-leaf nodes, where the degree of the merge is fixed by the topology of the tree, the merge is linear. At the leaves, the degree of the merge is equal to the number of temporary files, which is in turn proportional to the size of input. The merge phase at the leaves is thus the only point where the number of comparisons is non-linear. Because comparisons are a small portion of the total work done in sorting a large file, even the uniprocessor configuration of our sort only shows significant non-linear behavior in file sizes over 25 megabytes (see Section 4.2).

### 3.2. Implementation on *JASMIN*

In order to concentrate on designing the parallel part of the sort utility, and have a good baseline for evaluating our implementation, we chose to start with a robust serial sort program. We considered the *Linderman Sort*, a program recently developed at Bell Laboratories [13]. The program has been extensively tuned, and achieves very good performance on UNIX. The algorithm implemented in Linderman's program is a polymerge sort [11], where initial runs are generated with Quicksort. Each sorted run is stored as a temporary file. Then the files are merged in one or more $N-way$ merge passes. We started with adapting this code for sorting the subfiles in the sort phase of our parallel algorithm, and for merging subfiles on one processor. Then, we turned our attention to design issues that are critical to the performance of the parallel sort program:

(1) the process configuration and synchronization protocol

(2) the memory configuration (including buffer allocation),

(3) the layout of files on the local disks.

**3.2.1. Process Configuration** On each backend processor, processes were configured so that delays due to data transfer and processor synchronization would be minimized. In particular, data transfers (processor-to-disk, or processor-to-processor) had to be overlapped with computation whenever possible. To avoid copying data within a processor, we adopted a structure consisting of a single *JASMIN* load module per processor. Multiple processes within each load module share access to a single buffer pool, but communicate and synchronize using messages.

Data transfers are organized as *streams* of record blocks. Each stream consists of a series of fixed size blocks containing a number of records. There are three types of streams, each with its own block size. *Input streams* carry unsorted record blocks from the root to the leaves. *Disk streams* carry sorted temporary files between a leaf and its disk. *Merge streams* carry sorted record blocks between processors in the merge tree.

During the sort phase, the records to be sorted are sent as input streams from the root to the leaf processors. In order to overlap communication latency, leaf processors interact with multiple communications processes in the root. In the leaf processors, an *input spooler* process reads the *input stream* from the the root, and builds *sort buffers*, which consist of data records and an array of pointers to these records (Figure 3).

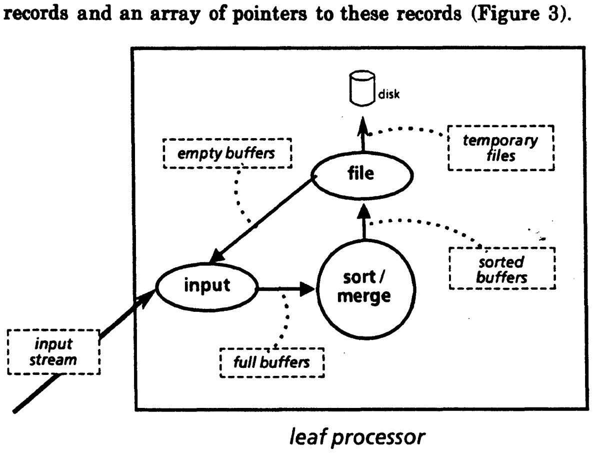


**Figure 3: Sort Phase Leaf Process Structure**

Once a sort buffer is built, it is passed to the main *sort/merge process.* This process performs a Quicksort on the pointer array, and passes the buffer to the *file spooler* process. The file spooler moves records for the first time from the input blocks, in which they were received, to disk blocks which are then written to disk. Each sorted buffer forms one *temporary file* on the disk. The use of two sort buffers per processor ensures a high degree of overlap between input, sorting and writing to disk.

During the merge phase, the sort buffers are unused and so are returned to free memory for use as disk or merge block buffers. The sort/merge process is central to the merge phase on both leaf and non-leaf processors (Figure 4). In the leaves, it merges sorted disk streams from the temporary files, one stream per file. In a non-leaf node, merge streams from the node's children in the tree are merged. In the former case, disk streams are read through the file spooler, which performs double buffering (disk read-ahead). In the latter case, a special *network spooler* process synchronizes data transfer from child processes and implements double buffering of merge streams. On all non-root processors, the output spooler process implements double buffering of merge streams being sent to the parent in the merge tree.

**3.2.2. Memory Configuration** Having only one megabyte of main memory attached to each processor restricts the maximum size of data buffers. After loading the operating system, utility tasks, the text segment for the sorting program, a stack segment for each sorting process, and miscellaneous sort program variables, about 576K remains for buffers. This led us to a configuration of two 256K sort buffers, resulting in temporary files somewhat smaller than that (the buffer must hold not only the records, but also an array of pointers to them). In contrast, a mainframe sort routine will typically use a 1 megabyte sort buffer.

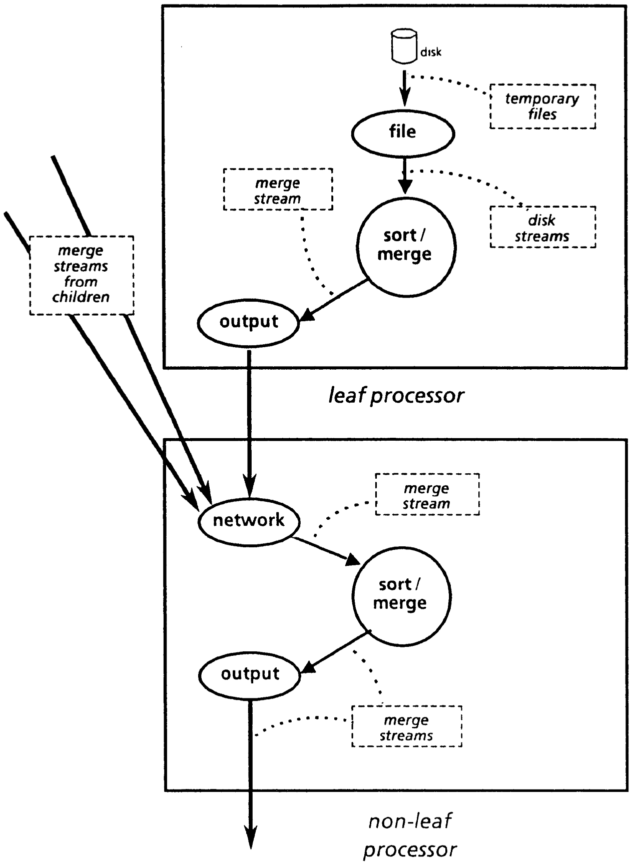


**Figure 4: Merge Phase Process Structure**

During the merge phase, each leaf processor must read a potentially large number of disk streams, and merge them to produce a single merge stream. Each stream is double buffered so the total memory used when merging $n$ files will be

$$M = 2nB_{disk} + 2B_{merge}$$

where the $B_x$ are block sizes. Now, the memory used as sort buffers during the sort phase can be reused as disk and merge blocks during the merge phase. Thus, all 576K are available for allocation. If we are to sort a 100M file using two leaf nodes, then each leaf must handle 50M, or 200 temporary files. Rearranging our equation, we see that

$$B_{disk} = (M - 2B_{merge})/2n \le (576K/400) \approx 1.4K$$

So, choosing $B_{disk} = 1K$, we get $B_{merge} \le 88K$.

The constraint on the size of a merge block does not arise at the leaves, but at non-leaf nodes. If a non-leaf node merges $m$ merge streams to form a single sorted merge stream, then it requires $2m + 2$ merge blocks for buffering. Since non-leaf nodes use no memory for disk blocks, if we want to handle values of $m$ up to 8, we get

$$B_{merge} = M/(2m + 2) \le 576K/18 \approx 32K$$

Thus, the demands of the problem we posed – sorting a 100Mbyte file in configurations ranging from two to eight leaf processors using varying tree structures – leads to a rather narrow choice of buffer sizes. We used the closest powers of two:

$$(B_{disk}, B_{merge}) = (1K, 32K)$$

**3.2.3 Disk File Layout** The layout of files on disk is central to minimizing access time. We experimented with two approaches: contiguous files and interleaved files. In contiguous file layout, the logical disk blocks of a file lie in consecutive physical disk addresses. This means they will be physically contiguous unless they cross a cylinder boundary. In interleaved file layout, the first logical block of all files lie in one consecutive set of disk addresses, followed by the second block of all files, etc. (Figure 5).

Contiguous file layout minimizes seeking when reading or writing an entire sequential file. It maximizes seeking when multiple sequential files are to be accessed concurrently.

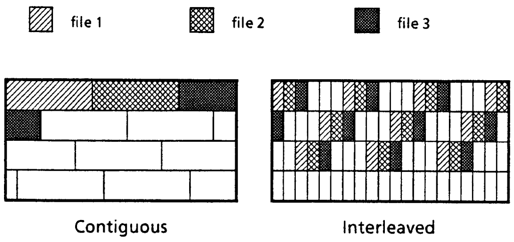


**Figure 5: Contiguous vs Interleaved File Layout**

Interleaved layout achieves the opposite: maximal seeking when accessing a single file sequentially; minimal seeking when accessing many files concurrently. Intermediate schemes which interleave files in segments larger than one logical block are possible, but we did not implement any. In our sort algorithm, temporary files are written one at a time during the sort phase, and then all are read concurrently during the merge phase. Thus, contiguous layout performs best during the sort phase, while interleaved layout wins in the merge phase.

### 3.3 Choosing Parameter Values

We experimented with various ways to partition the 576K of memory that were available for buffer configuration in each processor. The parameter to which the sort time was most sensitive was merge block size. Larger merge blocks decreased the overhead during the merge phase, and reduced the number of interprocessor synchronization points. With double buffering, the maximum size of a merge block that we could choose was 32K (see Section 3.2). To a lesser extent, increasing the input block size also increased sort speed and we settled on an input block size of 8K. Larger input blocks caused wasteful fragmentation of sort buffers.

The large amount of idle time in the leaves during the merge phase suggests that double buffering of disk blocks may not be advantageous. It may be better to consolidate the two buffers into one large buffer to reduce the number of disk accesses. Additional enhancement could also come from improving the scheduling of network data transfers. In the current implementation, the network spooler in a parent node transfers merge blocks from the node's children in the order in which they are requested. However, at the same time, double buffering may cause network traffic that blocks the transfer of data which is needed for merging to proceed. Children are serviced in approximately round-robin order, giving no priority to a transfer which is holding up the merge.

We compared the performance of our sort with the two alternative file layouts: *contiguous* and *interleaved*. Using the interleaved scheme provided an overall improvement of 5% in elapsed time.

## 4. Measurements

In this section, we present and analyze our measurements for a range of backend configurations and file sizes. In order to establish a performance baseline, we start with our uniprocessor sort (*JS1*) on *JASMIN*, and compare it to the Linderman Sort on a fast UNIX machine (Section 4.2). We then analyze in detail the performance of the parallel *Backend Sort*, for different multiprocessor configurations (Section 4.3). Finally, we estimate elapsed times for the *Distributed Sort* (Section 4.4).

### 4.1. Parameters Varied

In our experiments, we varied the following parameters:

(1) Number of processors in the backend:

We configured the backend as a single processor, as a two-level tree with 3 and 5 processors (2 and 4 leaves), and as a three-level binary tree with 7 processors (4 leaves, 2 internal nodes). In subsequent tables, these configurations are labeled as *JS1, JS3, JS5,* and *JS7*, respectively. In all the tree configurations, each leaf processor had a local disk.

(2) Structure and size of the data file:

We varied the size of our data file from 1 to 100 megabytes. The data was randomly generated as a stream of 100 byte records, with a 10-byte ascii-numeric sort key. The record format included a header consisting of a 2-byte record-length field and a 2-byte key-length field, which were interpreted by the sort program. Thus, our implementation can also sort variable length records. In addition to the data records, we used two types of control records, *end of block* and *end of file*, in order to form streams of records. These control records are four bytes long.

We also varied the memory configuration and the file layout scheme. However, all the measurements presented below correspond to experiments with double buffering and interleaved file layout, since these appeared to be more efficient.

### 4.2. Uniprocessor Efficiency

In order to establish a baseline performance measure for our implementation, we compared a single-processor configuration to the UNIX sort [13] on a faster machine. Figure 6 shows the times recorded for sorting files of 1 to 50 megabytes on a *JASMIN* 68000-based microcomputer and on a CCI Power-6 computer running UNIX 4.2 BSD. In both cases the disk was a Fujitsu Eagle. The *JASMIN* figures constitute a baseline that we will use to evaluate the parallel speedup achievable in multiprocessor configurations. We will use the UNIX-CCI numbers as a baseline for comparing the performance of a fast, general-purpose serial machine with that of a backend multiprocessor.

The 68000-based microcomputer is rated at 0.6 MIPS, while the CCI processor is rated at 5 to 6 MIPS. Considering the difference in processor capabilities, the comparison greatly favors the *JASMIN* sort. The UNIX-CCI sort is 1.6 times faster for small files, but only 1.2 times faster for large files. From Figure 6, we observe that, up to 35M, the rate of the *JASMIN* sort is almost constant at 18 Kbyte/sec. On the other hand, the rate of the CCI-UNIX sort degrades rapidly for file sizes above 20M. This degradation is at least partly due to the limit of 20 on the number of open files in a UNIX process, which necessitates multiple N-way merge passes in the merge phase. The comparison illustrates the limits imposed by a general-purpose operating system (UNIX) compared to *JASMIN*. The *JASMIN* kernel essentially gives an application the full power of the underlying computer. In this application, we observed a processor utilization rate of 90%.

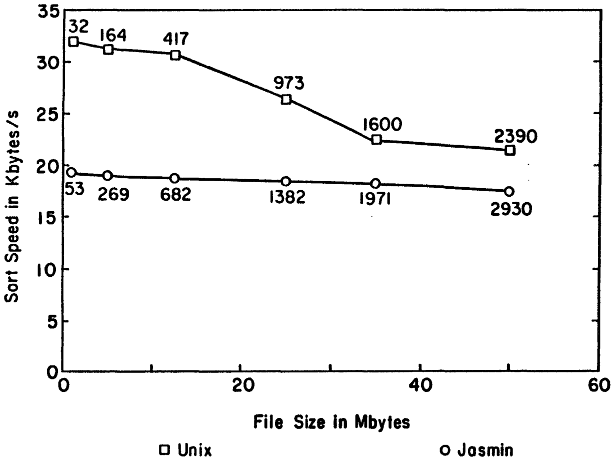


**Figure 6: Serial Sort Times in Seconds on *JASMIN* and UNIX**

### 4.3. Backend Sort

In Table 1, we show the total execution time of the Backend Sort for 4 processor configurations, and for file sizes of 5, 12.5, 25, 50 and 100 megabytes. The times shown correspond to the root's elapsed time, which was only slightly higher than the leaves' or internal nodes'. Figure 7 compares the sorting speed achieved in these experiments to the UNIX-CCI sort.

The most striking observation from this table is how fast the 100M sort is, for both JS3 and JS5. The backend multiprocessor sorts 100M in 1 hour with 3 processors, and in 52 minutes with 5 processors. This performance is comparable to that of highly-tuned commercial sort packages such as the IBM SYNC-SORT on a high-performance mainframe (e.g. an IBM 3081, rated at approximately 7 MIPS). It is thus clear that our parallel Backend Sort provides a very cost-effective alternative to main-frame, serial sorting of large files.

Further analysis of the data in Table 1 leads us to the following observations on parallel speedup:

(1) The 3 processor configuration is faster than the 1 processor configuration by a factor of 1.5 for small files to 1.6 for a 50M file.

(2) Using two additional leaves (i.e. the 5 processor configuration) further improved the sort time by up to 16%.

(3) Using a two-level merge tree with 7 processors does not improve performance. In fact JS7 shows slightly higher overall times than JS5.

| File Size | JS1 | JS3 | JS5 | JS7 |
|---|---|---|---|---|
| 5M | 269 | 178 | 178 | 173 |
| 12.5M | 682 | 439 | 419 | 427 |
| 25M | 1384 | 884 | 836 | 869 |
| 50M | 2930 | 1782 | 1584 | 1682 |
| 100M | —[c] | 3625 | 3131 | 3305 |

**Table 1: Total Sort Times In Seconds 1, 3, 5 and 7 Processor Configurations**

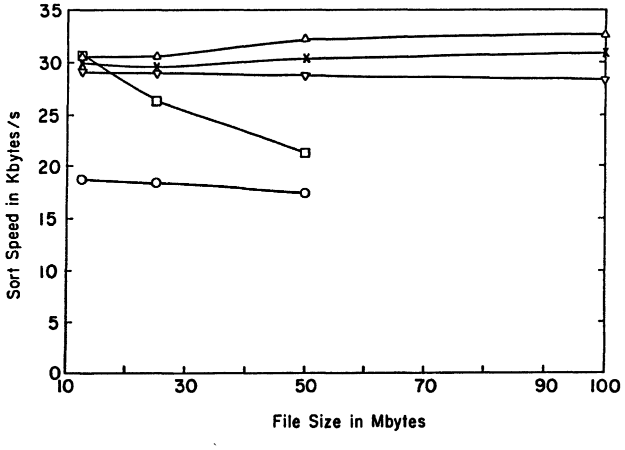


**Figure 7: Parallel and Uniprocessor Sort Speeds**

The added processing power and I/O bandwidth in JS3, with 3 processors and 2 disks, proved very effective in reducing up the elapsed time of the sort. However, as more processors and disks were added, the additional performance enhancement was limited. These observations suggest that the parallel sort is limited by network data transfer rather than by computation or disk activity. We tested this hypothesis by instrumenting the sort program to record idle and elapsed times for both the sort and merge phases. Table 2 shows these measurements for a 25 megabyte sort with 3 processors, and a 50 megabyte sort with 5 processors.

The times in Table 2 are for the root, and one of the leaves. (Since all leaves had similar elapsed and idle times, we only present our measurements for one.) Discrepancies between the elapsed times shown in Table 1 and Table 2 for the same configurations are due to distortions introduced by measuring idle time. The additional data in Table 2 supports our hypothesis in the following ways:

(1) The sort phase of the 50M sort lasts almost twice as long as the sort phase of the 25M sort, although the leaves do exactly the same work in both. The additional time shows up as idle time at the leaves, while they wait for data from the root. In both configurations, the root shows little idle time during the sort phase. I/O delays (shown as *Disk Busy* time) are almost null during the sort phase.

(2) The merge phase of the 50M sort lasts almost exactly twice as long as that of the 25M sort, and substantial idle time accumulates at both the leaves and the root. Unlike the sort phase, where the root processor is saturated, both root and leaves appear to be waiting on the network. Disk waiting during the merge phase accounted for less than 10% of the idle time, in both configurations. Furthermore, it did not decrease with the use of two additional disks (JS5 versus JS3).

(3) The non-idle time at the leaves during the sort and merge phases, is approximately the same on both configurations: sort phase 320 seconds, merge phase 340 seconds. The non-idle time at the root varied linearly with the amount of data in both phases. The sort phase might be speeded up by starting with a distributed file (see Section 4.4), or increasing the power of the root processor. However, it is clear that a higher network transfer rate is necessary in order to speed up the merge phase.

| | | JS3-25M | | JS5-50M | |
|---|---|---|---|---|---|
| | | leaf | root | leaf | root |
| **Sort Phase** | elapsed time | 638 | 639 | 1004 | 1005 |
| | disk busy | 2 | 0 | 0 | 0 |
| | idle time | 320 | 162 | 680 | 26 |
| **Merge Phase** | elapsed time | 937 | 938 | 1871 | 1873 |
| | disk busy | 53 | – | 55 | – |
| | idle time | 598 | 629 | 1534 | 1274 |
| **Total**<br>(Both Phases) | elapsed time | 1575 | 1577 | 2875 | 2878 |
| | disk busy | 55 | – | 55 | – |
| | idle time | 918 | 791 | 2217 | 1300 |

**Table 2: Sort, Merge and Idle Time In Seconds 3 and 5 Processor Backend Sort**

[c] This number is unavailable due to processor memory limitations.

We conclude that for the Backend Sort, the limitation of a low network transfer rate means that the use of more than 4 processors as leaves can speed the sort phase only marginally, and the extra processor power is not used efficiently. Similarly, use of additional processors as interior merge nodes in the 7 processor configuration only reduces non-idle time at the root, which does not improve the overall merge speed. A network which delivered data at the faster rate would be able to increase the utilization of the leaves and root during the merge phase, and so could effectively utilize more processors.

### 4.4. Distributed Sort

The above discussion of Table 2 indicates that the distribution of the file from the root to the leaf processors created a bottleneck. In a *Distributed Sort*, where the file is initially distributed across the local disks at the leaves, the sort phase would be much faster. In fact, since, during this phase, the computation is fully partitioned and every leaf processor has its own disk, the sort phase time at the leaves is simply the uniprocessor sort phase time for one partition. Thus, a good estimate of the sort phase time in the Distributed case is the sort phase time of one partition in the JS1 configuration. Merge phase time in the Distributed Sort is identical to that in the Backend case. By combining these estimates of the sort and merge phase times, we obtained the total times for the Distributed Sort. The results obtained are shown in Table 3.

| | | Sort | Merge | Total |
|---|---|---|---|---|
| **JS3** | 10M | 147 | 158 | 305 |
| | 50M | 738 | 828 | 1566 |
| **JS5** | 10M | 74 | 157 | 231 |
| | 50M | 370 | 696 | 1066 |

**Table 3: Distributed Sort Estimate**
**Sort, Merge and Total Times in Seconds**

While in the Backend Sort, the bottleneck at the root made the use of more than 3 processors inefficient, we now observe a substantial speedup with 5 processors. For a file of 50 megabytes, the elapsed time is 1566 seconds for a distributed 3 processor sort and 1066 seconds for a 5 processor sort. This is a speedup of 1.5 compared to 1.1 for the Backend Sort.

## 5. Conclusions

We have described our experience with developing and evaluating a parallel sort utility. Our goal was to explore the feasibility of fast file sorting algorithms under limited parallelism, limited inter-processor communication bandwidth and the constraints of conventional I/O devices. Our testbed was a multiprocessor that can be configured with up to 12 microprocessors, communicating through a fast packet-switched bus and running a distributed operating system kernel. Multiple disk drives, each attached to one microprocessor, provided high I/O bandwidth.

We implemented a parallel external merge-sort algorithm, and measured its performance for a range of file sizes and processor configurations. After detailed analysis of our measurements, and scaling of our results, we reach a number of conclusions:

(1) Using current off-the-shelf technology, 3 and 5 processor configurations provide a very cost-effective solution to sorting a large file. The 3 processor configuration sorts a 100 megabyte file in 3625 seconds (1 hour), which compares well with commercial sort packages available on high-performance mainframes. The 5 processor sort is 16% faster for large files. This level of performance, at a low hardware cost, results from a design which makes effective use of a streamlined, distributed operating system to exploit a limited amount of parallelism in computation and I/O.

(2) Latency in network data transfer prevents the effective use of higher levels of parallelism in the backend sort model, where the data file is not distributed. Interestingly, the 10 Mbyte/sec physical bandwidth of our network was not the limiting factor: Only 300 Kbyte/sec of that is available to any one processor, and operating system overhead reduces that to approximately 100 Kbyte/sec for the actual process-to-process data transfer rate. At this rate, network latency allows even a microprocessor (rated at 0.6 MIPS) to sort and merge data as quickly as it can be moved between machines. Adding processors to the backend configuration simply increased idle time.

(3) We observed that distributed storage of files provides a substantial benefit in sorting. For instance, with the 5-processor configuration, a 50-megabyte distributed file could be sorted in 17 minutes, compared to 26 minutes needed for a non-distributed file. Our distributed sort model makes efficient use of additional leaf processors to accelerate the sort phase, which the backend model is unable to do.

This study demonstrates that, with current microprocessor and communication technology, parallel sorting of large files is a viable, cost-effective alternative to highly tuned sort utilities on mainframes. Our analysis indicates that its efficiency can be expected to increase dramatically due to improvements in local area network technology in the next few years.

## 6. Acknowledgements

We would like to acknowledge the early work of Bill Burnette and Susan Fontaine at AT&T-Bell Laboratories in developing the JASMIN operating system (then called MAX). U. Premkumar was responsible for developing the network interface which distributes JASMIN across processors. His work with the JASMIN and S/Net software and hardware made this project possible. Hikyu Lee also made significant contributions to the JASMIN S/Net implementation. We thank Ming-Yee Lai for graciously allowing us to disrupt his use of the S/Net, and for helping us with an early draft of this paper.

We are grateful to two managers who have been especially supportive of our work. Dan Fishman saw JASMIN through its early days at Bell Laboratories, and, although now at HP Labs, he continues to encourage our efforts. Mike Lesk, of Bell Communications Research, has generously supported the collaboration with Cornell University which has made this research possible. Finally, we thank Jim Gray from TANDEM Computers for his helpful comments and suggestions.

## References

[1] Anon et al., A Measure of Transaction Processing Power, Tandem TR 85-2, February, 1985.

[2] Ahuja, S.R., S/NET: A High Speed Interconnect for Multiple Computers, IEEE J on Selected Areas in Communications, SAC-1, 5, November, 1983.

[3] Baskett, FH, Howard, JH, Montague, JT, Task Communication in DEMOS. Proceedings of the 6th ACM Symposium on Operating System Principles, November, 1977, pp. 23-31.

[4] Bitton, D, Design, Analysis and Implementation of Parallel External Sorting Algorithms, Ph.D. Thesis, University of Wisconsin-Madison, TR 464, January, 1982.

[5] Bitton, D, and DeWitt D.J., Duplicate Record Elimination in Large Data Files, ACM Trans. on Database Systems, June, 1983, pp.255-265.

[6] Even S., Parallelism in Tape Sorting, Comm. ACM, Vol.17, No. 4, April, 1974, pp.202-204.

[7] Fishman, DH, Lai, MY, Wilkinson, WK, An Overview of the *JASMIN* Database Machine. Proceedings of the ACM SIGMOD Conference, Boston, MA, June, 1984, pp.234-239.

[8] Kitsuregawa, M., Tanaka, H., Moto-oka, T., "Architecture and Performance of Relational Algebra Machine GRACE," Proceedings of the 1984 International Conference on Parallel Processing, 1984.

[9] Kamiya, S., Matsuda, S., Iwata, K., Shibayama, S., Sakai, H., Murakami, K, "A Hardware Pipeline Algorithm for Relational Database Operation," The 12th International Symposium on Computer Architecture, June, 1985.

[10] Kwan S.C., External Sorting: I/O Analysis and Parallel Processing Techniques, Ph.D. Thesis, University of Washington, TR 86-01-01.

[11] Knuth D., *The Art of Computer Programming III, Searching and Sorting,* Addison-Wesley 1973.

[12] Lee, H, Premkumar, U, The Architecture and Implementation of Distributed *JASMIN* Kernel. Bell Communications Research Technical Memo, TM-ARH-000324, October, 1984, Morristown, N.J.

[13] Linderman, JL, Theory and Practice in the Construction of a Working Sort Routine. AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, pp. 1827-1845, October, 1984.

[14] Selinger P.G. et al., Access Path Selection in a Relational Database Management System, Proceedings of SIGMOD, pp. 23-34, May, 1979.

[15] Schweppe, H, Zeidler, H.Ch., Hell, W., Leilich, H.-O., Stiege, G., Teich, W., "RDBM - A Dedicated Multiprocessor System for Database Management," in Advanced Database Machine Architecture, (D.K. Hsiao, ed.), Prentice-Hall, 1983.

[16] Tanaka, Y., "A Data-Stream Database Machine with Large Capacity," in Advanced Database Machine Architecture, (D.K. Hsiao, ed.), Prentice-Hall, 1983.

# A DISTRIBUTED IMPLEMENTATION SCHEME FOR COMMUNICATING PROCESSES

A. A. Aaby and K. T. Narayana

Department of Computer Science
The Pennsylvania State University
University Park, Pa 16802

*Abstract*: We present an efficient time stamp based distributed scheme for synchronization of communicating sequential processes. In the communication model assumed, processes name ports in their communication commands and both the input and output commands can appear in the guards of an alternative command, a general framework than that allowed by Hoare's CSP. We further assume synchronized communication between processes. An implementation of the scheme is described. The scheme can become part of the kernel for a programming language for distributed computation. We provide a discussion on fault tolerance of the algorithm against node failures. We prove the correctness of the scheme by establishing that the scheme is deadlock free and further that if two processes are willing to communicate and don't synchronize with any other, they do synchronize with each other. A specification in interval logic of the scheme and the verification of it, including strong fairness, is given elsewhere.

## 1. Introduction

A distributed system is an interconnection of a network of computing elements with each of the elements having its own local store. The local storage of one computing element is not shared by any other. A distributed program consists of a system of processes with each of the processes executing on some computing element of the distributed system. The processes of a distributed program cooperate so as to achieve a common purpose. Because of the absence of shared storage, the processes achieve cooperation by the exchange of messages. Thus, interprocess communication using message passing constitutes a basic paradigm for the computations evoked by a distributed program.

A model of distributed computation known as Communicating Sequential Processes(CSP) has been proposed by Hoare[8]. A programming notation for the expression of distributed algorithms, constituting a distributed program, has also been suggested in the same work. The model consists of a set of sequential processes, without nesting, with each of the processes confining access to its own data and effecting cooperation between one another by the use of input and output commands for message passing. Processes are explicitly named in the input and output command. Communication between the processes occurs when the output command of a process matches the input command of another process. Such a communication model is termed as synchronous or unbuffered. An alternative command based on Dijkstra's guarded command construct [6] permits an arbitrary selection of a command, for execution, from out of the commands whose guards have been successful. A guard in the command can be a boolean expression or a boolean expression followed by an input command or an input command itself. A boolean guard is said to be successful if it evaluates to true. An input command is successful if a matching output command from the process named in the input command can be found. A guarded input command is said to be successful if the boolean guard preceding the input command evaluates to true and the input command is successful. In an implementation scheme, an arbitrary selection of a command from out of those guards that are successful at the earliest may be made. This strategy assures that a process may synchronize, while in an alternative command, with any of the processes named by it and are ready to synchronize.

However, the introduction of output commands in the guards of an alternative command can simplify the specification of distributed algorithms using the CSP notation[7, 8]. Buckley and Silberschatz[4] suggest a distributed implementation scheme based on assignment of static priorities between processes. They offer four distinct criteria to be met by any implementation scheme. They are as follows:

1) The number of system processes involved in establishing a handshake must be minimal so as to reduce processor interrupts.
2) The amount of system information that each of the processes needs in order to make a decision about synchronization must be minimal.
3) There must be a bound on the number of messages exchanged so as to guarantee that two processes willing to communicate do so within a certain finite time.
4) The number of control signals exchanged between processes must be minimal.

The implementations suggested in [3, 7, 14, 15] violate one or more of these four criteria.

Time stamps have been used for the termination detection of CSP programs[2]. The algorithm given in this paper is based on the use of time stamps and is superior to that given in [12] both in its simplicity and its performance.

The paper is organized as follows. In section 2, a communication model is presented. In section 3, the time stamp based distributed implementation scheme (ANa scheme) is presented. In section 4, an implementation of the algorithm in a programming notation is given. In section 5, various properties of liveness, deadlock freedom, weak fairness and the performance parameters of the scheme are established. In section 6, a discussion is provided on a) the degree of

0190-3918/86/0000/0942 $01.00 © 1986 IEEE

concurrency, b) the comparative performance of the algorithm and c) the resilence of the algorithm against node failures. Section 7 concludes the paper with closing remarks.

## 2. The Communication Model

We assume a port directed communication model. In this model, each process is associated with a finite collection of input ports and output ports through which the messages are sent and received. A process can use an output port only to send messages and an input port only to receive messages. The input port of one process may be connected to the output port of another process. Such interconnection of ports using channels may be specified separately. We require that each port of a process must be connected to one and only one other port and the communication between processes is synchronous. Each process, while in an alternative command, names a collection of ports on which it is willing to communicate. The port based communication paradigm is in contrast to explicit naming of processes in CSP.

To facilitate communication between processes, each process is associated with a port manager. The process, upon entering an alternative command with communication commands in the guards, transmits to its associated port manager the set of ports (called the candidate communicant set) on which it is willing to communicate. The details of transmission of arguments are ignored for reasons of brevity and to concentrate on the implementation scheme establishing synchronization.

## 3. The Algorithm (ANa Scheme)

Whenever a process enters an alternative command, the process requests its port manager to provide the alternative command service by transmitting to the port manager the set of ports named in the alternative command of the process, i.e., the candidate communicant set. If an input command (output command) appears as a statement in the process, then the process utilizes the alternative command service made available by its port manager to achieve a matching output command (input command) on the designated port. Once a process triggers the alternative command service of its port manager, the process waits until the port manager establishes the handshake. Each of the port managers runs the same distributed synchronization algorithm with which it obtains a synchronization partner. Since we assumed a fully distributed system without shared memory, the port managers have to find a synchronization partner by exchanging control signals with other port managers of the ports linked to the candidate communicant set. We now describe, informally, the distributed scheme given in figure 1.

A port manager uses a variable Turn for each of the ports to mark the port to indicate whether it is its own turn to initiate communication on that port or if it is the turn of the port manager of the port linked to that port to initiate communication. Initially, when the distributed program is set up, the port manager of each of the processes marks for each port the variable Turn as MyTurn, meaning that it is MyTurn to contact a partner on that port. A port manager uses a variable Committed for each of the ports to mark the port to indicate that a Commit signal has been received on that port and that a reply has not been sent. A port manager acquires a time stamp for each of the alternative commands it services (the generated time stamps are monotonically increasing).

Assume now that some of the processes in the distributed program have requested the initiation of the alternative command service by their respective port managers. Consider a process P executing an alternative command. The port manager of P picks arbitrarily one of the candidate communicant ports, p, whose state is MyTurn and sends the control signal Commit(p,Tid), where Tid is the time stamp acquired for this alternative command service by the port manager. The port manager then waits for a response. During this period of waiting for a response, two things can happen. A response may arrive on the port p and has to be dealt with. In addition, during the waiting period, some other port manager(s) may send Commit signal(s) in the hope of finding a synchronizing partner in P.

If the response to the Commit signal sent has been a Commit signal, then the port manager regards that as a willingness of another port manager to synchronize and treats that a synchronization partner has been found. It synchronizes and then sets Committed(p) to false and Turn(p) to MyTurn.

If the response to the Commit signal sent has been a No signal, then the port manager regards that the port manager of the correspondent of the port p is at the moment unwilling to synchronize and marks that fact in the variable Turn(p) as ItsTurn, denoting that it is the turn of the correspondent's port manager to seek synchronization on that port.

Suppose that during the period of waiting for a response to a Commit signal sent on the port p, Commit signals arrive on other ports. Then what should the port manager do?

If the port on which a Commit signal has been received is not in the communicant set, then, naturally, the port manager of p cannot synchronize on that port, because it is not a suitable partner. And hence the port manager sends the control signal No as a response and sets Turn(p) to MyTurn.

If the port on which a Commit signal has been received is in the communicant set, then the port manager knows that, if he were not waiting for a response, he could have synchronized with that port manager. It may withhold sending a response.

However, the port manager cannot hold back a response to all of the Commit signals it has received on its candidate communicant set during its waiting period until it itself has received a response to its Commit signal. This is because there is a potential for the creation of a cycle of port managers each of which is waiting for a response, but each of which delays sending a response to every other Commit signal received on the candidate communicant set in that period. Thus there will be deadlock. We seek to break this deadlock by requiring that each port manager send the time stamp that it has acquired along with the Commit signal. Thus, during the period of waiting for a response to a transmitted Commit signal, the port manager of p sends a No signal on those candidate communicant ports on which a Commit signal with a time stamp less than that of its own time stamp is received. In addition, the port state variable Turn is set to MyTurn, denoting that the port manager of p must attempt communication on that port at any future instant. On the other hand, if the time stamp received on the Commit signal is larger than its own time stamp, then the port manager of p delays a response to that Commit signal and marks the variable Committed for that port as true and the port state variable Turn as MyTurn.

If a port manager has failed to synchronize on the chosen port p, then it seeks to send a Commit signal on a port, arbitrarily selected, on which the sending of a response to a Commit signal is pending. Synchronization then occurs on that port, Committed is set to false. The port manager sends the control Signal No on the rest of the ports on which response is delayed, marks the variable Turn for that port as MyTurn and Committed for that port to false, so that future invocations of the alternative command may use that information.

If there are no ports on which a response to a Commit signal is delayed, then the port manager selects a port q other than p and sends a Commit(q,Tid) signal. The port manager waits for a response and takes actions during the period of waiting in the same way as described above.

If it happens that the port manager receives a response of No on each of its candidate communicant ports to a transmitted Commit signal, then the port manager waits. It synchronizes with any port in its candidate communicant set on which it receives a Commit signal the earliest thereafter. If more than one Commit signal is received, then it selects one of them to synchronize and sends a No signal as a response to the others.

If the port manager is not servicing an alternative command, then the port manager sends a No signal as a response to every Commit signal received on any port and marks the Turn of that port as ItsTurn for future use of that information.

## 4. The Implementation of the Algorithm

An informal programming notation has been employed for the description of the implementation scheme. An explanation of some relevant notations is given below.

; denotes the sequential composition of statements.
***[S]** denotes that the statement S is repeated forever.
**S1||S2** denotes a parallel composition of the statements S1 and S2.
**Committed(p)** denotes that a Commit signal has been received on port p and a reply has not been sent.
**CP** denotes the set of candidate communicant ports.
**receive(...)** denotes that the event of the reception of a message has occurred on some port and the reception is asynchronous.
**Select(p)** denotes the procedure of the port manager which selects a port for a communication attempt.
**Selected(p)** denotes that a Commit signal has been sent on port p and either a reply has not yet been received or the transport of the message has not yet occurred.
**send(...)** is the primitive employed by the port manager to transmit a signal on some port and the transmission is asynchronous.
**synchronized** denotes that the port manager has achieved the transport of the message.
**Tid** denotes the current transaction identifier.
**Time** is the clock function of the port manager
**TM(p)** denotes the activity relevant to the transfer of the message datum.
**Turn(p)** denotes which of the predicates, MyTurn or ItsTurn is satisfied on the port p.
**wait(p)** is a command providing for the suspension of the task until the predicate p is true.

The algorithm consists of two loops and is given in figure 1.

The first loop waits for the reception of a Commit signal and upon the reception of a Commit signal certain actions are taken. If the port on which the signal was received is not a port in CP, then the offer to communicate is rejected with a No signal and Turn is set to MyTurn for that port. The case where the port is in CP is more complicated. We require the port manager which sends a Commit signal to wait for a response. This requirement could allow cycle of waiting port managers to form therefore, we prevent the formation of a cycle by a requirement that a port manager while awaiting a response must reject some offers to communicate that are received during the waiting time. The rejection of some offers is dependent on the time stamp received with the offer and the current state of the port manager. If the time stamp received with the signal is less than the time stamp obtained by the port manager in the current alternative command and the port manager is awaiting a response to a Commit signal of its own then a No signal is sent in response. Otherwise Committed is set to true for that port.

```
$\forall p$ Turn(p) := MyTurn;
$\forall p$ Committed(p) := false;
$\forall p$ Selected(p) := false;
CP := $\phi$ ;
BEGIN
  $\forall p$
  *[ wait( receive( Commit(p,u) ));
     Turn(p) := MyTurn
     If $p \notin CP$ then
        send( No(p,Time));
     else if $u > Tid \lor \forall r \in CP \neg Selected(r)$ then
        Committed(p) := true endif
     else if $u < Tid \land \exists\, r \in CP\ Selected(r) \land r \neq p$ then
        send( No(p,Tid)) endif
     endif ]
|| *[ wait( $CP \neq \phi$ );
     Tid := Time;
     synchronized := false;
     repeat
        select(p);
        Selected(p) := true;
        send( Commit(p,Tid));
        if Committed(p) then
           TM(p); synchronized := true;
           Committed(p) := false;
           Turn(p) := MyTurn
        else
           wait( receive( No(p,u)) or
                  receive( Commit(p,u)));
           if receive( Commit(p,u)) then
              TM(p); synchronized := true;
              Turn(p) := MyTurn;
              Committed(p) := false
           else Turn(p) := ItsTurn
           endif
        endif
        Selected(p) := false;
     until synchronized;
     $\forall p \in CP$
     If Committed(p) then
        send( No(p,Tid));
         Committed(p) := false;
         Turn(p) := MyTurn
     endif;
     CP := $\phi$ ]
END
```

Fig 1: Algorithm executed by a port manager.

The second loop executes only when a process is in an alternative command. Upon entering an alternative command Tid is set to a unique value obtained from a local clock which is maintained in loose synchronization with the local clocks of the other port managers in the system[10]. The selection of ports on which to attempt synchronization consists of two phases. The first phase involves active attempts by a port manager to synchronize with a candidate communicant via any port on which it is the port manager's turn to initiate communication (Turn = MyTurn) or on any port on which it has received an invitation to synchronize (Committed(p) = true). Selection of a port for a communication attempt is accomplished in the procedure select(p). (It is intended that priority be given to the port on which a Commit signal was least recently sent and on which Turn = MyTurn.) Next, the signal Commit(p,Tid) is transmitted on the port p and a period of waiting for a response (wait(receive(No(p,u)) or receive(Commit(p,u)))) follows. If the predicate Committed is satisfied on the port p which was selected, then no response is required therefore the transport of the message (TM(p)) occurs. Otherwise, if the response is Commit(p,u), then the transport of the message occurs and if the response is No(p,u), then some other port will be selected for another communication attempt. The second phase is entered when for each candidate communicant port it is the turn of some other port manager to initiate communication (Turn(p)=ItsTurn). the port manager awaits an invitation to synchronize and will synchronize with the first invitation and reject all other invitations received during this phase. In the second phase, the select operation must wait for the reception of a Commit signal on a port in CP.

## 5. Some Properties of the ANa Scheme

The specification and verification of the scheme in [1] establishes the results formally using interval based temporal logic[13].

*Theorem 1:*

*If the predicate function ItsTurn is true on all of the candidate communicant ports of a process which is in an alternative command, then, upon the reception of a Commit signal on a candidate communicant port, the port manager of the process will exit the alternative command eventually.*

*Proof:* The hypothesis corresponds to case 5 of the scheme. Since case 5 leads to case 4, and case 4 is specific to the port manager servicing the alternative command, the port manager will exit the alternative command service eventually.

*Theorem 2: Deadlock freedom of the implementation.*

*If a port manager is committed on some port p, then, eventually, it makes progress on that commitment.*

*Proof:*

*case 1:* Assume that there does not exist a cycle of commitments between port managers. Then, a port manager receives as a response either a Commit signal or a No signal to a Commit(p,id) signal transmitted by it on some candidate communicant port p. The two cases are dictated by case 2a and case 2b of the scheme. Thus the port manager makes progress on its commitment in these two cases.
*case 2:* Assume that there exists a cycle of port managers $P_1,P_2,...P_n$ with transaction identifiers $t_1,t_2,...t_n$ such that, for $1 \leq i \leq n$, $P_i$ has sent a commit signal to $P_{i+1 \bmod n}$ and is awaiting a response. Since the transaction identifiers obtained by each of the port managers in the cycle are totally ordered and responses to Commit signals may be delayed to processes with greater transaction identifiers, there exists a port manager, say $P_k$, with the minimum transaction identifier in the cycle which has sent a commit signal to a port manager $P_{k+1 \bmod n}$. Thus it may be assumed that the transaction identifiers are ordered as follows: $t_1 > t_2 > \cdots > t_k < t_{k+1} > \cdots t_n$ and $t_n > t_1$. The response in $P_{k+1 \bmod n}$ is dictated by case 3c of the scheme. Hence, a No signal is transmitted on a port to $P_k$ and the cycle is broken. Thereafter, progress in commitment is achieved as in case 1.

*Theorem 3:*

*If an alternative command names M ports as candidate communicants, then the signal Commit(p,*$\mathrm{id}_1$*) need be sent on some candidate communicant port p at most once, by the port manager, prior to synchronization with any one of the candidate communicants.*

*Proof:* By step 1 of the scheme a Commit signal may only be transmitted on a candidate communicant port p satisfying the predicate Turn(p) = MyTurn. If on such a port a Commit signal is transmitted and as in case 1 of the scheme and a Commit signal is received on the port p as a response to the Commit signal, then synchronization occurs and the theorem is satisfied. If, however, a No signal is received as a response to the Commit signal then case 2b of the scheme specifies that Turn(p) = ItsTurn is satisfied on that port and the scheme requires that no further signals be transmitted on p until Turn(p) = MyTurn is satisfied. The predicate Turn(p) = MyTurn is satisfied on p under the condition of the reception of a Commit signal on p as in case 2a, case 3a, case 3c or case 6 of the scheme. If on p a Commit signal is received, then, after a possible delay, either synchronization occurs following the transmission of a Commit signal on p as in case 4a or case 5 of the scheme or, if synchronization occurs on some other candidate communicant port then case 4b specifies that a No signal is transmitted on that port. In all of the cases, only one Commit signal is sent and the theorem is satisfied.

*Lemma 1:*

*If* $\mathrm{Cl}_i$ *and* $\mathrm{Cl}_j$ *are, respectively, the communicant lists of two processes* $P_i$ *and* $P_j$*, both of which are in alternative commands with transaction identifiers* $\mathrm{id}_i$ *and* $\mathrm{id}_j$*, respectively, and neither of which has synchronized, and further, the ports* $p_i$ *and* $p_j$ *are linked, where the port* $p_i \in \mathrm{Cl}_i$ *and the port* $p_j \in \mathrm{Cl}_j$*, then ItsTurn(*$p_i$*)→MyTurn(*$p_j$*)*$\wedge$ $\mathrm{id}_j < \mathrm{id}_i$.

*Proof:* By case 2b of the scheme the candidate communicant port $p_i$ satisfies Turn($p_i$) = ItsTurn iff the port manager of the process $P_i$ has received a No signal as a response to a Commit signal on port $p_i$. If the port $p_i$ is linked to the port $p_j$ and the port $p_j$ is a candidate communicant port of the process $P_j$, then the port manager of the process $P_j$, by case 2a, must satisfy Turn($p_j$) = MyTurn on the port $p_j$. Thus the lemma is established.

*Theorem 4: Synchronization Condition*

*If two processes* $P_i$ *and* $P_j$*, each of which is in an alternative command, each name some pairs of ports,* $p_i$ *and* $p_j$*, linking* $P_i$ *and* $P_j$ *as candidate communicants, and further if neither of them synchronizes with any other candidate communicant, then they synchronize with each other on any pair of the named pairs of linked ports* $p_i$ *and* $p_j$.

*Proof:* From Lemma 1, the linked pairs of ports $p_i$ and $p_j$ of the processes $P_i$ and $P_j$ satisfy, ItsTurn($p_i$)→MyTurn($p_j$)$\wedge \mathrm{id}_j < \mathrm{id}_i$, and since it is assumed that there is only a finite number of candidate communicant ports and since, by Theorem 2, the implementation is deadlock free, the port manager of at least one of the processes $P_i$ or $P_j$, say process $P_i$, by case 1, must eventually transmit a Commit signal on one of the linked ports, say $p_i$, to which the port manager of the process $P_j$, by case 4a, must respond with the transmission of a Commit signal on the linked port $p_j$. Thus the theorem is established.

*Theorem 5:*

*When a process is in an alternative command with M number of candidate communicant ports, the port manager of the process need send at most M Commit signals on the candidate communicant ports in order to synchronize.*

*Proof:* From Theorem 3, a Commit signal need be sent at most once on a candidate communicant port prior to synchronization. Since there are M candidate communicant ports, at most M Commit signal need be sent.

*Theorem 6:*

*When a process is in an alternative command with M number of candidate communicant ports, the port manager of the process need send no more than* 2M–1 *Commit or No signals on the candidate communicant ports.*

*Proof:* If, by case 1, the port manager of the process $P_i$, with M number of candidate communicant ports, has sent a Commit signal, on some candidate communicant port $p_i$, and is awaiting a reply to the Commit signal, it may, while awaiting the reply, receive at most M - 1 Commit signals on the other candidate communicant ports $p_j$. The port manager of the process $P_i$ must, by case 2b, send at most M - 1 No signals on the ports $p_j$ on which it received the Commit signals. (If after having sent the No signals the port manager continues to await the reply to the Commit signal sent on the port $p_i$, and receives a second Commit signal on any of the other candidate communicant ports $p_j$, then the port manager will delay response to that Commit signal since by Theorem 3 the second Commit signal received on the port $p_j$ represents a new alternative command transaction and contains a transaction id greater than the transaction obtained by the port manager of the process $P_i$.) If, thereafter, the port manager of the process $P_i$ receives a No signal on the candidate communicant port $p_i$, it may, by case 1 or case 4a of the scheme, send Commit signals, in turn, on each of the M - 1 candidate communicant ports $p_j$ on which No signals have been sent. For each of the Commit signals transmitted by the port manager of the process $P_i$, a No signal may received as a response. In this eventuality case 5 is applicable. That is Commit signals may be received on each of the M candidate communicant ports of the process $P_i$. The port manager then transmits a Commit signal on one of the candidate communicant ports of the process $P_i$. Upon synchronization, and prior to the termination of the alternative command, case 4b requires a No signal be sent, by the port manager of the process $P_i$, on those candidate communicant ports on which a Commit signal is pending. Since there can be at most M - 1 such ports, at most M - 1 No signals are sent after synchronization. Thus, when a process is in an alternative command with M number of candidate communicant ports, no more than 2M - 2 No signals and 1 Commit signal are transmitted by the port manager of that process.

*Theorem 7:*

*When a process is in an alternative command with M number of candidate communicant ports, the port manager of the process need expect no more than 6M - 2 signals on the M candidate communicant ports.*

*Proof:* From the theorems 5 and 6, the port manager need send at most 2M - 1 + M signals. Each of the signals sent is either in response to a Commit signal received or is a Commit signal soliciting a communication. Thus the total number of signal that a port manager need expect on the candidate communicant ports is no more than 6M - 2.

## 6. Discussion

a) Degree of concurrency obtained by the algorithm:

The degree of concurrency achieved by the algorithm is something hard to quantify. However, a comparison of the level of concurrency achieved by the implementation scheme with respect to that achieved by the implementations suggested in the literature can be made by identifying relevant computational steps in each of the algorithms. For such purposes of comparison, the implementation scheme suggested by Buckley and Silberschatz[4], hence forth referred to as BSi scheme, is considered. For purposes of comparison, the BSi scheme is mapped onto the communication model of this paper. The port manager of the process in the ANa (BSi) scheme awaits a response for a Commit (QUERY) signal transmitted by it. If during the period of waiting, the port manager receives a Commit (QUERY) signal on another candidate communicant port, it either delays a response or transmits a No (BUSY) signal, (If a Commit(QUERY) signal is received on a noncommunicant port then a No(NO) signal is sent). Thus each of the schemes are bound to achieve the same degree of concurrency or loss of concurrency there of in their abilities to synchronize. Unlike the ANa scheme, the port manager of a process, in the BSi scheme, transmits a RES signal on all its candidate communicant ports on which it has transmitted a BUSY signal in response to QUERY signals received by it. After the transmission of the RES signal, the port manager awaits RETRY signals on those ports. Of all the RETRY signals received by the port manager, the port manager selects one to synchronize with, by sending a COMMIT signal, and rejects all others by sending a NO signal. In the ANa scheme, on the other hand, if the port manager of a process has received a No signal in response to a Commit signal on any of its candidate communicant ports, then the port manager expects a Commit signal to arrive on that port. When on all of the ports the port manager has received No signals in response to Commit signal sent, the port manager awaits Commit signals on those ports. Then, of all Commit signals received by the port manager, the port manager selects one to synchronize with, by sending a Commit signal, and rejects all others by sending a No signal. Thus the degree of Concurrency in the selection of a port on which to synchronize is the same in both of the schemes.

b) Comparative Performance of the Algorithm

The performance of the algorithm is measured in terms of the number of signals which must be exchanged to achieve synchronization. In the BSi scheme, a process sends out at most 2M unsolicited signals per each execution of an alternative command where M is the number of candidate communicant ports. The unsolicited signals in the BSi scheme correspond to the Commit signals in the ANa scheme. In the ANa scheme the number of unsolicited signals needed is halved. Theorem 5 shows that in the ANa scheme at most M Commit signals need be sent out. Natarajan distinguishes between input ports and output ports. On the assumption that input and output ports are equally likely to occur in an alternative command, Natarajan's scheme is also able to halve the number of unsolicited signals required to achieve synchronization. Natarajan's scheme requires the port manager to initiate communication on output ports while on input ports the scheme requires the port manager to have received a signal prior to attempting synchronization. Thus, Natarajan's scheme sends at most N unsolicited signals on output ports, where N is the number of output ports. If there are N input ports and on those ports a Ready signal had been received and a Reject signal sent and a Not Ready signal has not yet been received, then the port manager may send the equivalent of an unsolicited signal on those ports, thus achieving the M number of unsolicited signals, where M = 2N. In the ANa scheme the total number of signals generated in the system due to the initiation of an alternative command by a process is 3M, where M is the number of candidate communicant ports. In the BSi scheme the total is 3M + Q, where Q is the number of candidate communicants with priority number less then that of the process.

c) Resilence against node failures:

In a distributed system nodes tend to fail. For the distributed program to achieve its goals in spite of node failures, the ANa scheme has to recover from node failures. A simple scheme for effecting such a recovery in the presence of node failures can be imposed on the ANa scheme. It is assumed that, at each node of the distributed system, there exists mechanisms for the detection of the failure and the restart of the node. In addition, it is assumed that when a node fails, the states of the port managers at that node are lost. The communication system is assumed to be reliable and that the messages and signals sent are delivered in the order of transmission and, further, when a message is transmitted, it is either delivered to the destination port or an exception is signaled to indicate the failure of the destination node.

Whenever the node kernel is restarted after the failure of the node, the kernel signals the restart to each of the port managers at that node. Upon the receipt of a restart signal from the node kernel, the port manager at the previously failed node transmits a Restart signal on all of its ports. The port manager then waits for responses to its Restart signal in order to reset it local clock. A port manager which receives a Restart signal on a port satisfies the predicate function MyTurn on that port and transmits on that port the signal, Time(p,t), containing the time, t, indicated on its local clock. The port manager receiving the signal Time(p,t), sets its local clock to the maximum of the Time signals received.

If the communication system detects a failure of the destination node, of a signal or a message, then the port manager which transmitted the signal or message receives from the communication system a failure signal on the port on which the message or signal was sent. The port manager then satisfies the predicate function ItsTurn on that port.

## 7. Conclusions

We have presented a simple and efficient time stamp based synchronization scheme for communicating processes, utilizing output commands as guards in the alternative command. The scheme was developed out of an interest in the utility of temporal interval logic for the specification and synthesis of parallel algorithms. A temporal interval logic based specification of the scheme is presented in [1]. The scheme is port based, utilizes the exchange of control signals to select a suitable communication partner and relies on time stamps to introduce the asymmetry required to avoid deadlock. The selection of a communication partner requires a process to send at least one and at most M signals indicating its willingness to establish communication, where M is the number of ports on which the process is willing to communicate. During the time that a process is attempting to find a communication partner, and prior to the determination of its communication partner, at most M - 1 signals will be sent rejecting pending offers to establish communication. These signals are required to prevent deadlock in the implementation. After the selection of a communication partner and prior to the termination of the alternative command, at most M - 1 signals are required to reject pending offers to establish communication. These are required to insure the progress of other processes in their own selection algorithm. The various properties of deadlock freedom of the implementation and performance of the scheme have been established. A critique of the scheme has been presented in the light of some other schemes appearing in the literature. It has been shown that the scheme is efficient in the number of signals transmitted for achieving synchronization. The degree of concurrency achieved by the scheme is also comparable to that achieved by that of other schemes. The time stamps may be implemented through the use of logical clocks in each process which are kept in loose synchronization using the algorithm due to Lamport[10]. The storage requirements of the scheme are small, a local clock and for each port two bits to indicate the state of the port.

It is hoped that the implementation scheme be imbedded in the runtime system of a compiler which extends OCCAM[9] to include output commands in the guards. Propositional temporal logic has been shown to be suitable for the synthesis of synchronization skeletons for communicating processes[5, 11] and we are exploring the suitability of temporal interval logic for the specification and synthesis of parallel algorithms.

*References*

1. Aaby,A.A. and Narayana,K.T., Specification and verification of a time stamp based distributed synchronization scheme for communicating processes, Computer Science research report CS 85-15, The Pennsylvania State University, University Park, Pa 16802.
2. Apt,K.R. and Jean Luc Richer, Real time clocks versus virtual clocks, in *Control flow and data flow: concepts of distributed programming*, Munich, Germany, August 1984. International Summer School.
3. Bernstein,A.J., Output guards and nondeterminism in Communicating Sequential Processes, *ACM Trans. Prog. Lang. and Systems 2*, 2 (April 1980), pp. 234-238.
4. Buckley,G.N. and Silberschatz,A., An effective implementation for the Generalized Input-Output construct of CSP, *ACM Trans. Prog. Lang. and Systems 5*, 2 (April 1983), pp. 223-235.
5. Clark,E.M. and Emerson,E.A., Using branching time temporal logic to synthesize synchronization skeletons, *Science of computer programming 2*, (1982), pp. 241-266.
6. Dijkstra,E.W., *A Discipline of Programming*, Prentice-Hall, 1976.
7. Francez,N. and Rodeh,M., A distributed abstract data type implemented by a probabilistic communication scheme, *21st Annual Symp. on Foundations of Computer Science*, , 1980, pp. 373-379.
8. Hoare,C.A.R., Communicating Sequential Processes, *Comm. ACM 21*, 8 (August 1978), pp. 666-677.
9. Inmos Ltd, *OCCAM Programming Manual*, Inmos Ltd, 1982.
10. Lamport,L., Time, Clocks, and the ordering of events in a distributed system, *Comm. ACM 21*, 7 (July 1978), pp. 558-565.
11. Manna,Z. and Wolper,P., Synthesis of communicating processes from temporal logic specifications, *ACM Trans. Prog. Lang. and Systems 6*, 1 (January 1984), pp. 68-93.
12. Natarajan,N., A distributed synchronization scheme for communicating processes, *Computer Journal*, . To Appear.
13. Schwartz,R.L., Melliar-Smith,P.M. and Vogt,F.H., An interval logic for higher level temporal reasoning, *ACM PODC*, , 1983.
14. Schwarz,J.S., *Distributed synchronization of communicating processes*, Department of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, July 1978. Technical Report.
15. Van De Snepscheut,J.L.A., Synchronous communication between asynchronous components, *Information Processing Letters 13*, 3 (December 1981), pp. 127-130.

# Intra-transaction concurrency in distributed databases and protocols which use transaction aborts to preserve consistency: a performance study.*

Mohan L. Ahuja**
Bellaire Research Center
Houston, TX 77025

J.C.Browne
U.T.Austin
Austin, TX 78012

*ABSTRACT:* Concern for performance of concurrency control protocols, in distributed database systems, raises an important question 'does performance improve by permitting intra-transaction concurrency'. This paper answers this question for protocols which use abortion of transactions to preserve consistency. Variants of the protocol proposed by Reed [REE78] are simulated and used as a representative of those protocols which abort transactions. The results strongly points out that as the workload increases the intra-transaction concurrency leads to poorer performance. Further, it suggests that all entities should be assigned an order and, whenever access set of a transaction is known a priori, the transaction should access entities in this assigned order with possible skips.

## 1. Introduction.

To improve performance of distributed database systems, we may like to execute each transaction concurrently at multiple sites. Further, if such Intra-transaction concurrency at multiple sites does improve the performance then we would like to increase this improvement by increasing the number of sites: we could stretch this to the extreme by having each entity as a site (i.e. by associating a processor with each entity.) In this paper we look in to the efficacy of permitting Intra-transaction concurrency, particularly in systems which abort transactions to preserve consistency. Some of the protocols which use transaction aborts for preserving consistency are, Optimistic protocol [KUN81], Reed's protocol [REE78], Two-phase locking protocol[1*] [ESW76], the protocol presented in [BAY80], etc. An increase in the workload[2*] will lead to increase in the transaction aborts for each of these protocols which in turn will increase the workload. For sustained workloads higher than a certain level (which is different for each protocol) this cyclic process will lead to a non-linear drop in performance. Effect of different system parameters on such drop is not well understood. The extent and the rate of this drop will vary, but for each protocol, the performance will be unacceptable for workloads higher than a certain workload level. We investigate the effect of intra-transaction concurrency on this level by conducting simulation studies. We note that, it is desirable to have this workload level as high as possible even if it results in a poorer performance for the lower workloads, since poor performance is more affordable at lower workloads than at higher workloads. For the purpose of this study we chose variants of the protocol proposed by Reed [REE78] which permit varying degrees of intra-transaction concurrency.

We found that as the workload increases the performance (in terms of the response time and the throughput) deteriorates at a faster rate for higher levels of Intra-transaction concurrency. In section 2 we summarize the variants of the Reed's protocol. In section 3 we briefly describe the simulated System Model. In section 4 we analyze the results and in section 6 we summarize the conclusions.

## 2. A summary of the Reeds protocol.

First we briefly describe a variant of Reed's [REE78] Protocol to be termed as RP. Then we shall explain how we can incorporate different levels of intra-transaction concurrency in RP.

Multiple versions and tokens are maintained at each entity and are logically ordered in the order of the logical precedence among their creator transactions. A token is a possible version written by a transaction which is still executing. A token may be either converted to a version or deleted, depending upon whether the creator transaction completes or aborts. Each version and token has four fields: first, value of the version; second, the 'start' field, which is the timestamp of the transaction which created the version or the token; third, the 'end' field, which is the timestamp of the transaction which last read the version (in the case of tokens, this is same as the start field); fourth, a boolean 'commit' which is 'true' for a version and 'false' for a token. In addition each token has a pointer to a record of the status of its creator transaction called possibility of the creator transaction.

When a transaction T is initiated, the initiating site assigns it a globally unique timestamp TS and creates its possibility P. Transactions logically execute in the order of their timestamps. P indicates whether T is active, or it has successfully completed or aborted. When T is active then associated with P is a list of messages which are replied when T either completes or aborts; so that the tokens created by T may be either converted to versions or deleted and the transactions waiting to read these tokens may be activated. When all the tokens created by T have been notified about completion or abortion of T then P is deleted.

When T wants to read an entity, it must select a version or a token with start field of the highest value less than TS. If a version is selected, then T reads the version and the value of the end field of the version is set to TS if it is less than TS. However, if a token is selected, then a message is associated with the token to activate T when either the token is converted to a version or is deleted and T waits.[3*] When a message is received indicating that the creator of a token has completed then token becomes a version and any waiting transaction is activated to reads it and to update the end field if required. When a message is received indicating that the creator of a token has aborted then the token is deleted and each waiting transaction is activated to restart the read operation.

When T wants to write an entity, it first selects a version (or token) just as for a read operation. The new token to be written will be placed after this selected version (token). End field of such selected version is then compared with TS. If it is not less than TS then T aborts,[4*] and the associated tokens and P are deleted. Otherwise, the new token can be written without violating consistency. If T wants to read the entity before writing then it does a read operation on the selected version or token. If reading of such token fails then T restarts the write operation otherwise the write operation continues as follows. A new token is created with TS as the start field and is appropriately placed among other versions and tokens of the entity. A message is placed in the list associated with P to notify the site of this entity whenever T completes or aborts.

When T has accessed all the required entities then a message is sent to T's initiation site. When this message is received, messages are sent to entities at which T created any version and T is committed. Old versions are deleted as soon as it is established that no transaction will need to read them in order to maintain

* This work was partially supported by National Science Foundation under Grant MCS-9214613.

** This work was done while the Author was at U.T.Austin.

[1*] In context of Two-phase locking protocol, we use the term 'abort' to mean releasing of the locks granted to a transaction during the locking phase, when a deadlock is detected.

[2*] In this paper we do not define workload quantitatively and refer to a change in the workload in terms of a change in the parameter values. Typical reasons for an increase in the workload are, a reduction is inter-arrival time between transactions, an increase in the average number of entities accessed by transactions, an increase in the extent of data sharing etc.

[3*] In [REE78], the proposed procedure is a little different.

[4*] A transaction with timestamp higher than TS has read the selected version after which the new token will be placed.

0190-3918/86/0000/0950 $01.00 © 1986 IEEE

consistency. This simulates a system with the least number of versions, without resulting in abortion of read transactions. This procedure is different, and deletes old versions sooner, then the procedure proposed in [REE78].

[REE78] does not require or suggest any particular order in which a transaction may access entities. We considered three options RP1, RP2, and RP3, as shown n Figure-1. These options permit different levels of intra-transaction concurrency. In RP1, a transaction executes sequentially (i.e. no intra-transaction concurrency) and it accesses entities in an order pre-assigned to all entities.[5*] This option requires that each transaction be able to incrementally declare its access subsets from the successive parts of the pre-ordered entities. In RP2, at each site, transactions access entities just as in option 1, and they may concurrently execute at all the sites to be accessed. This option permits partial intra-transaction concurrency, and requires that each transaction be able to incrementally declare its access subsets from the successive parts of the pre-ordered entities on each site . In RP3, a transaction concurrently attempts to access all the required entities. This option permits the highest possible intra-transaction concurrency.

Nature of a transaction may impose some constraints on the order in which entities should be accessed. A transaction may not be able to determine whether to read/write an entity $e_x$, or may not be able to determine the value to be written until it reads some entity $e_y$ after $e_x$ in the pre-assigned order. In all simulated options such writes are done after $e_y$ has been read. However, whenever possible such reads are done in anticipation (i.e. before reading $e_y$.)

## 3. The System Model.

In the process of describing the system model we shall define parameters and give, in parentheses, the values assigned to the parameter in the simulation runs. The simulated system has S (10) sites. There are a total of M (1000) entities in the database system, and each site has M/S (100) number of entities. There is no data replication. We presume a network topology in which the communication delays between any two sites have the same mean and the same distribution. Communication delay between any two sites either is zero or follows an exponential distribution with a mean DEL (100 time units) over the entire range of communication traffic generated in the system. The communication network of the system never fails. Transactions are of two types: read transactions and write transactions. Read transactions only read. Write transactions write all of the entities to be accessed; and, before writing an entity, they read it. A write operation may have a constraint such that before writing $e_x$ the $N^{th}$ ($5^{th}$) entity to be accessed after $e_x$ must be read. Each write transaction may have L (1) such constraints. R/W (3, 15) is the ratio of the number of read and write transactions entering the system. Inter-arrival time between transactions follows an exponential distribution with a mean of ARRL (1, 10, 20, 100 time units). Each transaction accesses SN (3) number of sites and an equal number of entities at each site. Transaction size, or number of entities accessed by a transaction, follows a geometric distribution with mean transaction size TZ (3, 6, 15, 100). Each read, write or a comparison takes 1 time unit.

When a new transaction is initiated, it is assigned a transaction type and an initiation site. Then sites to be accessed by the transaction are selected. First site is selected with uniform probability from the $1^{st}$ through the $(S-SN)^{th}$ sites. Let $k^{th}$ site be the last selected site when SM sites have been selected. The next site is chosen from $(k+1)^{th}$ to $(S-SN-SM)^{th}$ sites, such that the distance between two consecutively selected sites[6*] follows a geometric distribution with mean Dis-s (5.0, 1.66, 1.0). This process is repeated until all SN sites to be accessed have been selected or until n sites are yet to be selected and only n more sites remain to choose from, in which case all the remaining n sites are selected. Process for selection of entities is similar to the process for selection of sites except that the mean distance between two consecutively selected entities is Dis-e (5.0, 1.66, 1.0).

## 4. Results and the analysis.

The simulator was validated by validating various events and the results for extreme cases. The results were collected for 2000 (in few cases 1000) completed transactions. The data was collected after completion of 500 transactions. Since RP3 is a special case of RP2 (in which each entity is a different site) we planned to run simulations for RP3 only if RP2 performed better than RP1 for an interesting range of parameter values. As the results will show we did not have to run simulations for RP3. The average response times reported here refer to the response times of completed transactions. Hence, under heavy workloads (when many transactions have not completed), the reported average response times are less than the actual average response times. Each such occurrence has been marked with an asterisk (*) alongside the average response times.

[AHU84] reports results for a wide range of parameters and concludes that it is sufficient to study results for two extremes of dispersal of accessed sites and entities, which are termed high and low. A combination of Dis-s=5.0 and Dis-e=5.0 represents a high dispersal and combined effect of Dis-s=1.0 and Dis-e=1.0 represents low dispersal. Comparison of the performance of RP1 and RP2 based on the results reported in [AHU84] can be typically summarized by the selected results presented in Table-1. These results correspond to DEL=100, TZ=15, SN=3, R/W=3, for low and high dispersal.

We observe that at low workloads (i.e. for ARRL=100), the average response times are lower for RP2 compared to RP1. With respect to other parameters also, RP2 performs marginally better than RP1. The primary reason for this is the lower communication delay for RP2 during execution of a transaction.

As the workload increases, however, the response time of transactions and the rate of abortion of write transactions increase (as shown by an increase in the number of times a write transaction had to be submitted.) The aborted transactions result in the abortion of other transactions, and the abortion rate increases in a cyclic process. We are interested in finding out if this process occurs at substantially different workload levels for RP1 and RP2. In the following discussion, we shall cite a reduction in ARRL as the reason for an increase in the workload. However, the observations made here are true for an increase in the workload due to other reasons (e.g., an increase in TZ or dispersal.)

We observe that for high ARRL (e.g., 100) the abortion rate of transactions is marginally lower for RP2 than for RP1. With a reduction in ARRL, however, the abortion rate increases much more sharply for RP2 than for RP1 and for RP2 it quickly becomes extremely high. In Figure-2 we have plotted the average number of times a write transaction had to be submitted for both RP1 and RP2 against ARRL, for low and high dispersals. These plots support the above observations and they show the magnitude by which the abortion rate and the rate of its increase are higher for RP2 compared to RP1. The reason for this behavior is that above a certain workload level, when a transaction aborts, the increase in the probability of aborting other transactions is higher for RP2 than for RP1. At first it is due to the higher average number of entities accessed by aborted transactions[7*] and at yet higher workloads it is due to higher abortion rate in RP2.

[5*] An alternative to RP1 which does not permit any intra-transaction concurrency could be that each transaction executes sequentially and accesses entities in a random order. Few simulation runs strongly confirmed that this option does not perform as well as RP1 (due to a transaction visiting a site more than once and due to higher abortion rate, both due to transactions accessing entities in a random order.) So we eliminated this option from further detailed study.

[6*] Distance between two neighboring sites (entities) in the order of sites (entities) is considered to be one.

[7*] At low workloads, on average an aborted transaction will have accessed more number entities in RP2 than in RP1. For RP1, when a transaction is aborted it will have read entities before (but not after) the entity at which it could not write. For RP2, however, when a transaction is aborted it might have accessed the entities both before and after the entity which it could not write.

At low workloads, higher number of entities accessed by aborted transactions in RP2 leads to higher abortion rate, due to the following reason. The value of the end field of the version read by an aborted transaction $T_i$ will be equal to (or more than) the timestamp of $T_i$. This may lead to abortion of another transaction with a timestamp lower than the timestamp of $T_i$ (while logically, $T_i$ did not read the version, since it was aborted). Hence, for RP2 the cyclic process leading to higher abortion rates occurs at lower workloads and at a faster rate. For example, let us consider an increase in the dispersal of accessed sites and entities from low to high, for ARRL=10. For RP2 the abortion rate increased 7.8 times (from 22.28 to 179.0) as compared to a corresponding increase of 2.7 times (from 4.47 to 12.04) for RP1.

As the workload increases, the cyclic process leads to more frequent abortion of transactions in the earlier stages of execution: and this happens at higher rate for RP2. At high workloads this abortion of transactions at earlier stages can be confirmed for RP2 by the results in Tables-1. Consider the example of an increase in the dispersal of accessed sites and entities from low to high for ARRL=10. The average number of entities accessed by an aborted transaction reduced by approximately a factor of 1.95.[8*]

We conclude that at low workloads average number of entities accessed by an aborted transaction is higher for RP2 compared to RP1. This leads to the cyclic increase in the abortion rate at much lower workloads for RP2. This conclusion can be extrapolated to compare RP1 with RP3. If a transaction accesses TS entities, then RP3 can be viewed as RP2 with SN=TS. For SN=TS the analyses given above for comparing RP1 and RP2 are likely to be all the more valid. Hence, it is advisable to choose RP1 compared to RP2 and RP3, and it is advisable to not permit intra-transaction concurrency. This leads to chosing a marginally higher cost at lower workloads (when the higher cost is affordable) in favor of the savings at higher workloads. It is a wiser choice, since the need for a well performing protocol is more critical at higher workloads. Just as in RP, in other protocols which permit transaction aborts, an increase in the workload increases the abortion rate. Also the factors which increased this rate for RP when intra-transaction concurrency is permitted also occur in these protocols. Hence by the same reasoning, for these protocols also it is advisable to not permit intra-transaction concurrency.

## 5. Conclusions.

We conclude that permitting intra-transaction concurrency is not an effective way to improve performance, if transactions are aborted to preserve consistency. Also whenever possible transactions should access entities in some pre-assigned order. This improves the overall system performance at high workloads. If such ordering cannot be implemented over the entire database then it may be done over a part.

**COMPARISION OF SELECTED RESULTS FOR RP1 & RP2**

| ARRL | PROT-OCOL | 10 | | 20 | | 100 | |
|---|---|---|---|---|---|---|---|
| EXTENT OF DISPERSAL OF ENTITES AND SITES ACCESSED BY TRANSACTIONS | | HIGH | LOW | HIGH | LOW | HIGH | LOW |
| AVERAGE RESPONSE TIME OF WRITE TRANSACTIONS | RP1 | 866 | 690 | 830 | 494 | 456 | 436 |
| | RP2 | 489 | 442 | 489 | 337 | 304 | 288 |
| AVERAGE RESPONSE TIME OF READ TRANSACTIONS | RP1 | 1598* | 1456 | 2112* | 732 | 502 | 477 |
| | RP2 | 5004* | 888* | 1654* | 436 | 334 | 326 |
| THROUGHPUT OF WRITE TRANSACTIONS: AS PERCENTAGE OF WRITE TRANSACTIONS SUCCESSFULLY COMPLETED | RP1 | 58.6 | 77.0 | 77.6 | 97.2 | 98.7 | 99.0 |
| | RP2 | 32.1 | 68.0 | 45.5 | 100 | 100 | 100 |
| PERCENTAGE OF TOKENS CONVERTED TO VERSIONS | RP1 | 18.8 | 28.9 | 21.3 | 53.6 | 71.9 | 88.6 |
| | RP2 | 1.2 | 4.8 | 1.1 | 62.7 | 84.0 | 89.0 |
| AVERAGE NUMBER OF VERSIONS PER ENTITY | RP1 | 1.78 | 1.76 | 1.57 | 1.25 | 1.03 | 1.02 |
| | RP2 | 3.63 | 2.71 | 3.18 | 1.20 | 1.02 | 1.02 |
| AVERAGE NUMBER OF FIELDS PER ENTITY | RP1 | 5.34 | 5.28 | 4.71 | 3.75 | 3.09 | 3.09 |
| | RP2 | 10.89 | 8.13 | 9.54 | 3.60 | 3.06 | 3.06 |
| AVERAGE NUMBER OF TIMES A WRITE TRANSACTION HAD TO BE SUBMITTED FOR SUCCESSFUL COMPLETION | RP1 | 12.04 | 4.47 | 7.66 | 1.65 | 1.17 | 1.09 |
| | RP2 | 179.02 | 22.28 | 149.92 | 1.40 | 1.10 | 1.07 |
| AVERAGE TIME A TRANSACTION WAITS FOR READING TOKENS | RP1 | 526 | 346 | 484 | 158 | 34 | 14 |
| | RP2 | 211 | 155 | 217 | 47 | 19 | 7 |

PARAMETERS: DEL = 100, TZ = 15, SN = 3, R/N = 3

AN ASTERIK (*) INDICATES THAT THE REPORTED AVERAGE RESPONSE TIME IS SMALLER THAN THE ACTUAL AVERAGE RESPONSE TIME, DUE TO UNCOMPLETED TRANSACTIONS.

TABLE-1

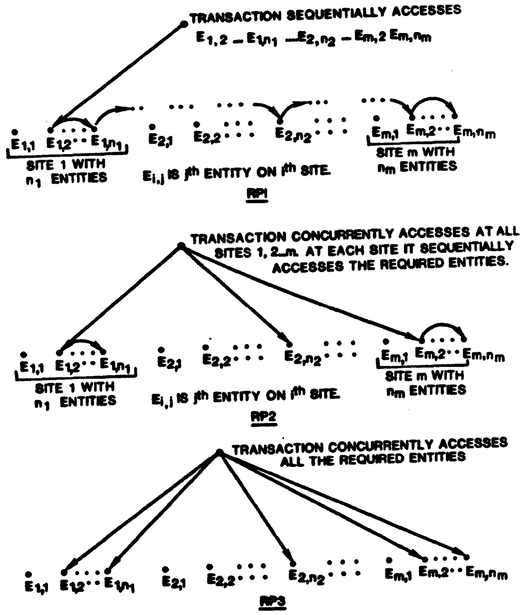

TRANSACTION SEQUENTIALLY ACCESSES
$E_{1,2} - E_{1,n_1} - E_{2,n_2} - E_{m,2}\ E_{m,n_m}$
$E_{1,1}$ $E_{1,2} \cdots E_{1,n_1}$
SITE 1 WITH $n_1$ ENTITIES
$E_{2,1}$ $E_{2,2}$ $E_{2,n_2}$
$E_{i,j}$ IS $j^{th}$ ENTITY ON $i^{th}$ SITE.
$E_{m,1}$ $E_{m,2} \cdots E_{m,n_m}$
SITE m WITH $n_m$ ENTITIES
RPI
TRANSACTION CONCURRENTLY ACCESSES AT ALL SITES 1, 2..m. AT EACH SITE IT SEQUENTIALLY ACCESSES THE REQUIRED ENTITIES.
RP2
TRANSACTION CONCURRENTLY ACCESSES ALL THE REQUIRED ENTITIES
RP3

FIGURE-1

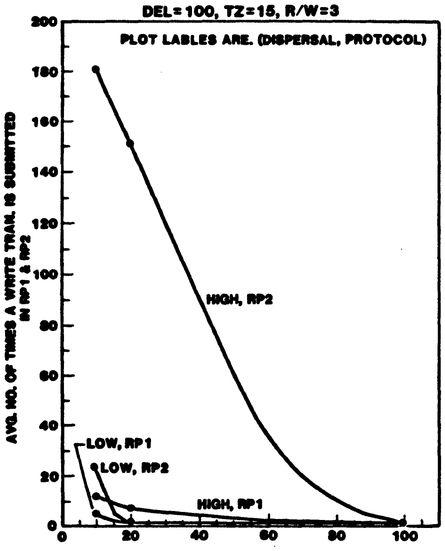


FIGURE-2

## References.

[AHU84] Ahuja Mohan L.,"Concurrency and consistency in distributed database systems; development and evaluation of protocols", Ph.D. dissertation, U.T.Austin, '84.

[BAY80] Bayer, R., Elhardt, Klaus, Heller, H. and Reiser, R., "Distributed Concurrency Control in Database Systems," Proc. VLDB, '80.

[ESW76] Eswaran K.P., Gray J.N., Lorie R.A., Traiger I.L.,"The Notions of Consistency and Predicate Locks in a Database System.", Communications of ACM, Nov. '76.

[KUN81] Kung, H. T. and Robinson, J. T., "Optimistic Methods for Concurrency Control," ACM Trans. on Database Systems, 6, '81.

[REE78] Reed, D. P., "Naming and Synchronization in Decentralized Computer Systems," Technical Report MIT/LCS/TR205, Sep. '78.

[8*] For an increase in the number of times a transaction had to be submitted for successful completion by approximately a factor of 7.8 (from 22.28 to 179.0) the tokens created by the aborted transaction reduced by approximately a factor of 4 (from 4.8 to 1.2).

# TASK ALLOCATION PROBLEMS IN DISTRIBUTED COMPUTER SYSTEMS

Pauline Markenscoff and Weikuo Liaw

Department of Electrical Engineering
University of Houston - University Park
Houston, Texas 77004

**ABSTRACT** -- Task allocation problems for a class of distributed systems are studied. The subtasks comprising the computational task should satisfy certain precedence constraints and the minimization of the system response time is the criterion used to determine the optimal allocation strategy. This task allocation problem is NP-complete. An optimal branch and bound algorithm as well as two approximate algorithms based on the greedy and local search approaches are developed for its solution. Finally, a comparison of the algorithms is presented.

## INTRODUCTION

An important design problem of distributed systems is the allocation of the subtasks comprising the computational task to the individual processors. Task allocation for distributed systems has been studied in [1-6]. These studies employed the maximization of the system throughput or the minimization of the interprocessor communication cost as the criterion to determine the optimal task allocation.

The requirement of fast real-time response, however, necessitates the use of a different performance criterion. The optimal task allocation must now be determined by minimizing the system response time. This criterion was used by Stone [7] to study the allocation problem for a task with no parallelism and a nonhomogeneous system of processors.

The allocation problem for a task that exhibits parallelism is considered in this study. This task is modeled as a directed acyclic graph (DAG), whose nodes denote the subtasks and whose arcs indicate precedence constraints that must be satisfied during the execution of the subtasks. The optimization problem considered corresponds to the partitioning of the nodes of the graph in such a way as to minimize the system response time.

If the execution of two subtasks must satisfy a precedence constraint and these subtasks are allocated on different processors, the constraint can be satisfied via interprocessor communication. However, communication between processors may incur significant overhead. In that case, a system not allowing for interprocessor communication may provide faster response times despite the fact that it introduces the need for redundancy of subtask executions. The system without interprocessor communication is also interesting for another reason. Communication links between processors may be among the least reliable parts of the system and communications between processors may collapse due to hardware or software faults or to externally inflicted damage. Therefore, in many real-time applications with high reliability requirements (e.g. critical control and navigation computers), the multiple processor system should be able to operate in an optimal way without interprocessor communication links.

Optimal and approximate algorithms are developed for the solution of the task allocation problem without interprocessor communication and their performance is evaluated.

## PROBLEM DEFINITION

The computational task considered consists of subtasks whose execution must satisfy certain precedence constraints. Such a task can be modeled as a directed acyclic graph (DAG) (see Fig. 1). The nodes of this DAG correspond to the subtasks and the arcs indicate precedence relationships for the execution of the subtasks. Arcs emanating from node i and ending at nodes j1, j2, ...., jk, for example, indicate that subtask i must wait for data from the execution of all subtasks j1, j2, ...., jk before it can be processed.

The problem is now to schedule the subtasks on m identical processors subject to the precedence constraints indicated by the DAG. It is assumed that there is no interprocessor communication and consequently there is no data flow between subtasks allocated on different processors.

Since no interprocessor communication is allowed, if a node is allocated to a processor then all of its immediate descendants should also be allocated to the same processor. For the example of

0190-3918/86/0000/0953 $01.00 © 1986 IEEE

Fig. 1, if node 5 is allocated to the i-th processor, then nodes 4, 2, and 3 must also be allocated to the i-th processor because node 5 requires data from nodes 2, 3, and 4. A root of a DAG corresponds to an independent process which includes all nodes that can be reached from that root. All the nodes of an independent process must be allocated to the same processor.

Clearly, the problem of task allocation corresponds to the allocation of independent processes derived from a DAG. For example, the DAG shown in Fig. 1 has three independent processes each corresponding to a root and defined by a subgraph of the DAG (or subdag) as follows.

**Process 1** ($p_1$) : Subdag 1 containing nodes 8, 6, 4 and 1.

**Process 2** ($p_2$) : Subdag 2 containing nodes 9, 6, 4 and 1.

**Process 3** ($p_3$) : Subdag 3 containing nodes 10, 7, 6, 5, 4, 3, 2 and 1.

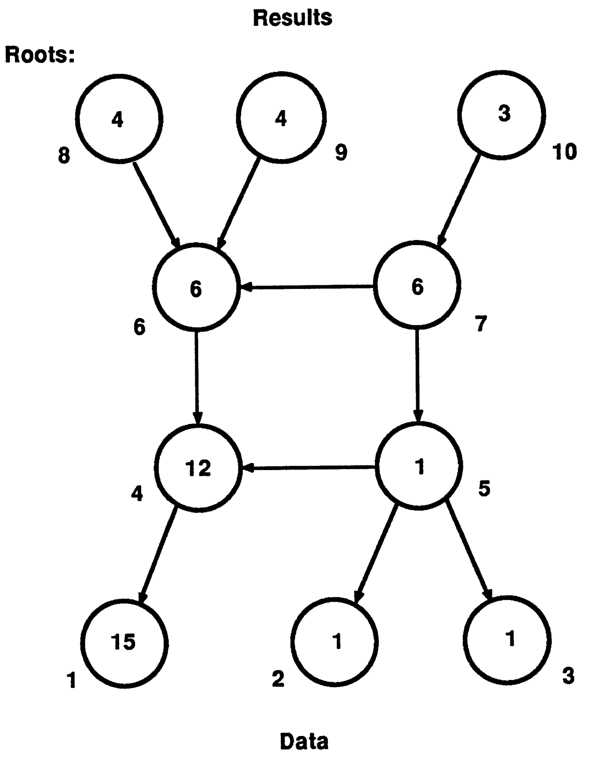


Figure 1: DAG of a computational task. The numbers inside the nodes represent the weights (execution times) associated with each subtask.

The combinatorial problem is now introduced when the number of independent processes in a DAG is greater than the number of processors. Consider the DAG of Fig. 1 and a two-processor system. There exist three possible root partitionings of the DAG of Fig. 1 leading in general to six possible allocations of the independent processes to the two processors. If the processors are identical, it is obvious that, for example, the allocation ($p_1$, $p_2$) to processor 1 and $p_3$ to processor 2 is equivalent to the allocation $p_3$ to processor 1 and ($p_2$, $p_3$) to processor 2. Thus, the number of possible allocations is reduced to three and these allocations are shown in Table 1.

A specific weight equal to the execution time $E_i$ of the subtask is associated with each node. Each independent process associated with a root partitioning has a total weight that is equal to the sum of the weights of the nodes it contains. Obviously, the process with the maximum weight determines the system response time.

In order to optimize the system response time, the following optimization problem must be solved: **"Minimize the system response time over all possible process allocations"**.

Table 1 shows all the possible allocations and the corresponding system response times for the example of Fig. 1. Note that while a subdag is allocated to only one processor, a subtask may be allocated to more than one processor if it belongs to more than one subdag. Thus redundancy is introduced whose amount can be significantly affected by the choice of process allocation. In our example, nodes 1, 4 and 6 are executed twice.

An investigation [8] of the computational complexity of the task allocation problem described above has shown that it is NP-complete [9].

**TABLE 1**

**Independent Processes and System Response Times for a two processor system.**

Three independent processes derived from Fig. 1
$p_1$ has nodes 8, 6, 4 and 1
$p_2$ has nodes 9, 6, 4 and 1
$p_3$ has nodes 10, 7, 6, 5, 4, 3, 2 and 1
Three possible allocations on two processors

| | TASK ALLOCATION | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| Processor 1<br>Processor 2 | $p_1,p_2$<br>$p_3$ | $p_1,p_3$<br>$p_2$ | $p_2,p_3$<br>$p_1$ |
| Response Time | 45 | 49 | 49 |

## ALGORITHM DESIGN AND ANALYSIS

Three types of algorithms have been developed for the solution of our task allocation problem : an optimal Branch-and-Bound algorithm and suboptimal algorithms based on the Greedy and Local Search approaches. The notation used in the presentation of algorithms is defined in Table 2.

### Optimal Algorithm

This algorithm is implemented using the branch-and-bound method. It finds the optimal solution for an instance but its run-time cannot be bounded by a polynomial of the input size of the instance. However, it finds the optimal solution in a tolerable time for instances of small input size.

The solution space is represented by a tree and a feasible point is represented by a node in the tree. Here and in the following discussion, **the term node applies to the tree of the solution space and not to a DAG.** For each node in the tree, a child is generated for every possible allocation of the next subdag. Thus, the number of branches from a node is equal to the number of possible allocations of the next subdag. A node in the tree is called a **solution node** if all subdags have been allocated. Otherwise, it is called a **branch node.**

The following procedure is used to "kill" nodes (i.e. not to generate children of this node) and thus reduce the computation time required for the search. An initial upper bound UB of the optimal solution is chosen arbitrarily. As the algorithm progresses, it is updated to the finish time of a solution node. A lower bound LB(X) of the optimal solution is computed for every node X and compared to the upper bound. If either one of the following two conditions holds

(a) LB(X) > UB, or

(b) LB(X) = UB and a solution exists

then the node is killed. If a branch node is killed, then all nodes derived from it will have lower bounds that are greater than the UB. Thus, there is no need to search for such nodes and the computation time is greatly reduced.

The lower bound LB(X) of a node is computed as follows for our problem. Let LBSPT(X) be the lower bound of the sum of processing times from node X after a task is completely allocated. LBSPT(X) is equal to the sum of the current processing times plus the sum of execution times of unallocated subtasks. Then

$$LB(X) = \max \{ \lceil LBSPT(X)/M \rceil ,\ PT_i,\ i=1,2,\ldots,M \}$$

**TABLE 2**

**Notation**

| | |
|---|---|
| N | : Number of subtasks in a task |
| M | : Number of processors |
| $N_s$ | : Number of independent subdags in a task |
| $S_i$ | : The set of subtasks of subdag i |
| $E_i$ | : Execution time of subtask i |
| T(S) | : Sum of execution times of subtasks in set S |
| $P_i$ | : Set of subtasks allocated to processor i |
| $PT_i$ | : Processing time for processor i, $T(P_i)$ |
| FT | : Finish time of task, max $PT_i$, i=1,2,....,M |

### Greedy Algorithms

When a task is completely allocated, the sum of processing times SPT is given by

$$SPT = \sum_{i=1}^{M} PT_i = \sum_{i=1}^{N} E_i + TRT \qquad (1)$$

where TRT is the total repeated execution time caused by redundancy, i.e. the allocation of some subtasks to more than one processors. The following inequality is derived for the finish time FT

$$FT \geq \lceil SPT/M \rceil \qquad (2)$$

Redundancy is minimized when TRT is made as small as possible. Also, better load balancing among the processors is achieved when FT is close to $\lceil SPT/M \rceil$. Note, however, that FT is not necessarily minimized when both TRT and ( FT - $\lceil SPT/M \rceil$ ) are at their minimum.

Three approximate algorithms (GR_1, GR_2 and GR_3) based on the greedy approach have been developed [8]. The greedy algorithms proceed stage by stage making the best decision at each stage [10]. In our case, a stage is the allocation of a subdag. GR_1 attempts to minimize the maximum processing time at each stage by balancing the computational load of processors. Algorithm GR_2 attempts both to balance the computational load of processors and to reduce the repeated execution time due to redundant execution of subtasks. Algorithm GR_3 is a refinement of GR-2 [8].

The general version of the greedy approach for our task allocation problem is shown below.

```
Greedy Approximate Algorithms

Input:  N, $N_s$, $S_i$, i=1,2,...,$N_s$, $E_j$, j=1,2,...,N, M

Output:  $P_i$, i=1,2,...,M, FT

Method:

(1) Call the function SORT(subdags) to sort the
    subdags in order of nonincreasing execution
    times.

(2) Remove the first subdag $S_\ell$ from the sorted
    sequence.

(3) j ← SELECT($P_i$, i=1,2,...,M, $S_\ell$).
    The function SELECT chooses a processor j for
    the subdag $S_\ell$ according to the rules discussed
    in this section.

(4) Allocate subdag $S_\ell$ to processor j and update
    $P_j$ to $P_j + S_\ell$.

(5) Repeat steps (2) through (4) until all subdags are
    allocated.
```

Algorithms GR_1, GR_2 and GR_3 employ different SELECT functions. Only the SELECT function for algorithm GR_1 is presented below and detailed descriptions of the corresponding SELECT functions for GR_2 and GR_3 may be found in [8].

```
SELECT_1:

Input:  $P_i$, i = 1, 2, ..., M, $S_\ell$

Output:  j

Method:

(1) Compute $T(P_i + S_\ell)$ for all used processors and
    one unused processor.

(2) Select processor j whose $T(P_i + S_\ell)$ is the
    minimum among those computed in step (1).

(3) Return.
```

## Local Search Algorithms

Local search is based on a trial and error optimization method [11]. Starting from some feasible solution x, a neighborhood NBH(x) of x is searched for a better solution. This neighborhood NBH(x) consists of points that are "close" in some sense to the point x. The general local search algorithm is shown below.

```
LOCAL SEARCH ALGORITHM

Input:  The starting solution t.

Output:  The local optimal solution x.

Method:

(1) x ← t.

(2) Repeat IMPROVE(x) until IMPROVE(x) is false.

(3) Output x since "no further improvement is
    possible".
```

IMPROVE(x) searches for a better solution in a neighborhood of x and is defined by:

$$\text{IMPROVE}(x) = \begin{cases} \text{TRUE and } x \leftarrow y, \text{ if there exists} \\ \text{a } y \text{ in NBH}(x) \text{ with } c(y) < c(x). \\ \\ \text{FALSE and a better solution} \\ \text{does not exist in NBH}(x). \end{cases}$$

The function c(x) above is the cost function for the optimization problem. If an improved solution exists, we adopt it and repeat the neighborhood search from the new solution. The search stops when a local optimum is reached.

This approach is now applied to our particular task allocation problem whose cost function is the finish time FT. IMPROVE(x) searches a neighborhood of x for a solution yielding smaller FT than that of x. Two schemes are used to reduce the finish time of a task and they are similar to those discussed in the previous section.

(1) Balance the computational load of processors by exchanging subdags among them. This yields processing times that are close to one another, thus reducing the finish time FT.

(2) Reduce the repeated execution time of shared subtasks by combining subdags that have repeated subtasks.

These exchange and combination schemes are applied to a pair of processors P1 and P2 at a time and a skeleton code of the functions that implement them follows.

```
EXCHANGE1 (P1, P2, TEST): BOOLEAN

(1) Try exchanging one subdag of processor P1 with
    one subdag of processor P2 or moving one
    subdag from processor P1 to processor P2.

(2) If the resulting FT is less than TEST, then return
    TRUE.

(3) If no further exchange is possible, then return
    FALSE. Else go back to step (1).
```

```
EXCHANGE2 (P1, P2, TEST): BOOLEAN

(1) Exchange two subdags of processor P1 with one
    subdag of processor P2 or one subdag of
    processor P1 with two subdags of processor P2.

(2) If the resulting FT is less than TEST, then
    return TRUE.

(3) If no further exchange is possible, then return
    FALSE. Else go back to step (1).
```

```
COMBINE (P1, P2, TEST): BOOLEAN

(1) Find all subtasks shared by P1 and P2.

(2) If there are no shared subtasks, then return
    FALSE.

(3) For a shared subtask, move all subdags
    containing it from P1 to P2. All subdags of P2
    that do not contain the shared subtask are
    considered for allocation to either P1 or P2
    according to a GR_2 scheme.

(4) If the resulting FT is less than TEST, then return
    TRUE.

(5) If there are no more shared subtasks, then return
    FALSE. Else go back to step (3).
```

These three functions form the basis for the two local search algorithms developed. The first algorithm LS_1 starts with the initial guess t and attempts to find a better solution according to the following procedure.

(1) Select the processor with the longest processing time and name it P1.
(2) Find the processor with the shortest processing time and name it P2.
(3) Try to exchange subdags between P1 and P2 in order to obtain a shorter finish time FT.
(4) If this search (performed by EXCHANGE1) is not successful, pick the processor with the next shortest processing time, name it P2 and repeat step (3) for P1 and the new P2. Repeat this process by always selecting as P2 the processor with the shortest processing time among those that have not been tested yet.
(5) If such an exchange of subdags leads to a shorter finish time, a new solution has been found and an attempt to further improve this solution is made by repeating steps (1) through (5).
(6) If all processors have been tested without improving the solution, a local optimum has been reached.

At this point the algorithm repeats steps (1) through (6) above in an attempt to improve the local optimal solution by combining the subdags containing shared subtasks. The search is now performed by calls to the COMBINE subroutine described above and is carried out if (a) this is the first pass or (b) a previous search using EXCHANGE1 was successful in improving the solution. If the new search using COMBINE is successful, the algorithm attempts still another search using EXCHANGE1 and the entire procedure is repeated until no further improvement of the solution is possible.

The second local search algorithm LS_2 enlarges the neighborhood of a point searched for a better solution by using the subroutine EXCHANGE2 instead of EXCHANGE1. LS_2 is likely to obtain a better solution at the expense of longer run-times.

## COMPARISON OF ALGORITHMS

The greedy and local search algorithms presented in the previous section were evaluated and compared. Problem instances (i.e. dags representing tasks) were randomly generated [8]. For a given problem instance, the error ERR(A) of algorithm A was defined as:

$$ERR(A) = [SOL(A) - EXACT] / EXACT$$

where SOL(A) is the solution of an instance by algorithm A and EXACT is the exact solution obtained by the branch and bound algorithm. Computation times necessary for computing the exact solution were tolerable for small instances. Thus, average results obtained via the approximate algorithms could be compared to the exact solution for small randomly generated instances.

### Greedy Algorithms

Table 3 presents the average error, maximum error and average computation time for the algorithms GR_1, GR_2 and GR_3. The results presented are for 50 randomly generated instances of 10 subtasks and varying number of processors (M = 2, 3 and 4). Table 4 presents again the average and maximum errors as well as the average computation time for the greedy and the branch-and-bound algorithms. A fixed number of processors (M = 2) was used for these runs while the number of subtasks varied (N = 10, 20 and 30).

The average and maximum errors decrease as we go from the GR_1 to the GR_3 algorithms while the computation time increases. This small increase in computation time is expected because the greedy algorithms become more complex as we go from GR_1 to GR_3.

TABLE 3

Results for Algorithms GR_1, GR_2 and GR_3

| | AVERAGE ERROR | | |
|---|---|---|---|
| ALGORITHM | M=2 | M=3 | M=4 |
| GR_1<br>GR_2<br>GR_3 | 0.0423<br>0.0276<br>0.0120 | 0.0483<br>0.0279<br>0.0216 | 0.0321<br>0.0197<br>0.0197 |
| | MAXIMUM ERROR | | |
| ALGORITHM | M=2 | M=3 | M=4 |
| GR_1<br>GR_2<br>GR_3 | 0.1774<br>0.1923<br>0.0839 | 0.2047<br>0.1912<br>0.1726 | 0.2268<br>0.2212<br>0.2212 |
| | AVERAGE COMPUTATION TIME (msec) | | |
| ALGORITHM | M=2 | M=3 | M=4 |
| Branch-and-Bound<br>GR_1<br>GR_2<br>GR_3 | 68.0<br>8.8<br>10.8<br>14.8 | 246.2<br>6.6<br>14.0<br>15.6 | 264.0<br>6.6<br>14.6<br>19.4 |

TABLE 4

Results for Algorithms GR_1, GR_2 and GR_3

| | AVERAGE ERROR | | |
|---|---|---|---|
| ALGORITHM | N=10 | N=20 | N=30 |
| GR_1<br>GR_2<br>GR_3 | 0.0423<br>0.0276<br>0.0120 | 0.0128<br>0.0093<br>0.0073 | 0.0444<br>0.0299<br>0.0280 |
| | MAXIMUM ERROR | | |
| ALGORITHM | N=10 | N=20 | N=30 |
| GR_1<br>GR_2<br>GR_3 | 0.1774<br>0.1923<br>0.0839 | 0.0921<br>0.0897<br>0.0560 | 0.1340<br>0.0982<br>0.0928 |
| | AVERAGE COMPUTATION TIME (msec) | | |
| ALGORITHM | M=2 | M=3 | M=4 |
| Branch-and-Bound<br>GR_1<br>GR_2<br>GR_3 | 68.0<br>8.8<br>10.8<br>14.8 | 562.0<br>20.2<br>25.2<br>32.8 | 505.6<br>33.0<br>44.6<br>49.8 |

## Local Search Algorithms

Randomly generated starting solutions for the local search algorithms were tested first. In addition, solutions generated from the greedy algorithms were used to start the local search.

Table 5 presents the results for one randomly generated instance and 20 randomly generated starting solutions. Two processors were considered and the number of subtasks varied (N = 10, 20 and 30 ) for these experiments.

The results show that when the optimal solutions were not obtained, the maximum errors were very small. As expected, LS_2 has a higher probability to provide a better solution, but it requires considerably more computation time. In some instances, LS_2 may run even more slowly than the branch and bound algorithm.

TABLE 5

Results for Algorithms LS_1 and LS_2

| ALGORITHM | N=10 | N=20 | N=30 |
|---|---|---|---|
| LS_1<br>Number of optimal solutions found<br>Minimum error<br>Maximum error<br>Average run-time | <br>3<br>0.000<br>0.033<br>20 | <br>15<br>0.000<br>0.004<br>78 | <br>5<br>0.000<br>0.023<br>291 |
| LS_2<br>Number of optimal solutions found<br>Minimum error<br>Maximum error<br>Average run-time | <br>20<br>0.000<br>0.000<br>58 | <br>19<br>0.000<br>0.002<br>357 | <br>9<br>0.000<br>0.020<br>2358 |
| Branch-and-Bound<br>Optimal Solution<br>Run-time | <br>271<br>50 | <br>530<br>610 | <br>738<br>1700 |

Table 6 shows the results obtained with LS_1 and LS_2 for 50 randomly generated instances and one randomly generated starting solution. Two processors and N = 10, 20 and 30 subtasks were considered.

The starting solution was also generated by using one of the greedy approximate algorithms GR_1, GR_2 or GR_3. A local search algorithm, LS_1 or LS_2 , was then applied to improve the starting solution. The average and maximum errors for the results obtained by these combinations are presented in Tables 7 through 9 together with the average computation times. At the same time the results for these combinations are compared to those obtained from the greedy algorithms alone.

The presented results show that by applying local search to a solution obtained by the greedy approach algorithms a significant reduction in the average error can be achieved. A fixed number of procesſsors (M = 2) and a variable number of subtasks (N = 10, 20 and 30) were used for these runs.

**TABLE 6**

**Results for Algorithms LS_1 and LS_2**

| | AVERAGE ERROR | | |
|---|---|---|---|
| ALGORITHM | N=10 | N=20 | N=30 |
| LS__1<br>LS__2 | 0.0016<br>0.0002 | 0.0005<br>0.0000 | 0.0021<br>0.0012 |
| | MAXIMUM ERROR | | |
| ALGORITHM | N=10 | N=20 | N=30 |
| LS__1<br>LS__2 | 0.0246<br>0.0052 | 0.0036<br>0.0000 | 0.0179<br>0.0199 |
| | AVERAGE RUN-TIME | | |
| ALGORITHM | N=10 | N=20 | N=30 |
| LS__1<br>LS__2 | 14.6<br>43.4 | 47.0<br>363.6 | 153.4<br>1047.2 |
| | MAXIMUM RUN-TIME | | |
| ALGORITHM | N=10 | N=20 | N=30 |
| LS__1<br>LS__2 | 30.0<br>120.0 | 80.0<br>620.0 | 270.0<br>2700.0 |

**TABLE 7**

**Average Error for GR_1, GR_2, GR_3 and for LS_1 and LS_2 with Starting Solution Obtained from GR_1, GR_2 and GR_3**

| ALGORITHM | N=10 | N=20 | N=30 |
|---|---|---|---|
| GR__1<br>LS__1 (GR__1)<br>LS__2 (GR__1) | 0.0423<br>0.0020<br>0.0012 | 0.0128<br>0.0004<br>0.0001 | 0.0444<br>0.0029<br>0.0025 |
| GR__2<br>LS__1 (GR__2)<br>LS__2 (GR__2) | 0.0276<br>0.0023<br>0.0010 | 0.0093<br>0.0017<br>0.0004 | 0.0299<br>0.0038<br>0.0020 |
| GR__3<br>LS__1 (GR__3)<br>LS__2 (GR__3) | 0.0120<br>0.0023<br>0.0009 | 0.0073<br>0.0014<br>0.0003 | 0.0280<br>0.0028<br>0.0018 |

Table 9 shows that the time required for the branch and bound method shows a large standard deviation. Finally, a comparison of Table 6 to Tables 7 through 9 shows that the choice of the starting solution does not greatly affect the results of the local search algorithms. The use of the solution obtained from a greedy algorithm as an initial guess for the local search does not lead to significant or even consistent improvement over the case of randomly generated initial guess.

**TABLE 8**

**Maximum Error for GR_1, GR_2, GR_3 and for LS_1 and LS_2 with Starting Solution Obtained from GR_1, GR_2 and GR_3**

| ALGORITHM | N=10 | N=20 | N=30 |
|---|---|---|---|
| GR__1<br>LS__1 (GR__1)<br>LS__2 (GR__1) | 0.1774<br>0.0295<br>0.0295 | 0.0921<br>0.0023<br>0.0019 | 0.1340<br>0.0208<br>0.0195 |
| GR__2<br>LS__1 (GR__2)<br>LS__2 (GR__2) | 0.1923<br>0.0295<br>0.0295 | 0.0897<br>0.0105<br>0.0037 | 0.0982<br>0.0247<br>0.0195 |
| GR__3<br>LS__1 (GR__3)<br>LS__2 (GR__3) | 0.0839<br>0.0295<br>0.0295 | 0.0560<br>0.0105<br>0.0037 | 0.0928<br>0.0247<br>0.0199 |

**TABLE 9**

**Average Run-Time for GR_1, GR_2, GR_3 and for LS_1 and LS_2 with Starting Solution Obtained from GR_1, GR_2 and GR_3**

| ALGORITHM | N=10 | N=20 | N=30 |
|---|---|---|---|
| GR__1<br>LS__1 (GR__1)<br>LS__2 (GR__1) | 8.8<br>18.2<br>42.4 | 20.2<br>51.0<br>353.4 | 33.0<br>142.4<br>768.2 |
| GR__2<br>LS__1 (GR__2)<br>LS__2 (GR__2) | 10.8<br>14.6<br>44.2 | 25.2<br>43.6<br>346.4 | 44.6<br>134.6<br>762.0 |
| GR__3<br>LS__1 (GR__3)<br>LS__2 (GR__3) | 14.8<br>15.8<br>42.6 | 32.8<br>46.2<br>339.8 | 49.8<br>134.8<br>725.2 |
| Branch-and-Bound<br><br>Avg. Run-Time<br>Max. Run-Time | <br><br>68.0<br>450.0 | <br><br>562.0<br>2440.0 | <br><br>505.6<br>4750.0 |

## CONCLUSIONS

An optimal branch and bound algorithm was developed first for solving the task allocation problem. Its computation time in the worst case, however, increases faster than

$$O ( M^{N_s} / M! )$$

as shown in [8]. Approximate algorithms were then developed in order to obtain solutions close to the optimal one in polynomial time. The greedy algorithms require

$$O ( M N^2 )$$

computational time and can reach a solution with average error below 5%. GR_3 is the best among them.

The local search algorithms can reach a solution with average error below 0.5% at the expense of longer computational times. Algorithm LS_2 needs on the average much more computation time than LS_1, sometimes even more than the branch and bound algorithm. Taking into account the computation time needed for LS_2, LS_1 is obviously a much better choice.

## Acknowledgement

The authors wish to acknowledge the financial support of the Engineering Foundation under Grant RI-A-82-12.

## REFERENCES

[1] P-Y.R. Ma, E. Y.S. Lee, M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems", IEEE Trans. on Computers (January 1982), pp. 41-47.

[2] W. W. Chu, L. J. Halloway, M.T. Lan, K. Efe, "Task Allocation in Distributed Data Processing", Computer (November 1980), pp. 57-69.

[3] V. B. Gylys, J. A. Edwards, "Optimal Partitioning of Workload for Distributed Systems", Digest of Papers, COMPCON Fall 76 (September 1976), pp. 353-357.

[4] J. T. Lawson and M. P. Mariani, "Distributed Data Processing System Design - A Look at the Partitioning Problem", Tutorial: Distributed System Design, IEEE Computer Society, (1979), pp. 225-230.

[5] K.B. Irani and K.W. Chen, "Minimization of Interprocessor Communication for Parallel Computation", IEEE Trans. on Computers (November 1982), pp. 1067-1075.

[6] G.S. Rao, H.S. Stone, T.C. Hu, "Assignment of Task in a Distributed Processor System with Limited Memory", IEEE Trans. on Computers (April 1979), pp. 291-299.

[7] H.S. Stone, "Multiprocessor Scheduling with the aid of Network Flow Algorithms", IEEE Trans. on Software Eng., Vol. SE-3, No. 1 (January 1977), pp. 85-93.

[8] W. Liaw, "Task Allocation Problems in Distributed Computer Systems," M.S. Thesis, University of Houston (December 1985).

[9] M. R. Garey and D. S. Johnson, Computers and Intractability, W.H. Freeman and Co., San Francisco (1979).

[10] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press Inc. (1978).

[11] C. H. Papadimitriou, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Inc. (1982).

# SPEEDUP BOUNDS AND PROCESSOR ALLOCATION FOR PARALLEL PROGRAMS ON MULTIPROCESSORS

*Constantine D. Polychronopoulos* +

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801

*Utpal Banerjee*

Control Data Corporation
215 Moffett Park Drive
Sunnyvale, CA 94089

**ABSTRACT** -- *The main aim of this paper is to study allocation of processors to parallel programs executing on a multiprocessor system, and the resulting speedups. We first consider a general program represented as a sequence of steps consisting of parallel operations, then one represented as a task graph whose nodes are do across loops and whose edges represent precedence constraints, and finally a single do across loop. General bounds on program speedup are discussed and measurements of code parallelism for the LINPACK numerical package are presented.*

## 1. INTRODUCTION

Recently there has been an almost unanimous agreement in the computing community that the near future supercomputers will be based on the shared memory multiprocessor architecture. But there are divided opinions when the question comes to the number of processors needed for an efficient and cost effective, yet very fast multiprocessor. A number of pessimistic and optimistic reports have come out on this topic. One side uses Amdahl's law and "intuition" to argue against large systems [13], [1]. The other side cites simulations and real examples to support the belief that highly parallel systems with large numbers of processors are practical, and could be efficiently utilized to give substantial speedups [5], [8], [11], [3], [7]. This question will probably be answered in a definite way only by experience, after many multiprocessor supercomputers with varying numbers of processors will have been built.

Presently the computing community has adopted a conservative approach to this issue, and that is reflected in the first multiprocessors (e.g., CRAY-X-MP, CRAY-2, Alliant FX/8) that have appeared on the market; each of them comes with a small number of processors. Of course, this is due mostly to our inexperience in using efficiently a large number of processors. Only recently the applications community has been systematically involved in research on parallel algorithms, languages and software.

The investment in traditional (serial) software is so enormous that it will be many years before parallel software dominates the market. It is then natural to ask: " How can we efficiently run existing software on multiprocessor systems?" The answer to this question is well-known: by using powerful restructuring compilers. One such powerful restructurer is the Parafrase compiler developed over the last fifteen years at the University of Illinois ([10], [17]). Besides restructuring Fortran programs into a form suitable for execution on a range of high performance machines, Parafrase also supplies various statistics useful for performance analysis and system design decisions. Such statistics include program speedup, efficiency, number of useful processors, serial/parallel timing estimates, etc.

In this paper we start with a parallel program that is the result of restructuring a serial program for execution on a multiprocessor. We discuss different types of parallelism that can be observed in such a program, the kinds of overhead involved in parallel execution on multiprocessor systems, and the effect of task size and scheduling overhead on the speedup. We then address the problem of allocating available processors to different parts of the program and estimating the possible speedup. Program Task Graphs have been chosen to provide concrete representations of parallel programs. These are directed graphs where a node represents a do across loop and an edge represents a precedence constraint. (A do across loop is a loop whose iterations may be partially overlapped [5].)

We have restructured LINPACK using Parafrase and computed the fraction of parallel code for all subroutines. Our results contradict the original Amdahl conjecture that most programs have at least 10% serial code, and hence can achieve a maximum speedup of at most 10. The experiments strongly support the view that there is enough inherent parallelism in real programs so that large numbers of processors can be efficiently utilized.

In Section 2, we present the basic concepts and definitions that are used throughout the paper. Section 3 discusses program restructuring, partitioning, critical task size, and gives a brief outline of the Parafrase compiler used for our measurements. Section 4 presents general bounds on program speedup and the measurements of parallelism in LINPACK subroutines. Section 5 presents three models of program execution on multiprocessor machines and a heuristic algorithm for allocating processors to parallel programs. In Section 6, we consider analytically our general parallel loop model and derive the speedup formula for multiprocessors. Finally, Section 7 gives some conclusions.

## 2. BASIC CONCEPTS

A basic sequential machine is a single CPU computer that can carry out operations serially, taking one unit of time for each. A $p$-unit *Multiple Execution Scalar* (MES)

+ This work was supported in part by the **National Science Foundation** under Grant No. US NSF DCR84-10110, and in part by the **U. S. Department of Energy** under Grant No. US DOE DE FG02-85ER25001.

machine is composed of $p$ identical basic sequential machines, and each processor is driven by its own control unit. Because of its flexibility, we will consider only the MES machine in this paper. It is variously referred to as a multiprocessor, a parallel machine with $p$-processors, or simply a $p$-processor machine.

An assignment statement is a statement of the form $x = E$, where $x$ is a variable and $E$ an expression. A *do across* or DOACR loop [5] with *delay* $d$ has the form

$$\begin{array}{c} L : \text{DOACR } I = 1, N \\ B \\ \text{END} \end{array}$$

where $d$, $N$ are integer constants, $I$ an integer variable with the range $\{1, 2, ..., N\}$, $B$ a sequence of assignment statements and DOACR loops, and it is understood that the iterations of $L$ can be partially overlapped as long as there is a delay of at least $d$ units of time from the start of iteration $i$ to the start of iteration $i + 1$, $(i = 1, 2, ..., N - 1)$. For $L$, the index variable is $I$, the number of iterations is $N$, and the loop-body is $B$.

Consider now the two extreme cases of overlapping. If $d = 0$, there is complete overlapping, i.e. all the iterations of $L$ can be executed simultaneously. In this case the do across loop is called a *do all* loop, and we write DOALL in place of DOACR. If $d = b$, where $b$ is the execution time (assumed to be independent of $I$) of the loop-body $B$, then there is no overlapping, i.e. the iterations of $L$ must be executed serially, one after another. In this case the do across loop is a standard serial loop, and we write DOSER for DOACR. A BAS or a block of assignment statements is a special kind of serial loop, namely a loop with a single iteration. (A BAS may also be regarded as a special case of a do all loop.)

A program is a sequence of steps where each step consists of one or more operations that can be executed simultaneously. A program is *serial* if each step has exactly one operation; otherwise it is *parallel.* Two programs are semantically *equivalent* if they always generate the same output on the same input. Parallel programs are conveniently represented in terms of do across loops (Section 3).

Let $PROG_1$, $PROG_2$ be two equivalent programs and let their execution times on a $p$-processor machine be $T_p(PROG_1)$ and $T_p(PROG_2)$ respectively. Then the *speedup* obtained (on this machine) by executing $PROG_2$ instead of $PROG_1$ is denoted by $S_p(PROG_1, PROG_2)$ and is defined by

$$S_p(PROG_1, PROG_2) = \frac{T_p(PROG_1)}{T_p(PROG_2)}.$$

An immediate consequence of this definition is the following lemma.

**Lemma 2.1.** If $PROG_1, PROG_2, ..., PROG_n$ is a sequence of programs any two of which are equivalent, then

$$S_p(PROG_1, PROG_n) = \prod_{i=1}^{n-1} S_p(PROG_i, PROG_{i+1}).$$

We usually write $S_p$ for the speedup when the two programs involved are understood. Of special interest to us is the case where $PROG_1$ is a serial program and $PROG_2$ is an equivalent parallel program obtained by restructuring $PROG_1$. We assume that the execution time of a program is determined solely by the time taken to perform its operations, and that the total number of operations in a program is never affected by any restructuring. These assumptions are not very far from the truth; they help to keep the formulas simple, and yet let us derive important conclusions. If $T_1$ is the number of operations in the serial program $PROG_1$, then $T_1$ is also the execution time of $PROG_1$ on the basic sequential machine, or on any parallel machine (i.e., $T_1 = T_p(PROG_1)$). The equivalent parallel program $PROG_2$ also has $T_1$ operations, but now these operations are arranged in fewer than $T_1$ steps. We call $T_1$ the *serial execution time* of $PROG_2$ and it can be obtained simply by counting the operations in $PROG_2$. The execution time $T_p \equiv T_p(PROG_2)$ of $PROG_2$ on the $p$-processor machine will depend on the structure of the program, the magnitude of $p$, and the way the $p$ processors are allocated to different parts of $PROG_2$. To distinguish it from $T_1$, $T_p$ is referred to as the *parallel execution time* of $PROG_2$. The speedup of a program is then the ratio of its serial execution time to its parallel execution time.

For a given program, we have the *unlimited* processor case when $p$ is large enough so that we can always allocate as many processors as we please. Otherwise, we have the *limited* processor case. These two cases will be often discussed separately.

## 3. RESTRUCTURING, PROGRAM PARTITIONING AND CRITICAL TASK SIZE

In our program model we assume that parallelism is explicitly specified in the form of tasks (disjoint code segments) which are parallel loops (DOALL or DOACR). This can be done for any Fortran program written in a serial form by employing restructuring compilers. For this paper we used Parafrase, a restructuring compiler developed at the University of Illinois, to transform programs into parallel form and compute some experimental values presented in the following section. Although a detailed presentation of the capabilities of Parafrase is beyond the scope of this paper, we briefly describe its basic functions for the sake of completeness.

Parafrase transforms serial Fortran programs into a parallel form suitable for efficient execution on a variety of parallel architectures including MES. The compiler consists of a front-end which applies a series of machine independent transformations, and a back-end which is a set of machine dependent optimizations. For each compiled Fortran program, Parafrase builds a graph called the data dependence graph whose nodes represent program statements and edges represent data and control dependencies among statements [2]. Each branch of an IF or GOTO statement is assigned a *branching probability* by the user, or automatically by Parafrase [11]. We can therefore view any program as a sequence of assignment statements, where each statement has an accumulated *weight* associated with it. All loops in a program are automatically normalized, i.e., loop indexes assume values in [1, $N$] for some integer $N$. As in the case of branching statements, unknown loop upper bounds are either defined by the user, or automatically by the compiler (using a default value). During parallel execution of the restructured program, data and control dependencies must be observed to assure that program semantics is preserved. The data dependence graph of the program is used by most transformations as a guide for this reason.

One of the most important transformations in Parafrase is the recognition and marking of DOACR loops (including DOALL and DOSER loops as special cases of DOACR). If we consider a block of assignment statements (BAS) as a serial loop with a single iteration, a restructured program can be viewed as a series of outermost DOACR loops with each such loop being arbitrarily complex. This defines a "natural" partition of a restructured program into a series of code segments or tasks. Dependencies may exist between any pair of segments in the program. We can thus define the *Program Task Graph* as a directed graph $G(V, E)$, where the nodes in $V$ are the outermost loops $L_i$ in the program, and there is an edge from a node $L_i$ to a node $L_j$ iff loop $L_j$ depends on loop $L_i$. Since backward dependencies are not allowed, $G(V, E)$ is acyclic.

In a restructured program we may observe two types of parallelism: *horizontal* and *vertical.* Horizontal parallelism results by executing a DOACR loop on two or more processors, or equivalently, by simultaneously executing different iterations of the same loop. Vertical parallelism in turn, is the result of the simultaneous execution of two or more different loops (tasks). Two or more loops can execute simultaneously only if there exists no control or data dependencies between any two of the loops. In the general case the program task graph exposes both types of parallelism. When we execute such a task graph on an MES machine, we must decide how to allocate the available processors to the program tasks so that the program speedup is maximized.

A serious problem arises when while executing a restructured program on an MES machine, we attempt to minimize the overheads of communication, synchronization and scheduling. This is a non-trivial optimization problem, and attempts to minimize such overheads usually results in reducing the degree of program parallelism. Most instances of this optimization problem have been proven to be NP-Complete [6]. Most heuristic algorithms attempt to minimize the communication cost by merging nodes of the graph together to avoid the overhead involved in communicating data from one processor to another. This however often reduces the degree of available vertical parallelism.

As an example of node merging, consider two loops $L_1$ and $L_2$ in our restructured program model, with data dependencies going from $L_1$ to $L_2$. The dependencies restrict the two loops to execute in this order since data computed in $L_1$ are used by $L_2$. In this case only horizontal parallelism inside each loop can be exploited. If we do not coordinate the execution of $L_1$ and $L_2$, then data computed inside $L_1$ will have to be stored in a shared memory upon completion of $L_1$, and then fetched from that memory to the processors executing $L_2$. If on the other hand we consider the two loops as a single task, then we can bind iterations of $L_1$ and corresponding iterations of $L_2$ to specific processors. In this manner data computed by a particular iteration of $L_1$ and used by the corresponding iteration of $L_2$ need only be stored in fast registers of the processor, thus avoiding the overhead of the redundant store and fetch. For relatively small loops the savings by such "task merging" can be very significant.

Task merging can also be used to decrease scheduling overhead that is involved when we distribute different program nodes across different processors. This scheduling overhead is in addition to the synchronization overhead and may become disastrous especially for very small tasks. For the CRAY-XM-P for example, the overhead involved with scheduling two parallel tasks can be several *msecs.* This overhead imposes a minimum size on parallel tasks, below which the speedup becomes rather a slowdown (i.e., $S_p < 1$). We call this the *critical task size.*

If during the execution of a program we schedule a set of parallel tasks, the parallel execution time is augmented by $O_T$, where $O_T$ is the scheduling overhead. The maximum expected speedup therefore is given by

$$S_p = \frac{T_1}{T_1/p + O_T}.$$

In order to have a speedup of at least 1, we must have $T_1 \geq T_1 / p + O_T$, i.e., $T_1 \geq p*O_T / p-1$ which gives the critical task size as a function of the overhead and the number of processors. More generaly, the minimum program size $T_{\min}$ required to obtain a given speedup $S^*$ on $p$ processors should satisfy:

$$\frac{T_{\min}}{T_{\min} / p + O_T} \geq S^*, \quad \text{or} \quad T_{\min} \geq \frac{p*O_T*S^*}{p - S^*}.$$

Program partitioning for minimizing data communication and scheduling overheads is a complicated optimization problem [15], [16].

## 4. GENERAL BOUNDS ON SPEEDUP

In this section we consider an arbitrary parallel program, and think of it simply as a sequence of steps where each step consists of a set of operations that can execute in parallel. The total number of operations (and hence the serial execution time) is denoted by $T_1$. Let $p_0$ denote the maximum number of operations in any step.

Suppose first we are using a $p$-processor system with $p \geq p_0$. (This is the unlimited processor case). Let $\phi_i T_1$ denote the number of operations that belong to steps containing exactly $i$ operations, $(i = 1, 2,..., p_0)$. Then $\phi_i$ is the fraction of the program that can utilize exactly $i$ processors, and we have $\sum_{i=1}^{p_0} \phi_i = 1$. We call $f = \sum_{i=2}^{p_0} \phi_i$ the *parallel part* of the program or the *fraction of parallel code,* and $1 - f = \phi_1$ the *serial part* of the program or the *fraction of serial code.* (At least $p - p_0$ processors will always remain unused.)

Consider now a limited processor situation with a $p$-processor machine where $p < p_0$. The steps with more than $p$ operations have to be folded over and replaced with a larger number of steps with $p$ operations. (For simplicity, we are assuming that each new step has exactly $p$ operations, although one of them may actually have fewer than $p$). Let $f_i \equiv f_i(p)$ denote the fraction of the modified program that can utilize exactly $i$ processors, $(i = 1, 2,..., p)$. Then we have

$$f_i = \phi_i \qquad (i = 1, 2,..., p - 1), \qquad \text{and} \qquad f_p = \sum_{i=p}^{p_0} \phi_i.$$

As long as $p \geq 2$, the parallel part $f$ is given by $\sum_{i=2}^{p} f_i$ and the serial part $1 - f$ by $f_1$.

An arbitrary $p$-processor machine is assumed in the following. The first two results are well-known [3], [12].

**Theorem 4.1.** $\displaystyle \frac{1}{S_p} = \sum_{i=1}^{p} \frac{f_i}{i}.$

**Proof:** When executing on a $p$-processor machine, the fraction of the program that uses exactly $i$ processors is $f_i T_1$, $(i = 1, 2, \ldots, p)$. Hence, the number of steps where $i$ processors are active is $\dfrac{f_i T_1}{i}$. The total number of steps is then given by

$$T_p = \sum_{i=1}^{p} \frac{f_i T_1}{i} = T_1 \sum_{i=1}^{p} \frac{f_i}{i}.$$

Since $T_1$ is the serial and $T_p$ the parallel execution time of the program, we get

$$\frac{1}{S_p} = \frac{T_p}{T_1} = \sum_{i=1}^{p} \frac{f_i}{i}. \quad \square$$

**Corollary 4.2.** $1 \leq S_p \leq p$.

**Proof:** We have $f_i \geq \dfrac{f_i}{i} \geq \dfrac{f_i}{p}$ $(i = 1, 2, \ldots, p)$. Hence

$$\sum_{i=1}^{p} f_i \geq \sum_{i=1}^{p} \frac{f_i}{i} \geq \sum_{i=1}^{p} \frac{f_i}{p}$$

$$\text{or,} \quad 1 \geq \sum_{i=1}^{p} \frac{f_i}{i} \geq \frac{1}{p}$$

$$\text{so that} \quad 1 \geq \frac{1}{S_p} \geq \frac{1}{p},$$

i.e. $1 \leq S_p \leq p$. □

**Corollary 4.3.** $S_p \leq 1 / f_1$.

**Corollary 4.4.** The speedup $S_p$, the number of processors $p$ and the fraction of parallel code $f$ satisfy (for $p > 1$)

$$S_p \leq \frac{p}{f + (1 - f)p}, \tag{4.1}$$

$$f \geq \frac{S_p - 1}{S_p} * \frac{p}{p - 1}, \tag{4.2}$$

$$\text{and} \quad p \geq \frac{f S_p}{1 - (1 - f) S_p}. \tag{4.3}$$

**Proof:** These three inequalities are equivalent; from any one the other two can be derived easily. Note that

$$\sum_{i=1}^{p} \frac{f_i}{i} = f_1 + \sum_{i=2}^{p} \frac{f_i}{i} \geq f_1 + \sum_{i=2}^{p} \frac{f_i}{p} = 1 - f + \frac{f}{p},$$

since $f = \sum_{i=2}^{p} f_i$ and $f_1 = 1 - f$. Then by Theorem 4.1,

$$\frac{1}{S_p} \geq 1 - f + \frac{f}{p}, \text{ so that } S_p \leq \frac{p}{f + (1 - f)p}. \ \square$$

| Sub. Name | $f$ | Sub. Name | $f$ | Sub. Name | $f$ |
|---|---|---|---|---|---|
| SPOFA | 0.9997 | SSIFA | 0.9862 | SPBFA | 0.9257 |
| SQRDC | 0.9988 | SPODI2 | 0.9853 | SGBFA | 0.9189 |
| SPBDI1 | 0.9975 | SGESL1 | 0.9807 | SGBSL2 | 0.9164 |
| SGBDI1 | 0.9974 | SSISL | 0.9806 | SGBCO | 0.8561 |
| SGEDI2 | 0.9961 | STRSL0 | 0.9773 | SPBCO | 0.8314 |
| SQRDC1 | 0.9961 | STRSL1 | 0.9773 | SSIDI3 | 0.7353 |
| SSIDI2 | 0.9961 | SPOSL | 0.9767 | SGBSL1 | 0.6545 |
| SSVDC1 | 0.9954 | SGESL2 | 0.9762 | STRCO | 0.6113 |
| SPODI1 | 0.9950 | STRSL2 | 0.9753 | SPBSL | 0.5659 |
| SSICO | 0.9905 | STRSL3 | 0.9753 | SGTSL | 0.5295 |
| SQRSL1 | 0.9900 | SPOCO | 0.9751 | SPPDI | 0.5064 |
| SQRSL2 | 0.9900 | SPPCO | 0.9746 | SSPSL | 0.4010 |
| SQRSL4 | 0.9900 | SSIDI1 | 0.9746 | SPTSL | 0.3799 |
| SQRSL5 | 0.9900 | SGEDI1 | 0.9745 | SSPDI | 0.3615 |
| SSPCO | 0.9896 | SGECO | 0.9664 | SSPFA | 0.1348 |
| SQRSL3 | 0.9868 | SPPSL | 0.9629 | | |
| SGEFA | 0.9862 | SPPFA | 0.9350 | | |

Table 1. Values of $f$ for LINPACK subroutines.

Now, we have a program that can use a maximum number of $p_0$ processors. If the fraction $\phi_1$ of serial code in it is very small, we can choose $p$ $(> 1)$ processors such that $f_p = \sum_{i=p}^{p_0} \phi_i \approx 1$. Then, since

$$\frac{1}{S_p} = \sum_{i=1}^{p-1} \frac{f_i}{i} + \frac{f_p}{p} \quad \text{we get}$$

$$\left| \frac{1}{S_p} - \frac{f_p}{p} \right| = \left| \sum_{i=1}^{p-1} \frac{f_i}{i} \right| \leq \sum_{i=1}^{p-1} f_i = 1 - f_p \approx 0,$$

or equivalently, $S_p \approx p / f_p \approx p$. Thus, if for some $p > 1$, $\sum_{i=p}^{p_0} \phi_i \approx 1$, then the program runs very efficiently on a $p$-processor system giving an almost linear speedup. In this case, given the coefficients $\phi_i$ for the particular program, we can always determine the maximum number of processors that would get a linear speedup.

Based on Corollary 4.3, Amdahl and some other researchers thereafter questioned the usefulness of very large MES systems, since, according to their argument, the majority of programs have an average of more than 10% serial code and therefore their speedup on any MES machine is bounded above by 10.

We conducted some experiments to measure the fraction of parallel code $f$ in LINPACK, a widely used numerical package for solving systems of linear equations. Knowing the serial execution time $T_1$, the parallel execution time $T_p$ and the number of processors $p$ that were used during the execution of a subroutine, we can easily compute a lower bound for $f$ from (4.2). All the above parameters are supplied by Parafrase. On the other hand, if the value of $f$ for a particular subroutine is known and we want to achieve a specific speedup $S_p$ for this subroutine, then (4.3) gives us a lower bound on the number of processors that we must use.

The sorted (minimum) values of $f$ are shown in Table 1. The measurements were done on LINPACK subroutines after they had been restructured by Parafrase. From Table 1 we observe that the majority of subroutines have a very high fraction of parallel code. For the first 37 subroutines (out of 49), the average fraction of parallel code was $f \geq 0.9784$. Almost 76% of the subroutines have $f > 0.9$ and only 18% have $f < 0.8$.

Considering that LINPACK is a typical numerical package not very amenable to restructuring, the results of Table 1 are very encouraging. EISPACK for example (another numerical package), should be expected to have a much higher value of $f$ than LINPACK [11]. Since most numerical packages are more amenable to restructuring than LINPACK, Amdahl's pessimism should not be taken very seriously when designing large multiprocessor systems. His claim for the non-effectiveness of systems with large numbers of processors is mostly based on programs that exhibit $f < 0.9$. As mentioned in Section 3, the real performance threat for large MES systems lies in scheduling and interprocessor communication overheads.

Secondly, we should consider all possible operating modes of a multiprocessor. There is no question that there exist numerical programs that could fully exploit hundreds or thousands of processors. For programs that utilize only a few processors, MES systems can be operated in a multiprogramming mode to keep system utilization high. The question then breaks down to whether we can have sites with enough users (workload) to keep system utilization at acceptable levels. The answer to this question is rather obvious and it is more of a managerial and planning issue.

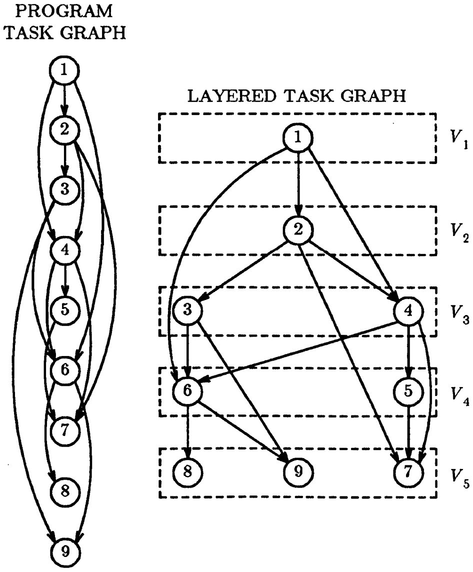


Figure 1: A example of a program task graph and its corresponding layered graph.

## 5. SPEEDUP AND PROCESSOR ALLOCATION FOR TASK GRAPHS

We consider here an arbitrary parallel program represented by a task graph $G \equiv G(V, E)$. Recall from Section 3 that this graph is defined on a restructured program with nodes representing outermost DOACR loops, and edges representing data and control dependencies among loops. Let there be $n$ nodes in $V$: $v_1, v_2, \ldots, v_n$. These nodes can be partitioned into disjoint layers $V_1, V_2, \ldots, V_k$, such that (1) all nodes in a given layer can execute in parallel, and (2) the nodes in a layer $V_{i+1}$ can start executing as soon as all the nodes in layer $V_i$ have finished, $(i = 1, 2, \ldots, k - 1)$. To construct this layered graph of $G$, we use a modified Breadth First Search scheme for labeling the nodes of the graph. Initially, the first node of the graph (corresponding to the lexically first loop of the program), is labeled 1 and queued in a FIFO queue $Q$. At each following step $v_j$, the node at the front of $Q$ is removed, and if $i$ is its label, all nodes adjacent to $v_j$ are labeled $i + 1$, and are queued in $Q$. Note that a node may be relabeled several times but its final label is the largest assigned to it. When $Q$ becomes empty, the labeling process terminates and we get the layered graph by grouping all nodes with label $i$ into layer $V_i$. An example of a program task graph and its corresponding layered task graph is shown in Figure 1. We consider below three execution models of a layered task graph on a $p$-processor MES machine. The most general and the two extreme cases are discussed.

As usual $T_1$ denotes the total number of operations in the whole program. For the node $v_j$, let $g_j T_1$ denote the number of operations in the node and $T_{pj}$ its parallel execution time, $(j = 1, 2, \ldots, n)$. Then $\sum_{j=1}^{n} g_j = 1$. The absolute speedup $S_{pj}^A$ of $v_j$ is the speedup obtained by considering the node separately as a program and is given by $S_{pj}^A = g_j T_1 / T_{pj}$.

**Case 1.** (Horizontal parallelism). Let $k = n$ and each layer $V_i$ consist of a single node $v_i$, $(i = 1, 2, \ldots, n)$. To get the maximum speedup for the whole program on $p$ processors, we need to get the maximum speedup for each node. Detailed formulas are given below.

The relative speedup $S_{pi}^R$ of $v_i$ is the speedup of the whole program when only $v_i$ is executed in parallel and all other nodes are executed serially. Thus

$$S_{pi}^R = \frac{T_1}{T_1 - g_i T_1 + T_{pi}}.$$

**Lemma 5.1.** The absolute and relative speedups of a node $v_i$ are connected by the equation

$$\frac{g_i}{S_{pi}^A} = \frac{1}{S_{pi}^R} - 1 + g_i \qquad (i = 1, 2, \ldots, n).$$

**Proof:** It follows directly from the above definitions. □

**Theorem 5.2.** The speedup $S_p$ of the whole program (when all nodes are executed in parallel) is related to the absolute and relative speedups of the individual nodes by the following equations:

$$\frac{1}{S_p} = \sum_{i=1}^{n} \frac{g_i}{S_{pi}^A} = \sum_{i=1}^{n} \frac{1}{S_{pi}^R} - n + 1.$$

**Corollary 5.3.** If all $n$ nodes give the same absolute speedup $S_p^A$, then $S_p^A = S_p$. If all $n$ nodes give the same relative speedup $S_p^R$, then

$$S_p^R \leq \frac{np}{np - p + 1} < \frac{n}{n - 1}.$$

**Proof:** The first assertion follows immediately from the above theorem, since $\sum_{i=1}^{n} g_i = 1$. For the second, we see that when the relative speedups are all equal

$$\frac{1}{S_p} = \sum_{i=1}^{n} \frac{1}{S_p^R} - n + 1 = \frac{n}{S_p^R} - n + 1.$$

Since $S_p \leq p$, this implies

$$\frac{n}{S_p^R} - n + 1 \geq \frac{1}{p} \text{ or}$$

$$S_p^R \leq \frac{np}{np - p + 1} = \frac{n}{n - 1 + \frac{1}{p}} < \frac{n}{n - 1}. \square$$

Each node of the graph can be an arbitrarily complex nested loop containing DOSER, DOALL, and DOACR loops. The problem of optimal processor allocation to such nodes has been solved optimally in [14].

**Case 2.** (Vertical parallelism). Let the task graph be *flat,* i.e. let there be a single layer $V$ consisting of $n$ nodes. This is the case when no dependencies exist between any pair of nodes. Here we may exploit vertical parallelism by executing all program nodes simultaneously. We consider the extreme case where each node requests exactly one processor. Since each node is allocated one processor, if $n \leq p$ the execution time is dominated by the largest task. In the general case bin-packing can be used to evenly distribute the $n$ nodes into $p$ bins. The Multifit heuristic algorithm, for example, can be used in this case [4]. Then, if $b_i$, $1 \leq i \leq p$ denotes the largest bin, we have the following theorem.

**Theorem 5.4.** The total speedup resulting from the parallel execution of an $n$-node flat graph on $p$ processors, where each node is allocated one processor, is given by

$$S_p = \frac{1}{\sum_{v_j \in b_i} g_j}.$$

**Proof:** The proof follows directly from the definition of speedup and the assumptions stated above. □

**Corollary 5.5.** If $n \leq p$ then

$$S_p = \frac{1}{\max(g_1, g_2, ..., g_n)}.$$

**Proof:** This follows from the previous theorem since each bin contains one node. □

**Case 3.** (Horizontal and vertical parallelism). In the most general case we have a program that exposes both types of parallelism, horizontal and vertical. In other words, the task graph consists of $k$ ($> 1$) disjoint layers $V_1, V_2, \ldots, V_k$ with at least one layer containing two or more nodes (Figure 1). If $|V_i|$ is the cardinality of the i-th layer, we assume that $|V_i| \leq p$, $(i = 1, 2, ..., k)$. (In the case of $|V_i| > p$ we fold and fuse nodes such that $|V_i| \leq p$ [15].) Our aim in this case is to exploit horizontal and vertical parallelism in the best possible way. Maximizing speedup is equivalent to minimizing parallel execution time. For each node of the task graph $v_j$ we define $r_j$ to be the maximum number of processors that the node could use. When $r_j = 1$, $(j = 1, 2, ..., n)$ our problem is reduced to the classical multiprocessor scheduling problem, which has been proved NP-Complete [6]. Our general problem can be reduced to the latter one by decomposing each node $v_j$ into $r_j$ independent sub-nodes of equal size. This trivially proves that our problem is also NP-Complete. Heuristic solutions are therefore the only acceptable approach to solving the problem suboptimally in polynomial time.

Below we present a simple linear-time heuristic algorithm for allocating processors to general task graphs. We call this *Proportional Allocation* heuristic since it allocates to each node a number of processors which is proportional to the size of the node (this term was borrowed from [16] where a similar idea was independently proposed). The idea behind proportional allocation is to allocate processors to the task graph on a layer-by-layer basis, so that the load in each layer is evenly distributed across the available processors, resulting in a suboptimal execution time. Variations of the proportional allocation heuristic and experimental results are given elsewhere [15].

For each layer $V_i$, $(i = 1, 2, ..., k)$ of $G$ we carry out the following steps. (The notation $x \leftarrow a$ used below indicates the assignment of an expression $a$ to variable $x$.)

**Step 1.** Each node $v_j \in V_i$ is allocated one processor. If $|V_i| = q_i$, then the number of remaining processors is $p_R = p - q_i$. The tasks in $V_i$ are arranged in order of decreasing size.

**Step 2.** The remaining $p_R$ processors are allocated to the nodes of $V_i$ with $r_j > 1$ so that each node receives a number of processors proportional to its size. For a node $v_j$ in $V_i$ with $r_j > 1$, the serial execution time is $t_j = g_j T_1$. Let $\tau_i = \sum_{v_j \in V_i} (g_j T_1)$ denote the total execution time of all nodes $v_j \in V_i$ with $r_j > 1$. Then, for all such nodes perform:

$$p_j = \left\lceil p_R * \frac{t_j}{\tau_i} \right\rceil \tag{5.1}$$

$$p_j \leftarrow \min(r_j - 1, p_j) \tag{5.2}$$

$$p_R \leftarrow p_R - p_j \tag{5.3}$$

where $p_j$ is the number of processors allocated to node $v_j$. Steps (5.1), (5.2), and (5.3) are repeated until all processors are allocated ($p_R = 0$), or all nodes in $V_i$ are processed. It should be noted that if at the end $p_R > 0$, then $p_j + 1 = r_j$, $(j = 1, 2, ..., q_i)$. A formal description of the proportional allocation heuristic is given in Figure 5. A simple example of the application of this algorithm to a single layer with DOALL loops, is shown in Figures 2 and 3. The number of processors allocated to each loop by our algorithm is shown in the table of Figure 2. Figure 3(a) shows the processor/time diagram when loops are executed one by one on an unlimited (in this case) number of processors, with a total execution time of 22 units. Figure 3(b) shows the processor/time diagram for the allocation performed by our algorithm. Processors were allocated so that both horizontal and vertical parallelism are utilized; 16 units is the total execution time in this case. The total program speedup on $p$ processors that results from the application of the above heuristic is given by the following theorem.

**Theorem 5.6.** The total program speedup that results from

```
 DOALL 1 I1 = 1, 7
      }1
1 END

 DOALL 2 I2 = 1, 14
      }8
2 END

 DOALL 3 I3 = 1, 5
      }5
3 END

 DOALL 4 I4 = 1, 20
      }4
4 END

 DOALL 5 I5 = 1, 24
       }6
5 END
```

| Number of processors allocated to each loop | |
|---|---|
| Loop Number | No. of Processors |
| 1 | 1 |
| 2 | 9 |
| 3 | 3 |
| 4 | 7 |
| 5 | 12 |
| Total | 32 |

Figure 2. A simple program with DOALLs and the processor allocation profile.

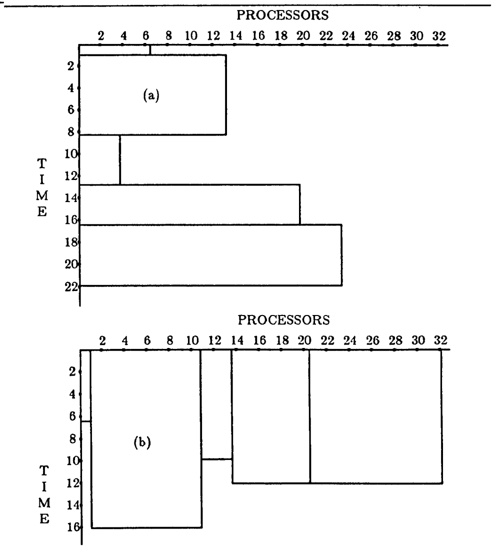


Figure 3. (a): The processor/time diagram for the program of Figure 2 (Case 1).
(b): The processor/time diagram after the application of the algorithm (Case 3).

```
DOACR I = 1, 8    {d=4}
    }10
END
```

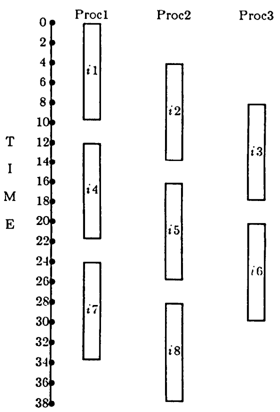


Figure 4. An example of the application of Theorem 6.1 for $p = 3$.

the parallel (vertical and horizontal) execution of a $k$-layer task graph on $p$ processors is given by

$$S_p = \frac{1}{\sum_{i=1}^{k} \lambda_i} \quad \text{where} \quad \lambda_i = \max_{v_j \in V_i} \left\{ \frac{g_j}{S^A_{p_j}} \right\} \qquad (5.4)$$

(where $S^A_{pj}$ is the absolute speedup of node $v_j$ for the number of processors $(p_j)$ allocated to it by the proportional allocation heuristic.)

Note that Theorems 5.2 and 5.4 are special cases of Theorem 5.6. If the graph is reduced to a flat graph, then $k=1$ and Corollary 5.5 holds. On the other hand, if each layer contains one node (linearized) then $q_i = 1$ in (5.4), ($i = 1, 2, ...., k$), and thus Theorem 5.2 holds true.

## 6. SPEEDUP AND PROCESSOR ALLOCATION FOR DOACR LOOPS

In this section we focus on a single node of the task flow graph representing the given program, i.e. a DOACR loop. We extended and generalized the do across model, to allow for idling processor time caused by do across delays. In [5] it is assumed that no processor may become idle unless it completes all iterations assigned to it. This is true only in certain special cases. As we shall see later by means of an example, each processor may have to idle between successive iterations. The following theorem generalizes the do across model and accounts for idle processor time.

**Theorem 6.1.** Consider a DOACR loop with $N$ iterations and delay $d$, and let $b$ denote the execution time of the loop-body. Then $p$ processors can be allocated to the iterations of the loop in such a way that the speedup is given by $S_p = Nb/T_p$, where

$$T_p = \Big( (\lceil N/p \rceil - 1) \max(\mathrm{b}, \mathrm{pd}) + d((N - 1) \bmod p) + b \Big) \qquad (6.1)$$

**Proof:** Let us number the iterations $1, 2, ..., N$ in their natural order and the processors $1, 2, ..., p$ in any order. The processors are allocated to the iterations as follows. Assume first that $N > p$. Iteration 1 goes to processor 1, iteration 2 goes to processor 2, ..., iteration $p$ goes to processor $p$. Then iteration $(p + 1)$ goes to processor 1, etc., and this scheme is repeated as many times as necessary until all the iterations are taken care of. We can now think of the iterations arranged in a $\lceil N/p \rceil$ X $p$ matrix, where the columns represent processors. If $N \leq p$, we employ the same scheme, but now we end up with a 1 X $N$ matrix instead.

Let $t_k$ denote the starting time of iteration $k$, ($k = 1, 2, ..., N$), and assume that $t_1 = 0$. (We measure time when iteration 1 starts executing). Let us find an expression for $t_N$. First, assume that $N > p$. For any iteration $j$ in the first row, the time $t_j$ is easily found:

$$t_j = (j - 1)d \qquad (j = 1, 2, ..., p)$$

Now iteration $(p + 1)$ must wait until its processor (i.e., processor 1) has finished executing iteration 1, and $d$ units of time have elapsed since $t_p$, the starting time of iteration $p$ on processor $p$. Hence

$$t_{p+1} = \max(b, t_p + d) = \max(b, pd).$$

The process is now clear. If we move right horizontally (in the matrix of iterations), each step amounts to a time delay of $d$ units. And if we move down vertically each step adds a delay of $\max(b, pd)$ units. Thus the time $t_k$ for an iteration that lies on row $i$ and column $j$ will be given by

$$t_k = (i - 1) \max(\mathrm{b}, \mathrm{pd}) + (j - 1)d.$$

For the last iteration, we have $i = \lceil N/p \rceil$ and

$$j = \begin{cases} N \bmod p & \text{if N mod p} > 0 \\ p & \text{otherwise} \end{cases}$$

Since $j - 1$ can be written as $(N - 1) \bmod p$, we get

$$t_N = \Big( \lceil N/p \rceil - 1 \Big) \max(b, pd) + d((N - 1) \bmod p).$$

Now let $N \leq p$. It is easily seen that

$$t_N = (N - 1)d$$

$$= \left(\lceil N/p \rceil - 1\right) \max(b, pd) + d((N - 1) \bmod p).$$

Finally, since the parallel execution time $T_p$ is given by $T_p = t_N + b$, the proof of the theorem is complete. □ Figure 4 shows an example of the application of Theorem 6.1. The DOACR loop of Figure 4 has 8 iterations, a delay $d = 4$, and a loop-body size of 10. The total parallel execution time on $p = 3$ processors is 38 units, as predicted by Theorem 6.1. We can maximize the speedup of an arbitrarily nested DOACR loop which executes on $p$ processors by using an optimal processor allocation algorithm described in [14].

**Corollary 6.2.** Consider a sequence of $m$ perfectly nested DOALL loops numbered 1, 2,..., $m$ from the outermost loop to the innermost. Let $S_{p_i}$ denote the speedup of the construct on a $p_i$-processor machine when only the $i$th loop executes in parallel and all other loops serially, ($i = 1, 2, \ldots, m$). Then the speedup $S_p$ on a $p$-processor machine, where $p = \prod_{i=1}^{m} p_i$ and $p_i$ processors are allocated to the $i$th loop, is given by $S_p = \prod_{i=1}^{m} S_{p_i}$.

## 7. CONCLUSIONS

We have studied the problem of allocating processors to parallel programs executing on a multiprocessor system, and measuring the speedups. Our experiments with the software packages LINPACK show that there is a high degree of parallelism available in ordinary programs, and that after proper restructuring such programs would be able to utilize fairly large multiprocessors systems. As a convenient unit of parallel program we have taken a DOACR loop which generalizes serial and DOALL loops. The speedup formula for DOACR loops has been generalized, and several other results have been derived that increase our understanding of how the speedup of a program depends on various factors. The general problem of processor allocation to programs represented as task graphs of DOACR loops being NP-Complete, we have given heuristic methods that seem to work well in practical situations and in linear time.

**ACKNOWLEDGEMENTS:** Many thanks are due to Prof. David J. Kuck for discussions, suggestions, and guidance on this work, and to Prof. David A. Padua for encouragement and discussions on the topic.

## REFERENCES

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Computer Conference Proc.*, Vol. 30, 1967.

[2] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-989, October, 1979.

[3] U. Banerjee, "Bounds on the Parallel Processor Speedup," *Proc. of 19th Annual Allerton Conference on Communication, Control, and Computing,* September-October, 1981.

[4] E.G. Coffman, M.R. Garey, D.S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling," *SIAM J. Comput.*, Vol. 7, No. 1, February, 1978,

[5] R. Cytron, "Compile-Time Scheduling and Optimizations for Asynchronous Machines," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-84-1177, October, 1984.

[6] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. Freeman and Company, San Fransisco, 1978.

[7] M. J. Flynn, J. L. Hennessy, "Parallelism and Representation Problems in Distributed Systems," *IEEE Trans. on Computers,* C-9, 1980.

[8] C. Kruskal, "An Optimistic View on Speedup Bounds," Lecture Notes, NATO Summer School, West Germany, July, 1984.

[9] D. J. Kuck, *The Structure of Computers and Computations,* Volume 1, John Wiley and Sons, New York, 1978.

[10] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference,* October, 1980.

[11] D. J. Kuck et al, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," *International Conf. on Parallel Processing.* August, 1984.

[12] R. B. Lee, "Performance Bounds in Parallel Processor Organizations," *Proc. Symposium on High Speed Computer and Algorithm Organization,* Academic Press, N. Y., 1977.

[13] M. Minsky, "Form and Computer Science," *JACM,* Vol. 17, 1970.

[14] C. D. Polychronopoulos, D. J. Kuck, D. H. Padua, "Execution of Parallel Loops on Parallel Processor Systems," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, August, 1986.

[15] C. D. Polychronopoulos, PhD Thesis in preparation, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1986.

[16] A. Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," CSRD Rpt. No. 520, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1985.

[17] M. Wolfe, "Supercompilers for Supercomputers," Ph.D. Thesis, UIUCDCS-R-82-1105, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

# SOFTWARE BASED MUTUAL EXCLUSION IN A MULTIPROCESSOR

Bernard T. Geary

Goodyear Aerospace Corporation
Akron, Ohio 44315

Abstract -- This paper is directed to the solution of a mutual exclusion problem involving a multiprocessor system. The multiprocessor contains six asynchronous microprocessors operating in a tightly coupled network. The system software has a master/slave organization. One processor is system executive or master while the remaining processors function as slaves and receive their tasks from the master. The task, or process, of master was previously performed on a full time basis by only one of the processors in the system. In the interests of efficiency and fault tolerance, the task of master can now float from processor to processor whenever needed. This paper presents a fast, software based method that allows only one processor at a time to assume the master task.

## Introduction

A multiprocesssor system exists called the MAPP (Modular Adaptive Parallel Processor). It was built by Goodyear Aerospace for AFWAL at Wright Patterson AFB. The MAPP contains six asynchronous microprocessors operating in a tightly coupled network. The system software has a master/slave organization. One processor is system executive or master while the remaining processors function as slaves and receive their tasks from the master. The task, or process, of master was previously performed on a full time basis by only one of the processors in the system. It was desired, in the interests of efficiency and fault tolerance, that the task of master be allowed to float from processor to processor whenever needed.

The primary problem associated with floating the master task was how to prevent two or more processors from acquiring the master task simultaneously. In accomplishing this, the following goals also had to be met:

1.) No system hardware modifications were allowed,
2.) The selected method had to be compatible with the current system software,
3.) The selected method had to be fault tolerance,
4.) The selected method had to be fast,
5.) Memory requirements had to be kept to a minimum,
6.) Future expansion of the system had to be allowed.

## Overview of the MAPP System

The MAPP is a homogeneous multiprocessor that has been primarily used for real-time signal correlation. The MAPP system architecture is shown in Figure 1. Each of the six system processors contains a general purpose 32-bit CPU. Each processor is capable of executing between 2 and 3 MIPS. Associated with each processor is a private local memory. This memory can be used for algorithm storage and data storage. The processors function asynchronously, running on independent internal clocks.

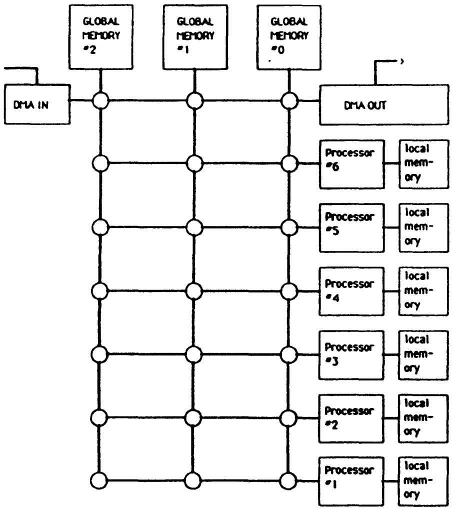


Figure 1 - MAPP System Architecture

Contained within the system are three global memories. The processors can each access these general purpose memories through a crossbar type port structure. Processor contention for access to global memory could cause a speed disparity among the processors if accesses are serviced in a completely random fashion. For the MAPP, speed disparity will not occur due to the discipline imposed by the global memory controller. The global memory controller prevents one processor from monopolizing global memory at any time. In order to access global memory, each processor must interface with the global memory controller. This controller contains a priority encoder that takes a "snapshot" of all the processor′s requests for access to global memory. If a processor fails to have its request honored in one snapshot it is guaranteed to have its request serviced in the next. There is no way for a processor to circumvent the action of the global memory controller.

At least 500 times a second, the MAPP system causes each processor to perform a self test. If a processor fails its own self test or if a processor detects a failure in another processor, a system-wide interactive test is performed. If a faulty processor is identified and confirmed during the system-wide test, it will be disabled automatically and the system will continue operation without it. This fault checking software, in addition to looking for actual

0190-3918/86/0000/0969 $01.00 © 1986 IEEE

hardware errors in each processor, requires that each processor respond within a certain period of time. This timed response checks a processors speed of operation and causes a processor to be flagged as faulty if it is operating below the speed of the other processors in the system. This self testing insures that each processor is operating at nearly the same speed.

## The Mutual Exclusion Method

The MAPP has several attributes that are useful when trying to float the master task. These attributes are both hardware and software based. First, each processor in the system is identical and runs at the same clock speed. The clocks are asynchronous, however. Second, the system periodically runs fault checking software that insures that each processor is operating normally. Third, in the MAPP system, the global memory controller insures that a processor cannot be arbitrarily delayed because of competition for memory accesses.

The devised mutual exclusion algorithm called Software Based Mutual Exclusion (SBME) uses a technique that can be categorized under the heading of busy waiting. Busy waiting is a general operation that requires a processor to loop several times while checking to see if another processor has acquired the master task. The SBME method uses a single location in global memory called MasterID as an indicator flag for the master task. If the master task is being performed by a processor, that processor´s ID number will be present at location MasterID. If no one is currently master, MasterID will contain a value of zero. The SBME algorithm is shown in figure 2 for the i´th processor of the system.

To acquire the master task, a processor reads location MasterID in global memory. If it is zero, the processor writes its ID number to MasterID immediately. Allowance is made for for the fact that another processor contending for the master task can write over the ID number in MasterID. The devised method requires that a processor loop and read MasterID several times until the danger of its ID being overwritten has passed. If the processor discovers a value in location MasterID different from its ID number, it drops out of contention. Any of the other processors that read MasterID after the first ID is written will avoid writing to it.

After a processor finishes the master task it clears MasterID in global memory. This allows another processor to assume the master task as soon as required. If a processor is ever stopped while operating outside the master task another processor would not be prevented from reserving the master task for itself.

The number of MasterID reads required for master task acquisition was determined, based on the number of contending processors. The variable m, in the SBME algorithm, is used to define the amount of time that a processor must wait before checking MasterID to see if it is master. The variable n determines the number of times that a processor must check MasterID before declaring itself as master.

```
processor i:

loop
   counter = 0

   read MasterID                ; Uninterruptable
   if MasterID > 0 exit         ; Section
   write ID to MasterID         ; of Code

 loop
   wait m cycles                ; See if anyone
   read MasterID                ; is trying to
   if MasterID  ne. ID exit     ; be Master
   increment counter
 until counter = n;

out: perform master task

   write 0 to MasterID          ; Allow another
                                ; to be Master

exit: do something else

endloop
```

Figure 2 - SBME Algorithm

A complex relationship exists between m and n to insure mutual exclusion. The relationship involves the memory speed of the global memory being accessed and the maximum number of processors that are represented in a global memory controller snapshot. A complete explanation of this relationship is beyond the scope of this short paper. Some discrete relationships between m and n are shown in Figure 3. The curves are shown for various delays of m and also for the total number of processors in the system. As m or the amount of delay between reads increases, the total number of reads, n, required to insure mutual exclusion decreases.

A trade off occurs when deciding upon the number of checks vs. delay periods that should be performed. The greater the number of checks, i.e. the more frequent the checking, the sooner a processor discovers that it is not master. However, more frequent checking also decreases the available bandwidth of global memory. Longer delays between checks would reduce memory bandwidth requirements but would tie up a processor longer when it was unsuccessful in acquiring the master task.

## Conclusion

Classical and contemporary mutual exclusion methods were reviewed as alternatives to the adopted method. Most were found to be unsuitable in that they imposed additional hardware requirements or are incompatible with the existing memory controller. One contending method, Dijkstra´s multiprocessor version of Dekker´s method, was suitable, but required longer acquisition time for present and expanded versions of the system.

The SBME method requires approximately 8.5 microseconds from the time a processor initially

```
               I
             3-I   *        m = 0.4 microseconds
 # OF          I              # CPU´s = 7 or more
  DELAY      2-I           *
INTERVALS      I
  OF   m     1-I           *
               I
             0-I           *
               I---I---I---I---I
                   1   2   3   4

                # OF CHECKS - n
```

```
               I
             3-I   *       m = 0.75 microseconds
 # OF          I             # CPU´s = 7 or more
  DELAY      2-I   *
INTERVALS      I
  OF   m     1-I           *
               I
             0-I           *
               I---I---I---I---I
                   1   2   3   4

                # OF CHECKS - n
```

```
               I
             3-I   *        m = 1.2 microseconds
 # OF          I              # CPU´s = 7 or more
  DELAY      2-I   *
INTERVALS      I
  OF   m     1-I   *
               I
             0-I           *
               I---I---I---I---I
                   1   2   3   4

                # OF CHECKS - n
```

Figure 3 - Worst Case Operation (7 or more Processors)

checks to see if anyone is master until the time it acquires the master task itself. As an alternate, the method patterned after Dijkstra took almost 49 microseconds. The primary reason for the time difference is that Dijkstra´s method for a best case required 15 accesses to global memory via read/write commands. A worse case for Dijkstra´s method could almost double the number of memory accesses required. The SBME method for a worse case requires only 6 read/write instructions. This difference is significant on the MAPP system, where read/write commands to global memory take twice as much time to execute as any other instruction.

The time required to acquire the master task in the MAPP system can be carefully controlled because of several factors:

1) The processors execute instructions at similar rates,
2) Each processor executes the same SBME code,
3) Instruction execution time is frequently tested,
4) The critical section of the SBME code, ie;

```
        read MasterID
        if MasterID > 0 return
        write ID to MasterID
```

is kept to a minimum and cannot be interrupted because all interrupts are disabled here.

Factors 3 and 4 as listed above require additional attention. The periodic system tests serve two purposes, they insure that the processors operating speeds are within limits and that if a processor fails as master that it can be disabled and a new processor can become master. When executing the mutual exclusion algorithm, a processor must not be interrupted between when it checks if someone is master and when it writes to MasterID. If a delay occurs here such as might result from a processor servicing an interrupt, one processor could assume the master task when it was not really available. An easy solution to this problem is to temporarily disable the interrupts while the critical section of code is being executed. If done properly, any interrupts that might have occurred during this time will still be pending and can then be serviced.

The SBME method always produces a "winner" whenever two or more processors vie for control of the master task. A processor´s value may be over written by another processor during the first part of the resolving routine. However, a valid ID number always remains in MasterID during conflicts. This ID belongs to the processor that ultimately acquires the task.

## References

Dijkstra, E.W., "Co-operating Sequential Processes", Programming Languages, edited by F. Genuys, New York, N.Y.: Academic Press, 1968.

Geary, B.T., "Mutual Exclusion in a Multiprocessor System," Masters Thesis, May 1985

Holt, R.C, Concurrent Euclid, the UNIX System and TUNIS, Reading, Mass.: Addison-Wesley Publishing Co., 1983.

Kuck, D.J., The Structure of Computers and Computations, New York: John Wiley & Sons, Inc., 1978.

Ries, R.H., et al.," Microcomputer Array Processor", Akron, Ohio: Goodyear Aerospace Corp., 1978.

# A PARALLEL EXECUTION SCHEME FOR EXPLOITING AND-PARALLELISM OF LOGIC PROGRAMS

*Yow-Jian Lin and Vipin Kumar*

Artificial Intelligence Laboratory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

## Abstract

In this paper we present a parallel execution scheme for exploiting AND-parallelism of logic programs. This scheme follows the generator-consumer approach of the AND/OR process model. Using problem-specific knowledge, literals of a clause are linearly ordered at compile time. This ordering and run-time binding conditions are then used to dynamically select generators and consumers for different variables at run time. The scheme can exploit all the AND-parallelism available in the framework of generator-consumer approach. Compared with other execution schemes based on the same approach, our scheme is simpler, and potentially more efficient.

## 1. Introduction

Exploiting AND-parallelism in the execution of logic programs is difficult when literals of a conjunct share variables. One possible solution is to evaluate every subgoal of a conjunctive goal in parallel, and then perform join operation on the solutions of all the subgoals. This approach would uncover all the AND-parallelism in a logic program. But, for most practical problems, this approach requires excessive computation and is suitable only if unlimited resources are available. Therefore, various approaches for exploiting AND-parallelism in a more limited form have been proposed [2] [5] [7] [10] [11]. In the generator-consumer approach introduced by Conery and Kibler in the context of AND/OR process model [4] [5], if more than one literal in a clause body share a variable $V$, then one of these literals is designated as the generator of $V$ and the remaining literals are designated as the consumers of $V$. The execution of any consumer of $V$ is not started until the execution of the generator of $V$ has finished. Conery and Kibler use an ordering algorithm to identify the generator of each uninstantiated variable. Each time a non-ground binding is generated, the ordering algorithm is applied again to identify a new set of generators. Note that the run-time overhead of setting up the AND-parallelism in their approach could be very high, as the ordering algorithm may have to be repeatedly executed while interpreting a logic program. In order to avoid the problem of excess run-time overhead, several schemes have been presented [1] [6]. PLM of Aquarius project at UC Berkeley, proposed by Chang, et. al. [1], makes use of static data-dependency analysis. The execution graphs and backtracking paths are all derived at compile time, based on the worst case activation of each predicate. The restricted AND-parallelism (RAP), proposed by Degroot [6], creates one execution graph expression for each clause at compile time. It then dynamically generates parallel execution sequences based on these execution graph expressions. However, since PLM makes use of compile-time worst case analysis and Degroot's RAP utilizes limited execution graph expressions, these two schemes, unlike Conery and Kibler's scheme, can not always explore AND-parallelism to the fullest extent allowable in the generator-consumer approach. This paper presents an execution scheme to achieve the same degree of AND-parallelism as it is exploited in Conery and Kibler's execution model, but with less run-time overhead.

## 2. Execution Scheme

In this execution scheme, we are dealing with pure Horn Clause logic programs. On top of that, we allow variable annotations. User may also suggest a partial order on the execution of literals in a clause. Unlike Concurrent Prolog [11], the variable annotations for shared variables are not absolutely necessary; i.e., it is possible that a variable is shared by many literals in a conjunct and none of its appearance is annotated. The suggested partial ordering is also for execution efficiency only. But the ordering would not affect the semantics of the logic program.

Similar to the AND/OR process model proposed by Conery and Kibler, our scheme also includes three algorithms: ***ordering***, ***forward execution***, and ***backward execution***. In Conery and Kibler's approach, the ordering algorithm creates the data-dependency graph for a clause when the head of the clause unifies with a goal literal. The graph is used to select generators and consumers for different variables until some literal in the body produces a non-ground binding for a shared variable, at which time a new data-dependency graph has to be generated. In our scheme, the ordering algorithm is run at compile time. Using problem-specific knowledge (such as variable annotations, programmer-suggested ordering, etc.), it merely specifies a linear ordering for literals of each clause [9]. Our forward execution algorithm uses this linear ordering and run-time binding conditions to choose generators for variables shared by more than one literal, and the data-dependency graph is generated implicitly as the clause is executed. A simple but intelligent backward execution algorithm has also been developed for deciding which literals to re-solve in case some literal fails [8] [9]. Due to limited space, we focus only on the forward execution algorithm in this paper.

### 2.1. Forward Execution Algorithm

Suppose that the literals in the body of a clause have been ordered, and $P_i$ represents the $i$-th literal in the linear sequence. In addition, $P_0$ stands for the head literal. During the execution, there exists a token for each variable in the binding environment. Conceptually, if a literal holds the token of an uninstantiated variable $V$, it simply means that the literal is currently designated as the generator of $V$.

When a clause is called and the clause has non-null body, an AND process, corresponding to the head literal $P_0$, is created. Tokens for different variables in the clause (including free variables in the clause body) are given to $P_0$.[1] After $P_0$ has successfully unified the head literal with the given goal, it creates several descendant OR processes, one for each literal in the body, and then executes Procedure $S_0$ to pass tokens to literals in the body. A literal $P_i$ becomes executable when it receives tokens for all of its variables in the current binding environment. If $P_i$ shares variables with any literal appearing later in the linear sequence, then after its execution, $P_i$ also invokes Procedure $S_0$ to pass tokens.

This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory at the University of Texas at Austin.

[1] For brevity, we will often use "$P_i$" to mean "the process corresponding to $P_i$".

**Procedure $S_0$**

$P_i$ does one of the following for each variable $V$ for which it has produced binding $T$.

i) **Variable V is bound to a ground term T.**
$V$ is substituted by $T$ in all the literals. No token has to be passed around.

ii) **Variable V is bound to a non-ground term T and T is dependent** *(i.e., there is another variable V1 in $P_i$ which is also bound to a non-ground term T1 where T and T1 share variables*[2]*).*
$V$ and $V1$ are substituted by $T$ and $T1$ in all the literals. For each new variable in $T$ which is not shared, a new token is created and passed to $P_m$ where $m = \min\{k \mid P_k$ had $V$ and $k > i\}$. Similarly, for each new variable in $T1$ which is not shared, a new token is created and passed to $Pn$ where $n = \min\{k \mid P_k$ had $V1$ and $k > i\}$. Tokens for new variables shared by $T$ and $T1$ are created and passed to $P_j$ where $j = \min\{k \mid P_k$ had $V$ or $V1$ and $k > i\}$.

iii) **Variable V is bound to a non-ground term T and T is independent** *(i.e., any variable in T is not shared by any other non-ground bindings produced for other variables of $P_i$).*
$V$ is substituted by $T$ in all the literals. The token for $V$ is then replaced by several new tokens, one for each new variable in $T$. These tokens are passed to $P_j$ where $j = \min\{k \mid P_k$ had $V$ and $k > i\}$.

Note that bindings of different variables can become dependent only if these variables are in the same literal. Therefore, the dependence checking for bindings can be done by each literal independently.

Depending on various binding conditions, different parallel execution sequence can be explored by this algorithm. An example is shown in Figure 1, where an arc with label $X$ means the token of variable $X$ has been sent along the direction of arc. Furthermore, a literal which has a dotted variable $V$ means that the literal holds the token of $V$. Given the same linear sequence of the literals (Figure 1a), this example shows two possible sequences of parallel execution. In the first sequence, the clause is called with bindings X/X, Y/Y, Z/Z. Token for X is passed to literal p2, token for Y is passed to p1, and token for Z is passed to p3. The binding of a free variable (such as W in this example) is always non-ground and independent in the beginning, therefore token for W is passed to p1. At this time both literals p1 and p3 have received tokens for all of their variables in the current binding environment, and can be executed in parallel (Figure 1b). Suppose that p1 generates bindings $y_0$/Y, $w_0$/W, and p3 generates binding $z_0$/Z. Since all new bindings are ground, no token has to be passed around. Both literals p4 and p5 are executable, as there are no variables in their current binding environments. Literal p2 is also executable, since it has received token for X from p0 previously. Therefore p2, p4, and p5 can all be solved in parallel (Figure 1c). The data-dependency graph corresponding to the first execution sequence is shown in Figure 1d.

In the second sequence, the clause is called with bindings V/X, V/Y, and $z_0$/Z. Since the bindings of variables X and Y are dependent (they share the same variable V), the tokens for X and Y are replaced by a single token for V, and the new token is passed to p1 (the first literal in the linear sequence which had either X or Y). The token for free variable W is passed to p1 again. This time literals p1 and p3 can be executed in parallel (Figure 1e). After the execution of p1, binding S/W and $v_0$/V is generated, and p2 and p5 can start executing in parallel (Figure 1f). After p2 generates binding $s_0$/S, p4 becomes executable (Figure 1g). The data-dependency graph corresponding to the second execution sequence is shown in Figure 1h.

## 3. Comparison With Other Approaches

### 3.1. Compile-time algorithm

At compile time, our scheme only applies the ordering algorithm once. Given problem-specific information represented as partial orders between literals, it is straightforward to apply the ordering algorithm to come up with a linear sequence of literals for each clause. Note that good linear sequences of literals are beneficial to other schemes as well. Conceptually, we are not doing extra work.

In PLM proposed by Chang, et. al. [1], the bindings of variables in a clause are classified into three types, where NGI means the binding is non-ground and independent, NGD means the binding is non-ground and dependent, and G means the binding is ground. Among these three types, NGD is worse than NGI, and NGI is worse than G. Programmers are asked to declare initial activation mode of each predicate, where the activation mode of a predicate $P$ is a possible binding condition of variables in $P$ when $P$ is activated. Using an iterative procedure, the compiler then derives the worst case activation of each predicate.[3] The data-dependency graph of each clause is then generated based on the worst case activation of its head literal. In order to provide reasonable initial activation mode of each predicate, programmers must be very much aware of what kinds of binding conditions can happen at run time. Moreover, the iterative procedure of doing data-dependency analysis tends to be time-consuming.

Degroot's approach [6] is even more complicated. The compiler is responsible for transferring each clause into an execution graph expression. However, in order to achieve all the AND-parallelism allowable in the generator-consumer approach, the execution graph expressions would have to be very complex and may have substantial run-time overhead due to redundant checkings. On the other hand, using simpler expressions may reduce the degree of AND-parallelism. Constructing a compiler which can produce reasonable execution graph expressions appears to be a nontrivial task.

### 3.2. Forward execution algorithm

Unlike other approaches, our scheme does not try to find if two literals are independent to be solved in parallel. Instead, our scheme just lets each literal decide whether it is executable. The parallelism comes out automatically when there are more than one literal ready to be executed.

The same clause

p0(X,Y,Z) :- p1(Y,W), p2(X,W), p3(Z), p4(W?,Z), p5(Y).

used as an example for the forward execution algorithm will be used again in the following to show advantages of our forward execution algorithm. We assume that the worst activation mode of p0 is

p0(NGD,NGD,NGD)

and the derived activation modes and exit modes[4] (based on the

[2] For simplicity, we are only considering the case in which two variables are bound to dependent terms. Extension to the case in which more than two variables are bound to dependent terms is obvious.

[3] The worst case activation, as defined in [1], refers to the binding condition where each variable has the worst type of its possible bindings.

[4] The derived exit mode of a predicate $P$ is the binding condition of variables in $P$ after the execution of $P$, based on a given activation mode. The derived activation mode of $P$ is the activation mode derived from the activation mode of the head literal and the derived exit modes of previously solved literals.

worst activation mode of p0) of p1 to p5 are

| **activation mode** | **exit mode** |
|---|---|
| p1(NGI,NGI) | p1(G,NGI) |
| p2(NGI,NGI) | p2(NGI,G) |
| p3(NGI) | p3(G) |
| p4(G,G) | p4(G,G) |
| p5(G) | p5(G) |

Note that if the activation mode of p3 is p3(G), its exit mode is also p3(G). The following discussion is based on our assumption of these activation modes and exit modes.

Suppose at run time when p0 is called, the bindings of variables X, Y, and Z are V/X, V/Y, and $z_1$/Z. Our forward execution algorithm will achieve the dynamic data-dependency graph shown in Figure 1h.

PLM of Chang, et. al. uses only one static data-dependency graph, which is generated at compile time based on the worst case activation. In our scheme, we take run-time binding conditions into account, and hence our scheme is able to exploit parallelism in cases where PLM would fail to do so. Figure 2 shows the static data-dependency graph used by PLM. Compared with Figure 1h, we can see that when the activation mode of p0 is "better", PLM fails to exploit possible parallelism.

Degroot's approach can explore different data-dependency graphs dynamically. However, due to the predetermined execution sequence of substatements it may not be able to start processing a subgoal as soon as the subgoal is ready to be executed. Suppose the following execution graph expression is produced by the compiler:

```
(GPAR(Y,Z)
     (SEQ         p1(Y,W)
                  (IF IPAR(Z,X)
                           (GPAR(W)  p2(X,W)  p4(W,Z))
                           (SEQ  p2(X,W)  p4(W,Z))))
     (IPAR(Z,Y)  p3(Z)  p5(Y)))
```

If at run time p0 is activated with the bindings V/X, V/Y and $z_1$/Z, then based on our assumption of activation mode and exit mode of each literal, the data-dependency graph of Figure 3 will be achieved by Degroot's approach. As we can see, the execution of p3(Z) is postponed even if it is executable from the very beginning. The execution of p5(Y) is also postponed due to the predetermined execution sequence of substatements. One of the drawbacks of these situations is that if the executable literals happen to fail, Degroot's approach will work longer in the failure branch of the solution tree.[5] Note that sometimes type checking could be redundant in Degroot's approach. For example, if GPAR(Y,Z) is true, then IPAR(Z,Y) is surely true. However since IPAR(Z,Y) has to be there to cover the case when GPAR(Y,Z) is false, the redundant type checking is unavoidable. Overall it appears that the run-time overhead in Degroot's approach is not much less than the overhead in our approach, even though Degroot's approach exploits only limited amount of AND-parallelism.

It is important to point out that, for any given clause, it is possible to create execution graph expressions in Degroot's approach which would be able to explore all the AND-parallelism allowable in the generator-consumer approach. Unfortunately, these expressions are too complicated, and would involve too many redundant run-time checkings.

Compared with Conery and Kibler's approach, our scheme as well as their scheme can explore AND-parallelism to the fullest extent allowable in the generator-consumer approach, as independent literals can always be executed in parallel. Moreover, if all the bindings produced by the generators are ground, then both schemes would achieve the same data-dependency graph for each clause. When generators can produce non-ground bindings, their scheme must execute the ordering algorithm every time a non-ground binding is generated, hence the data-dependency graphs achieved by both schemes could be different. Although repeated execution of the ordering algorithm may reorder the literals, it does not guarantee that a better data-dependency graph would be achieved. As we can see in Figure 4, during the execution of this particular example, the ordering algorithm must be executed three times (after p0 is unified with the goal and after the execution of both p1 and p2) by their scheme, but the resulting data-dependency graph is never better. When the heuristic rules used by their scheme are fallible, the new graph could be even worse. Furthermore, since the token passing scheme is simple, the run-time overhead of our algorithm in general is less than the overhead of Conery and Kibler's AND-parallel execution scheme.

In some cases, dynamically changing the linear sequence of literals could lead to better performance. However, when the choice of alternative linear sequences depends only on the activation mode of the head literal, techniques such as control alternatives used in IC-Prolog [3] would allow our scheme to use different optimal sequence for each possible activation. The only case in which dynamic reordering of literals can be helpful is when the linear sequence must be reordered with respect to the bindings produced by a generator in the clause body.

Our scheme also has the potential for distributed implementation, as the token passing scheme can be carried out by the corresponding process of each literal independently. In Conery and Kibler's approach, AND process must be responsible for coordinating the execution of conjunctive goals, and hence could become the communication bottleneck.

## 4. Concluding Remarks

We have presented a scheme of achieving limited AND-parallelism in logic programs. Compared with Conery & Kibler's approach, our scheme has less run-time overhead, and still achieves the same degree of AND-parallelism. Chang, et. al. [1] and Degroot [6] have also proposed schemes which reduce the run-time overhead presented in Conery and Kibler's scheme. However, these schemes tend to limit AND-parallelism, and require more complicated compile-time analysis than our scheme. An additional feature of our execution scheme is that the forward (and backward) execution algorithm can be implemented in a distributed manner. This is significant, as the centralized version of these algorithms can cause communication bottlenecks. We are in the process of designing message structures and communication schemes of the distributed version of these algorithms.

### References

1. Chang, J.-H., A.M. Despain, and D. Degroot, ''AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis,'' *Proceedings of the 30th IEEE Computer Society International Conference*, pp. 218-226, February, 1985.

2. Clark, K. L. and S. Gregory, ''PARLOG: A Parallel Logic Programming Language,'' Research Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, London, May, 1983.

3. Clark, K.L. and F. McCabe, ''The Control Facilities of IC-Prolog,'' in *Expert Systems in the Microelectronic Age*, ed. D. Michie, pp. 122-149, Edinburgh University Press, 1979.

[5] The approach of Chang, et. al. also has the same drawback.

4. Conery, J. S., "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Ph.D. Thesis, (Technical Report 204), University of California, Irvine, California, June, 1983.

5. Conery, J.S. and D.F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs," *New Generation Computing*, vol. 3(1985), pp. 43-70, OHMSHA,LTD. and Springer-Verlag, 1985.

6. Degroot, D., "Restricted AND-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 471-478, Tokyo, 1984.

7. Kasif, S. and J. Minker, "The Intelligent Channel: A Scheme for Result Sharing in Logic Programs," *Proceedings of the 9th IJCAI*, pp. 29-31, Los Angeles, August, 1985.

8. Lin, Y.J., V. Kumar, and C. Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," *to be presented at the Third International Conference on Logic Programming*, London, England, July, 1986.

9. Lin, Y.J., "A Parallel Implementation of Logic Programs," Ph.D. dissertation, The University of Texas at Austin, Austin, Texas, in preparation.

10. Lowry, A., S. Taylor, and S.J. Stolfo, "LPS Algorithms," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 436-448, Tokyo, 1984.

11. Shapiro, E.Y., "A Subset of Concurrent PROLOG and its Interpreter," Technical Report TR-003, ICOT, Tokyo, Japan, 1983.

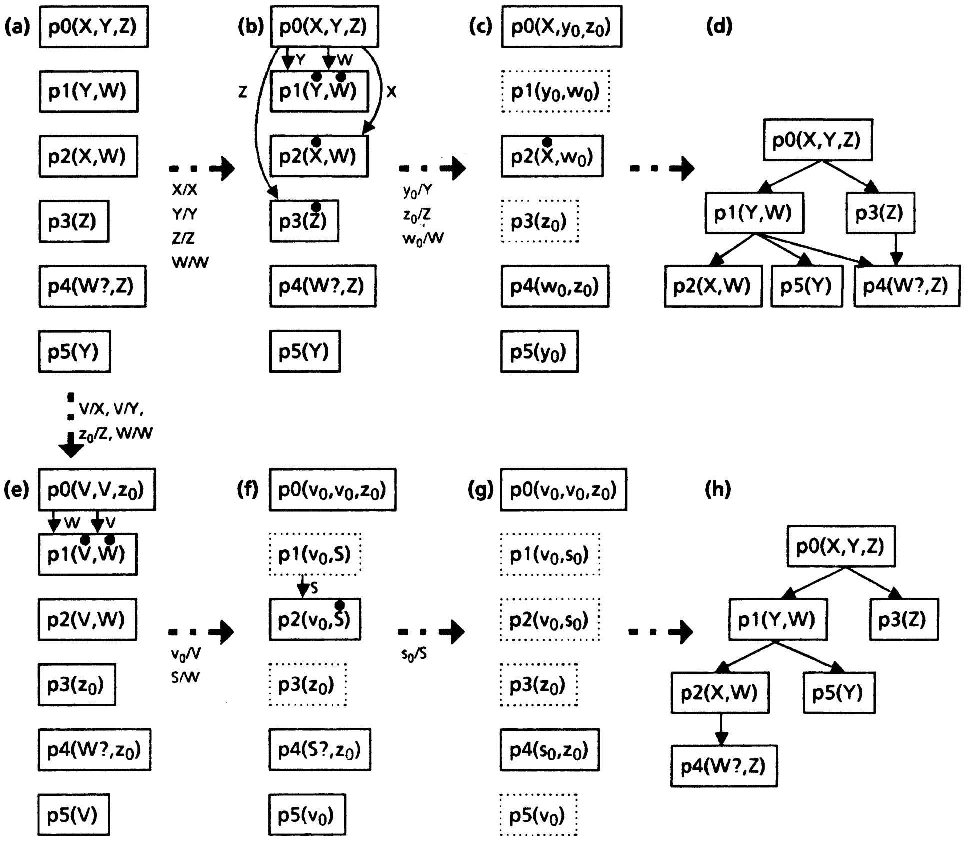


**Figure 1.** An example of the forward execution algorithm

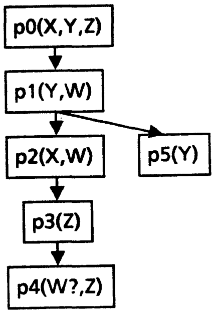


**Figure 2.** static data dependency graph of PLM

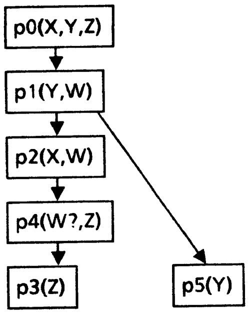


**Figure 3.** dynamic data dependency graph of Degroot's approach.

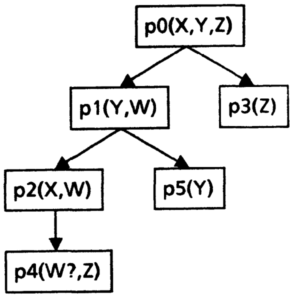


**Figure 4.** dynamic data dependency graph of Conery's approach.

# A Parallel Execution Model of Logic Program Based On Dependency Relationship Graph

Seungbeom Kim, Seungryoul Maeng, and Jung Wan Cho

Department of Computer Science, KAIST

PO Box 150 Cheongryang, Seoul, Korea

*ABSTRACT*

In this paper, a parallel execution model based on the Dependency Relationship Graph(DRG) is proposed. Compared with another approaches, the execution ordering of AND subgoal is determined by the role of the shared variable. The DRG is an execution ordering graph of AND subgoals and guarantees near maximal AND parallelism. The necessary information for constructing DRG is generated automatically in compile time and used in run time. By this automatic control technique, the logic is completely separated from the control and the original flexibility and the power of logic programs are maintained without programmer's burden of control.

## 1. Introduction

The logic language is a concurrent language in which parallelism is embedded naturally. This parallelism originates from the nondeterminism which is the most powerful characteristic of the logic language. However, it cannot be fully exploited under the typical von Neumann architecture because of the complicated control mechanism for the nondeterministic behaviour of the logic program. Such restriction makes the current logic languages less powerful and inflexible than the original language based on Horn clause logic.

There are various approaches to the parallel processing of logic programs which support the natural parallelism efficiently. Especially, AND, OR parallelism have been the key issues. Compared with the OR parallelism, the AND parallelism requires some sophisticated control.

To perform the AND parallelism easily, the mode concept and the variable annotation method have been widely used [1], [2], [4], [12]. Since these methods require programmer's control for the program execution, the logic language loses one of its original purposes. In other words, the logic is incompletely separated from the control.

When there is no programmer's control information, some heuristic algorithms have been adopted [1],[2],[4],[12]. These heuristics, however, may decrease the AND parallelism by the eager ordering of AND subgoals. The problems of the heuristic algorithms will be described in section 2.

In summary, these conventional methods have following disadvantages.

i) Programmer's burden of control when the mode or variable annotation are used.

ii) Inflexibility of program by unidirectional use of one argument as input or output.

iii) Expense of the completeness.

iii) Decrement of the potential AND parallelism by the eager ordering of AND subgoals when heuristics are used.

## 2. Some considerations of AND parallelism

AND parallelism can be considered as a synchronization strategy of the shared variables. Without the programmer's control information, the conventional method of achieving AND parallelism is based on the heuristic algorithms, such as the connection rule [4]. Some careful considerations about the role of the shared variable can extract more AND parallelism than the heuristic search algorithms. In convenience, we will use the C-Prolog syntax except the cut symbol.

### 2.1 Problems and solutions of AND parallelism

Given a clause such as
f(X) :- p(X),q(X)., there are two problems when p(X), q(X) are executed in parallel. One is the binding conflict which attempts to instantiate(bind) a shared variable X to two different values [5]. The other problem arises when one subgoal is a deterministic dependent of the other subgoal like coroutine, so the execution ordering of AND subgoals is required [4], [11].

At first, we will discuss the solutions of the binding conflict and propose a solution called Lazy Ordering. For solving the binding conflict, the join technique can be used. If a program has a deterministic search space, the join technique would exhibit the greatest parallelism as

0190-3918/86/0000/0976 $01.00 © 1986 IEEE

well as higher efficiency [5]. In spite of assuring the maximal parallelism, this technique results in the explosion of the search space and known as an inefficient solution [4]. That is, the nondeterministic characteristics of the logic program makes the join technique impractical. There have been used many heuristic search algorithms to avoid the explosion of the search space and to solve the binding conflict. In logic programs, the heuristic search means the selection criteria of AND subgoals(literals).

If a search algorithm can expand nodes that generate fewer descendants, then it might save itself needless work by traversing fewer unsuccessful branches [14]. This heuristic reduces the search space and solves the binding conflict. However, if it is adopted as a selection strategy in every execution of AND subgoals, it is efficient only when the parallel execution of them causes the search explosion and the AND parallelism would be decreased. If we can find such explosion points and the heuristic is applied only to these points, the AND parallelism is increased and the effort to use the heuristic algorithms is reduced. That is, the use of the heuristic algorithm in every execution step of AND subgoals is an eager ordering which reduce the possibility of the AND parallelism.

As another search heuristic, the selection by number of uninstantiated variables has been used. This heuristic assumes that AND subgoals with fewer uninstantiated variables will generate fewer branches. Based on this algorithm, Conery suggests Connection rule which is the main ordering algorithm of AND subgoals [4]. Although Connection rule can find some AND parallelism, it can be thought as a sort of an eager ordering and the possibility of AND parallelism still remains.

The shrinked AND parallelism of the two heuristics originates from the application of these techniques to every execution step(eager ordering). To extract more AND parallelism, we should invest our attention to find the point of search explosion. For example program in Fig. 2.1(a), if all AND subgoals are executed in parallel, they reach such clause that are made up of only ground unit clause, e.g. s(X,U), t(U,Z), q(Z,Y), r(X,Z) in (b). In this situation, s(X,U) and r(X,Z) have an instantiated argument(X=a) but t(U,Z) and q(Z,Y) have no instantiated value. The explosion of search space takes place in the execution of t(U,Z) and q(Z,Y) and these clause are called the explosion points. Since these explosion points cause the search explosion, the further execution is delayed until some solutions are transferred. The principle of the Lazy ordering is to proceed the execution as possible.

The second problem between two AND subgoals arises when one cannot be executed until another subgoal generates a solution of shared variables. If the execution ordering is changed, an infinite loop or any meaningless operations will be occurred. If there is no programmer's control information, the above two heuristics cannot order these AND subgoals properly, as the following procedure illustrates:

```
append3(A,B,D,E) :-
      append(A,B,C),append(C,D,E).
append([A|B],C,[A|D]) :-
      append(B,C,D).
append([],A,A).
```

Consider the goal
:- append3(A,[2],D,[1,2]).

The first AND subgoal of append3 is to be append(A,[2],C) and the second AND subgoal append(C,D,[1,2]). According to Conery's algorithm, the left append(A,[1],C) is executed first and results in an infinite loop. If we previously know that the first append clause forms an infinite loop, the execution ordering should be right to left. This automatic ordering technique is suggested by Naish and used in Prolog environment to check infinite loops [11]. This technique is also useful in parallel environment. We will extend this technique and use as a solution for the second problem.

### 2.2 Relationships between AND subgoals

To extract more AND parallelism than the heuristics, the dependency relationship between AND subgoals should be redefined according to the role of the shared variables in each subgoal. By the definition of AND subgoal relationship, we can classify the problems of AND parallelism and adopt the proper strategies, e.g. the Lazy ordering or loop detection technique, which solve them and guarantee more AND parallelism.

When any two subgoals share a variable, four cases can be considered. The first case arises when the instantiated value of the shared variable of one subgoal play a role of pruning the search tree of the other subgoal immediately ( Fig. 2.2a). The second case is similar to the first case but the pruning of the search tree is delayed for some time (Fig 2.2b). The third case arises when the shared variables is used as a data channel from one subgoals to the other subgoal, that is, the relationship of the producer and the consumer (Fig 2.2c). In this case the consumer will perform meaningless operation like infinite loop without producer's solutions about the shared variable. Finally, the system and the evaluable predicate may not be executed until the shared variables are sufficiently instantiated (Fig 2.2d). According to these observation, we can derive the following definitions.

Definition 1) If there is no shared variable between two subgoals or if shared variables are already instantiated at execution time, the relationship between two subgoals is defined as an independent relationship. Let I(p,q) be the relationship which represents the independent relationship between two subgoal p,q.

Definition 2) If two subgoals share variables and are in one of the following three conditions, then the relationship between them is defined as a deterministic dependent relationship (tightly coupled relationship).

i) When instantiated value of one subgoal about shared variables reduces the search space of the other subgoal immediately.

ii) When the execution of the subgoal is impossible like the system predicate and evaluable predicate without the instantiated value of shared variable.

iii) In recursive clause, when one subgoal cannot be executed until the other subgoal generates some solutions of shared variable because an infinite loop is created by constructing variable continuously or by the repeated recursive call without the status change of its arguments.

To represent deterministic dependent relationship between subgoal p and q, Dx(p,q) will be used in first case and Sx(p,q) in second and third case, where x = {x1, x2,...,xn} and xi is a shared variable of p, q.

Definition 3) If there are shared variables between two subgoal p, q and the relationship is not the deterministic dependent relationship, the relationship between two subgoal p, q is defined as nondeterministic dependent and is represented by Nx (p,q), where x = {x1, x2, ..., xn} and xi is a shared variable of p, q (loosely coupled relationship).

## 3. A parallel execution model based on Dependency Relationship Graph(DRG)

### 3.1 Automatic generation of the control information

To find the dependency relationship between AND subgoals without programmer's control information, the control information should be generated automatically. From this control information we can derive the necessary condition for the valid execution of a clause in compile time and the DRG in run time.

The control information is generated in compile time by the analysis of variable matching patterns. There are four patterns when an argument in a goal clause matches with a corresponding argument in a candidate OR clause. The matching patterns are in table 3.1 and the abbreviated forms will be used.

### 3.2 The necessary condition of a procedure

In the point of operational semantics, a logic program is a finite set of procedures and each procedure has a set of OR alternative Horn clauses. To invoke a procedure, we can consider the following necessary condition of the procedure.

i) Necessary condition of a procedure

The necessary condition of a procedure is the union of the sets, where each element is the necessary condition of a OR alternative clause. The empty set means that at least one argument should be instantiated to invoke the procedure.

When a procedure is called, this condition play a role of a guard to derive a valid solution. If the condition of a procedure is violated, it is reasonable to delay the execution of this procedure until some solutions are generated from other procedures of which the necessary conditions are satisfied.

ii) Necessary condition of a clause

The necessary condition of a clause is a set of tuples and each tuple contains the argument numbers of the clause.

When a system or evaluable predicate, the necessary condition is a set of single tuple and the tuple is composed of the argument numbers which should be instantiated for valid execution.

When a self recursive clause, the necessary condition of the body recursive clause is determined first and the necessary condition of the head clause is obtained from those of body clauses. The recursive clause may cause an infinite loop because some arguments are repeatedly constructed. In this case, we should discriminate the real constructive argument from the pseudo constructive argument. The real constructive argument appears in such a clause that there is no generator clause for the variables in the body recursive clause except the head clause ,so the execution always makes an infinite loop without the instantiated value of 'mg' argument. On the other hand, the pseudo constructive argument means that the infinite loop can be avoided because the generator for the variables of the body recursive clause exists and the status of recursive call is changed in every recursion.

When a ground unit predicate (fact), there is no necessary condition. In other cases like sort([X|U],S) in the following example, the necessary condition is obtained by the Algorithm 1.

These necessary conditions play a role similar to that of the mode declaration or the variable annotation, but provides more powerful and general methodology. The usage of them will be described in the section 3.3 and 3.4.

### 3.3 Static Data Dependency Graph(SDDG)

This section defines the Static Data Dependency Graph(SDDG) and an algorithm to obtain the necessary condition of a clause. The necessary condition of a clause can be made only after those of body literals are determined.

Definition 3.1) For a given clause A :-B1,B2,B3,..Bn. Static Data Dependency Graph(SDDG) is a graph over the literal set(={A,B1,B2, ...,Bn}) with labeled arcs by the set of variables in the clause. An arc(p,q) is labeled by V iff p <> q and V is the set of shared variables between p,q.

Definition 3.2) The entry node in a SDDG is defined as follows ; For a given SDDG, a node Bj is an entry node Bi iff the dependency relationship between Bi and A(head) is Sv(Bi,Bj) or iff the necessary condition is an empty set and the arc(A,Bi) is Nv(Bi,Bj) or Dv(Bi,Bj) for the non empty set V.

Given a SDDG, the entry node should have some shared variables with the a node, but the adjacent node to the head node is not always an entry node. If some input from the head node cannot satisfy the necessary condition of an adjacent node in SDDG, this node cannot be an entry node.

Algorithm 1 : the necessary condition generation algorithm

input :
a SDDG
the control information of body clauses
the necessary conditions of body clauses
output :
N - the necessary condition of the clause

[step1] Determine the dependency relationship : For each literal of the SDDG, determine the dependency relationship with adjacent literals. Assume that U is the necessary condition of Bj, V is the set of shared variables between Bi, Bj.

If a subset u of U is also a subset of V then the dependency relationship between Bi, Bj is Su(Bi,Bj). If a subset of v of V plays a role of the argument pruning of Bj, the dependency relationship is Dv(Bi,Bj). The argument pruning can be detected by the control information 'gm' in the position of the shared variable. In other cases, the relationship is Nw(Bi,Bj), where w= V - (u union v).

[step2] Make necessary condition set:

s0) make a test set S from the necessary condition of the entry points of the head node A. S={t|t is a tuple} and the element of the tuple is the argument number of A.

s1) Select an element t from S. Delete t from S. Search entry nodes of A which are satisfied by t. If S is empty, terminate this algorithm.

s2) For newly determined entry nodes, search their entry nodes.

s3) Repeat s2 until there is no entry nodes.

s3) If there is no isolated node, t is inserted in the necessary condition N. The isolated node means a unsatisfiable node because there is no arc which satisfies the necessary condition of the node.

s5) goto s1

Example 1 )

```
1) sort(X,Y):-
      perm(X,Y),ord(Y).
2) perm([],[]).
3) perm(Z,[X|Y]):-
      delete(X,Z,Z'),perm(Z',Y).
4) delete(X,[X|Y],Y).
5) delete(X,[Y|Z],[Y|U]):-
      delete(X,Z,U).
6) ord([]).
7) ord([X]).
8) ord([X,Y|Z]):-
      X <= Y, ord([Y|Z]).
```

The necessary condition of delete(X,Z,U) is {<2>,<3>} and that of delete(X,[Y|Z],[Y|U]) is also {<2>,<3>}. The necessary condition of ord([Y|Z]) and ord([X,Y|Z]) is {<1>}. However, the perm(Z',Y) in 3) has an empty set because there is no real constructive term. The necessary condition of the perm(Z,[X|Y]) and that of the sort(X,Y) is {<1>,<2>}, which can be obtained by the Algorithm 1.

### 3.4 Dependency Relationship Graph(DRG)

This section describes the abstract run time algorithm to construct the DRG of AND literals(subgoals).

Definition 3.3) For a given SDDG of A :- B1,B2,...,Bn., Dependency Relationship Graph(DRG) is a directed graph over the literal set(={A,B1,B2,...,Bn) with the labeled arcs by the dependency relationships between nodes. The dependency relationship is defined in the previous section.

```
Algorithm 2 : The DRG construction Algorithm
input :
 a SDDG
 t - the tuple in which
 the instantiated argument numbers of the head are.
 the control information of body clauses
 ni - the necessary conditions of body clauses
 N  - the necessary condition of the head clause
 C  - Node set of the SDDG
output : a DRG
```

[step 1] Test if t satisfies N, then search entry nodes of the head node A which are satisfied by t. Otherwise, select a node which receives the input by t and let it be an entry node. If t is an empty tuple and N is not empty set, select the left most literal node and let it be an entry node. Make directed arcs from A to each entry node Bi labeled with St(A,Bi). From C, delete such entry nodes that are satisfied by only t.

[step 2] For newly determined entry node Bi, B2... Bn, search their entry nodes E1, E2, ...En in C. Make directed arcs from Bi to the corresponding entry node Eis labeled with Sv(Bi,Ei), Nv(Bi,Ei), Dv(Bi,Ei), where v is the set of shared variable between Bi, Ei. From C, delete such Eis that are satisfied by only the shared variable with Bis.

In Fig. 3.1, example programs and DRGs of goal statements are illustrated.

### 3.5 A parallel AND/OR process model based on DRG

In our model, a goal statement is solved via message communications between AND processes and OR processes. An AND process is created for solving a goal clause and generates a DRG which is the execution ordering graph of body literals. The OR process is created to solve a body literal and the solution generated in OR process is not only sent to the parent AND process, but flows along DRG directly.

There are five types of messages; start message, wait message, fail message, solution message and end message. Compared with Conery's model, the wait message, the end message are added and the redo message, the cancel message are deleted.

The wait message is created in following three cases;

i) OR process --> parent AND process : When the Nondeterministic subgoals reach the clauses which need the instantiated value from the ancestor(Lazy ordering).

ii) OR process --> parent AND process : When the variable solution from the generator causes the violation of the necessary condition. In this case, the reordering problem is solved automatically by generating the wait message.

iii) parent AND process --> child OR process : When the OR process waits the solution from the preceding OR processes, this process receives a wait message from the parent AND process instead of a start message.

The end message is generated by OR processes when it cannot find any further solutions. This message is sent to the parent AND process and to the succeeding OR processes. The redo message is eliminated because there is no backtracking mechanism in our model.

Fig. 3.2 shows the message transfer diagram of an AND process and Fig. 3.3 shows the message transfer diagram of an OR process.

### 3.6 Analysis of the proposed model

The proposed model is a process model based on Conery's AND/OR process model. Compared with Conery's model, the AND parallelism is enhanced by using DRG and the backtracking mechanism is removed. Instead of the redo message, the solution is sent to the parent process or to the succeeding OR process directly and the end message is used to indicates the end of solutions. Fig. 3.4 shows the message generation in solving qsort([2,1],S,[]) by Conery's model and the proposed model (program in Fig. 3.1). We can obtain the following results.

The total # of messages = 34
The total # of processes = 13
The total # of time step = 38
In Fig. 3.4(b),

The total # of messages = 40
The total # of processes = 13
The total # of time step = 23

From these results, we can find that the parallelism of the proposed model increases 1.65(38/23) times because the qsort([1],S,[2|Y]) and the qsort([ ],Y,[ ]) are executed in parallel. In Conery's model, they are executed in sequential because of the shared variable Y. Since there is only one solution in solving qsort([1],S,[2|Y]) the redo message in Conery's model is not diffused over all procedure, so the proposed model generates more messages. However, if there are many solutions and the backtracking takes place, the number of messages in the proposed model can be less than Conery's model. We obtain the following results in solving sort([2,1],Y) which is the ordinary sort in Fig.3.1.

Conery's model
total # of messages = 83
total # of process = 29
total # of time steps = 50

the proposed model
total # of messages = 78
total # of process = 27
total # of time steps = 23

The parallelism increases 2.01(50/23) times and the number of generated messages are less than Conery's model. The increased parallelism results from the pipelining solutions between OR processes. The parallelism of the proposed model will increase in proportion to the number of solutions. The redo messages for backtracking increase the total number of messages in Conery's model.

## 4. Conclusion

In procedural oriented approach [6], the AND parallelism is more important than the OR parallelism because the OR parallelism can be achieved better by the Goal rewriting approach. If the AND parallelism is restricted, the advantages of the procedural approach is decreased. However, the nondeterministic behaviour of the logic programs makes it difficult to control the program and the conventional heuristic algorithms cannot perform the full AND parallelism because the explosion of the search space takes place.

In this paper, we proposes a method to control the nondeterministic behavior of logic programs and an AND/OR process model based on this method. Compared with the other approaches, the proposed model performs near maximal AND parallelism by finding the dependency relationships between AND subgoals in execution time. The dependency relationships between AND subgoals are determined by the role of the shared variable. In the proposed model, the backtracking mechanism doesn't take place and the solution pipelining between OR processes is used. This mechanism, also, increases the parallelism of the proposed model. As a future study, the run time overhead should be reduced by compiling technique.

## REFERENCES

[1] J.H.Chang and A.M.Despain, "Semi-intelligent Backtracking of Prolog Based on Static Data Dependency Analysis," 1985 Symposium on Logic Programming, July, 1985, pp. 10-21.

[2] K.Clark, F.McCabe, and S.Gregory,"IC-PROLOG Language Features," In Logic Programming, Academic Press, 1982, pp.234-266.

[3] K.Clark and S.Gregory,'"Notes on Systems Programming in PARLOG," Proc. of FGCS'84, Nov. 1984, pp.299-306.

[4] J.Conery,"The AND OR Process Model for Parallel Interpretation of Logic Programs," Technical Report 204, University of California, Irvine, Jun. 1983.

[5] D.Degroot,"Restricted And-Parallelism," Proc. of FGCS'84, Nov. 1984, pp.471-478.

[6] A.Goto, H.Tanaka, and T.Moto-oka,"Highly Parallel Inference Engine PIE-Goal Rewriting Model and Machine Architecture," New Generation Computing, Vol.1, No.1, 1984, pp.37-58.

[7] A.Ciepielewski and S.Haridi,"Execution of Bagof on the Or-Parallel Token Machine," Proc. of FGCS'84, Nov. 1984, pp.551-562.

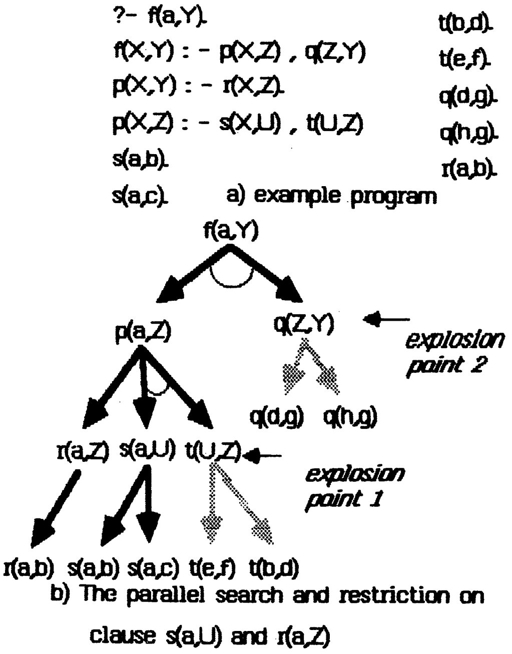
[8] S.Kasif, M.Kohli, and J.Minker,"PRISM: A Parallel Inference System for Problem Solving," Proc. of Logic Programming Workshop'83, Jul. 1983, pp.123-152.

[9] R.Kowalski, Logic for Problem Solving, North-Holland, 1979.

[10] H.R.Lee, S.B.Kim, S.R.Maeng and J.W.CHo, " A Parallel Execution Model of Logic Programs on Tree Structured Multiprocessor," Proc. of SCS'85, Dec. 1985, pp. 65-72.

[11] L.Naish,"Automatic Generation of Control for Logic Programs," Technical Report 83 6, Department of Computer Science, University of Melbourne, 1983.

[12] E.Shapiro,"A Subset of Concurrent Prolog and its Interpreter," Technical Report TR-003, ICOT, Jan. 1983.

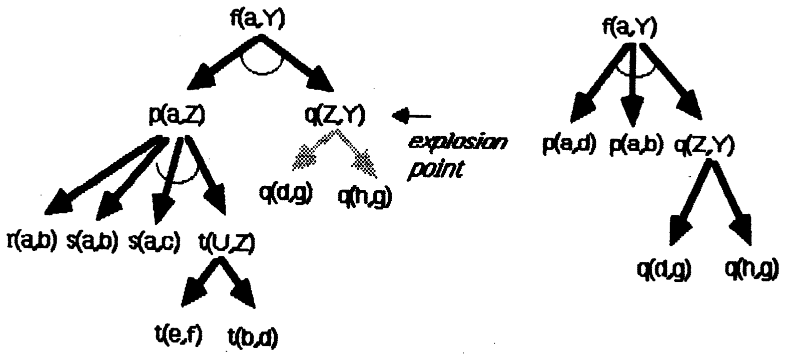


```
?- f(a,Y).                        t(b,d).
f(X,Y) : - p(X,Z) , q(Z,Y)        t(e,f).
p(X,Y) : - r(X,Z).                q(d,g).
p(X,Z) : - s(X,U) , t(U,Z)        q(h,g).
s(a,b).                           r(a,b).
s(a,c).
```

a) example program

b) The parallel search and restriction on clause s(a,U) and r(a,Z)

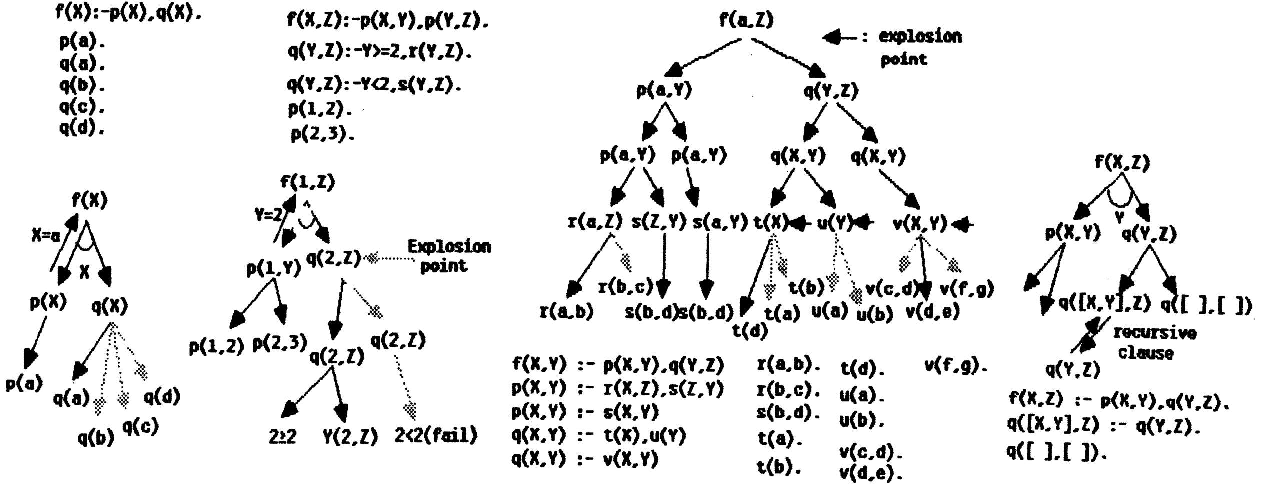
Fig. 2.1 Lazy ordering of an example program



c) Restriction on clause t(U,Z)
by the instamtiated value of
the shared variable U

d) Restriction on clause q(Z,Y)
by the instantiated value of
the shared variable Z

Fig. 2.1 Continued



```
f(X):-p(X),q(X).
p(a).
q(a).
q(b).
q(c).
q(d).
```

```
f(X,Z):-p(X,Y),p(Y,Z).
q(Y,Z):-Y>=2,r(Y,Z).
q(Y,Z):-Y<2,s(Y,Z).
p(1,2).
p(2,3).
```

```
f(X,Y) :- p(X,Y),q(Y,Z)     r(a,b).  t(d).     v(f,g).
p(X,Y) :- r(X,Z),s(Z,Y)     r(b,c).  u(a).
p(X,Y) :- s(X,Y)            s(b,d).  u(b).
q(X,Y) :- t(X),u(Y)         t(a).
q(X,Y) :- v(X,Y)            t(b).    v(c,d).
                                     v(d,e).
```

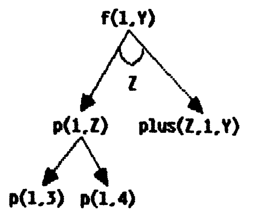
```
f(X,Z) :- p(X,Y),q(Y,Z).
q([X,Y],Z) :- q(Y,Z).
q([ ],[ ]).
```

a) Immediate pruning by the value of argument or evaluable predicate

b) lazy pruning

c)producer and consumer

Fig. 2.2 The role of shared variables



```
f(X,Y) :- p(X,Z),plus(Z,1,Y).
p(1,3).
p(1,4).
```

d) insufficiently instantiated system predicate

Fig. 2.2 Continued

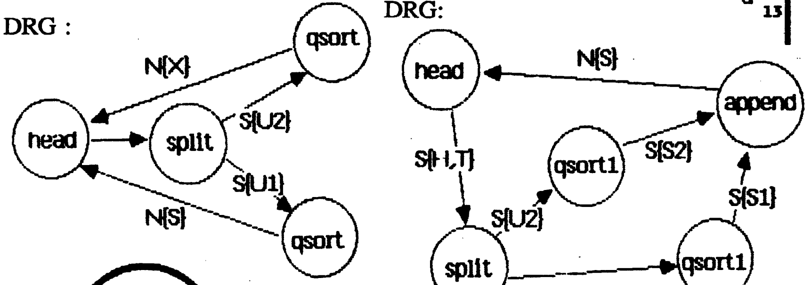
| an argument of a goal | corresponding argument of a candidate | name | abbreviated form |
|---|---|---|---|
| ground term | ground term | ground matching | gm |
| | variable term | ground matching | gm |
| | function term | ground matching | gm |
| | list term | ground matching | gm |
| variable term | ground term | ground matching | gm |
| | variable term | as general term | ag |
| | function term | function matching | fm |
| | list term | more general term | mg |
| function term | ground term | ground matching | gm |
| | variable term | as general matching | ag |
| | function term | as general matching | ag |
| list term | ground term | ground matching | gm |
| | variable term | as general matching | ag |
| | list term | as general matching | ag |
| | | more general matching | mg |

table 3.1 variable matching patterns and their name

```
qsort([H|T],S,X) :-                     N={<1>}
              split(H,T,U1,U2),         n1={<2>,<3>,<4>}
              qsort(U1,S,[H|Y]),        n2={<1>}
              qsort(U2,Y,X).            n3={<1>}
qsort([],X,X).

qsort1([H|T],S) :-                      N={<1>,<2>}={}
               split(H,T,U1,U2),        n1={<2>,<3>,<4>}
               qsort1(U1,S1),           n2={}
               qsort1(U2,S2),           n3={}
               append(S1,[H|S2],S).     n4={<1>,<3>}
qsort1([],[]).
split(H,[H1|T1],[H1|U1],U2) :-          N={<2>,<3>}
               H1<=H,                   n1={<1,2>}
               split(H,T1,U1,U2).       n2={<2>,<3>}
split(H,[H1|T1],U1,[H|U2]) :-           N={<2>,<4>}
               H1>H,                    n1={<1,2>}
               split(H,T1,U1,U2).       n2={<2>,<4>}
split(_,[],[],[]).

append([],L,L).
append([E,L1],L2,[E|L]):-               N={<1>,<3>}
               append(L1,L2,L).         n1={<1>,<3>}
```

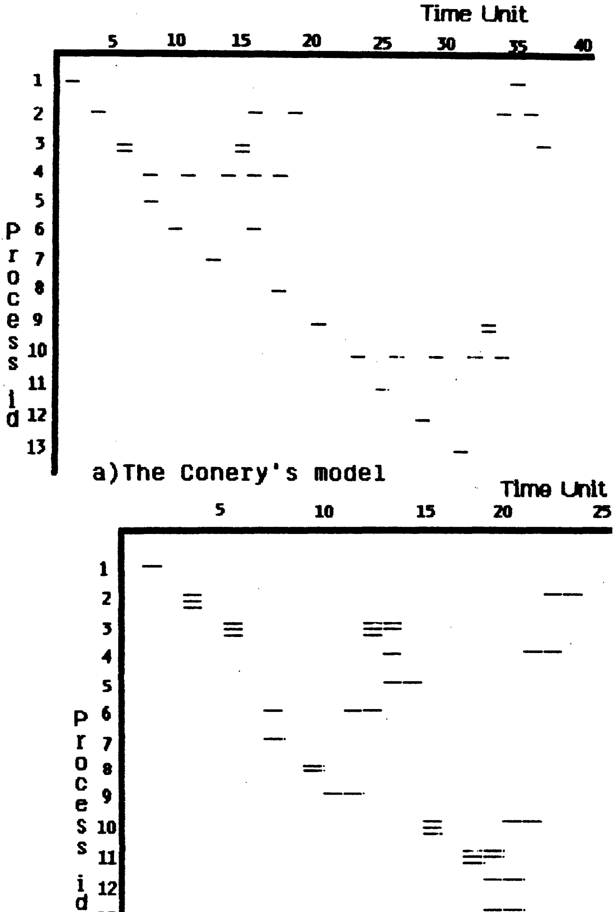
a) two quicksort programs and control informations

?- qsort([2,1],S,[]).     ?- qsort1([2,1],S).



b) DRG of each goal statements

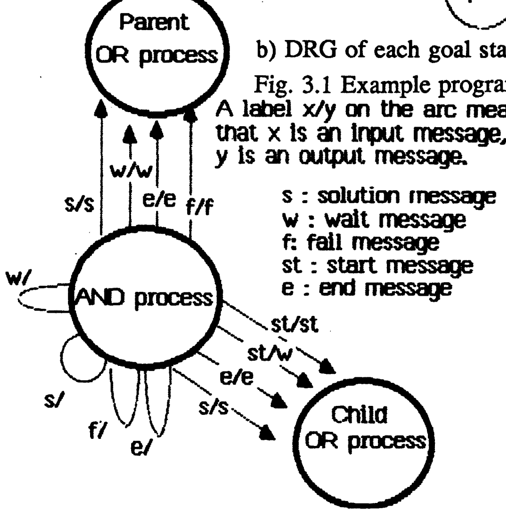
Fig. 3.1 Example programs and their DRGs



a)The Conery's model

b) the proposed model

Fig. 3.4 Message Generation Diagram of qsort([2,1],S,[]).



A label x/y on the arc means that x is an input message, y is an output message.

Fig. 3.2 Message transfer diagram of an AND process

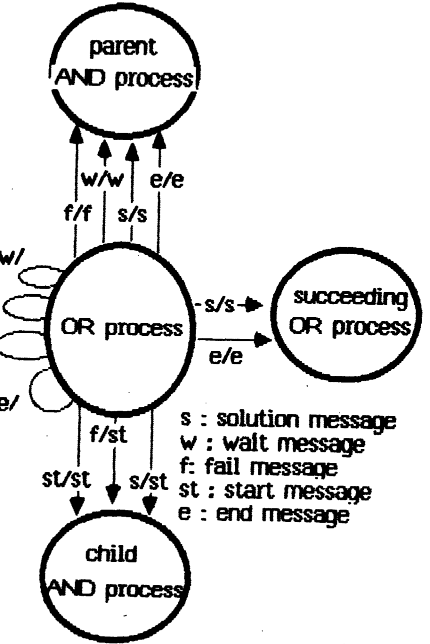


Fig. 3.3 Message transfer diagram of an OR process

# DETECTION OF AND-PARALLELISM IN LOGIC PROGRAMMING[a]

Yu-Wen Tung and Dan I. Moldovan

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0781

**Abstract** -- In this paper, we propose an ordering algorithm based on the automatic detection of AND-parallelism in logic programming at compile time. This algorithm consists of three phases: (1) analysis and detection of input-output mode, (2) automatic detection of data dependencies, and (3) subgoal ordering. Existing algorithms for the analysis of AND-parallelism proposed by previous researchers rely on mode declaration by the programmer. Based on the ordering scheme proposed in this paper, parallel compilers for logic programs can be conceived. A performance analysis of the ordering algorithm is presented through several examples.

## 1. Introduction

In this paper, automatic detection of parallelism in logic programming is studied. Definitions of technical terms employed in this paper can be found in [1,11,12,13,19].

There are four kinds of parallelisms in Horn clause logic programming identified by Conery and Kibler [4], Ito and Masuda [9], Pollard [16] and other researchers. They are (1) OR-parallelism, (2) AND-parallelism, (3) stream parallelism; and (4) unification parallelism. Among these parallelisms, OR- and AND-parallelisms are more important than others. OR-parallelism means to execute alternative clauses simultaneously, and AND-parallelism means to execute many subgoals in a clause simultaneously. OR-parallelism is much more straightforward then AND-parallelism because there is no interaction between the processing paths. On the other hand, when two subgoals in a clause share some variables, the answers obtained by executing these subgoals simultaneously may conflict each other. Thus, AND-parallelism must be carried out carefully.

For realization of AND-parallelism, various schemes had been proposed [2,5,6,7,15,16] and they fall into one of the following two principles:

1. Whenever there is an output variable shared by two subgoals, these subgoals are executed in sequence. This is called none-shared-variable (NSV) scheme in [19], and most of the current works fall into this category, such as Conery's scheme [5].
2. Execute all subgoals simultaneously, and than "reconciliate" possible conflicts via extra unification. This is called reconciliation (REC) scheme in [19], and the work of Pollard [16] is an example.

According to the current research, the NSV scheme seems to outweigh the REC scheme for applications with strong declarative inference, such as knowledge-based expert systems [19]. Also, it is known that proper subgoal ordering is important in the NSV scheme [2,5,6,7,19]. However, most of these works rely on "mode declaration" to implement ordering for AND-parallelism [2,5,6,7]. The "mode declaration" is a programmer-provided control command which tells the machine the binding transfer direction of a variable in a clause. Although this approach is efficient, its machine-oriented nature is not desirable in a declarative language such as logic programming. In this paper, we propose the compile time ordering algorithm without programmer-specified "modes."

## 2. Concept of Data Dependencies

This work is based on the concept of input-output mode of each argument in a given subgoal and the data dependency relationships among subgoals, as explained below:

### 2.1 Input and Output

The concept of *input* and *output* in logic programming was first introduced by Kowalski [11]. It denotes the direction of the binding transfer during unification, in a way similar to the input and output arguments in procedure calling. Here we extend this concept to binding transfer through substitution and subgoal computation as well as unification. For simplicity, let $C^+$ denote the clause head of C, and $C^-$ denote the clause body of C: $C^+ \leftarrow C^-$. Consider the goal

$$\leftarrow A_1, \ldots, A_i, \ldots, A_k$$

where the selected atom $A_i$ is unified with some $C_i^+$, with a *most general unifier* (mgu) $\theta$. Any variable X in $C_i$ which is instantiated by $C_i\theta$ is called an ***input variable*** of $C_i$. Such an X is also called an ***input variable*** of atom $B_j$, $B_j \in C_i^-$. If an uninstantiated

[a] This research was supported by NSF Grant ECS-8307258, JSEP Contract F49620-85-C-0071.

variable Y in $C_i^+$ becomes instantiated only when some subgoal $B_j$ in $C_i^-$ is computed, then Y is called an ***output variable*** of $C_i^+$, and of $B_i$.

If a term has only input variables, it is called an ***input term***. Similarly, if it has only output variables, it is called an ***output term***, and a ***mixed term*** or ***i-o term*** if it has both. An example of a mixed term is f(X,b) and f(a,Y) with a unifier $\theta = \{X/a, Y/b\}$.

Although the computation of logic programs can be viewed as computing output arguments from given input arguments for each subgoal, the input-output mode of any given argument is not predetermined in any subgoal due to the fact that subgoals are "relations" rather than "functions." As a result, when two subgoals share the same variable X, they may not be executed at the same time if X is an output variable. This is due to a possible binding conflict. However, they may be executed simultaneously when X is an input variable to both of them. We say there exists some data dependency relation in the first case, but not in the second one. Below, four types of data dependencies in logic programming are defined, based on the concept of input and output.

## 2.2 Data dependencies

Consider a clause

$$p \leftarrow q_1, q_2, \dots, q_k$$

let v(p) be the set of variables that occur in p, $v(q_i)$ be the set of variables that occur in $q_i$, $1 \le i \le k$.

### Definition 2.2.1 Functional dependency:

We say there is a ***functional dependency*** between $q_i$ and $q_j$ if there exists a variable $x \notin v(p)$, where x is an output variable of $q_i$ and an input variable of $q_j$. The atom $q_i$ is called the ***producer*** of x and $q_j$ is called the ***consumer*** of x. The variable x is an ***intermediate result*** generated by $q_i$ and consumed by $q_j$.

### Example 2.2.1:

The following clause expresses the calculation of Fibonacci-numbers:

F(x+2,n) ← F(x+1,n1), F(x,n2), Plus(n1,n2,n)

where F(x,n) is a Fibonacci procedure that computes result n from a given input x, and Plus(n1,n2,n) is the addition procedure that adds two numbers n1 and n2, returning the result n. Clearly, the addition of n1 and n2 must be performed after F(x+1,n1) and F(x,n2) have been evaluated. Therefore, Plus(n1,n2,n) is *functionally dependent* on F(x+1,n1) and F(x,n2).

### Definition 2.2.2 Internal dependency:

We say there exists an ***internal dependency*** between $q_i$ and $q_j$ if there is a variable $x \in v(q_i) \cap v(q_j)$ and x is output in both $q_i$ and $q_j$.

### Example 2.2.2:

Assume a company is hiring people who must have engineering degrees plus five years or more experience each and with salary requirements no more than 50K a year each. The top level program could be written as:

candidate(x) ← degree(x), engrg(x), exp(x,5),
salary(x,50)

with a goal ← candidate(x). Clearly, x is an output variable and internal dependencies exist between any pair of atoms in the program clause body, because they all share variable x.

Next, the concept of the *coupling effect* is introduced. This effect makes two variables bearing different variable symbol names equal and can be viewed as one variable.

### Definitions of the coupling effect:

1. Two variables are said to be ***coupled*** if they become identical non-ground terms after a unification. For example, x and y are coupled if p(x,y) is unified with p(h(w),h(w)) or p(z,z).
2. Two variables are said to be ***coupled*** if their corresponding terms share one or more common variables after a unification. For example, x and y are coupled if p(x,y) is unified with p(z,f(z,w)).
3. Two variables are said to be ***coupled*** if they are unified with a pair of coupled variables. For example, if z and w are known as coupled, then x and y are coupled when p(x,y) is unified with p(z,w).

We can define ***coupled term*** in the same way. The phrase "coupled term" was introduced by Chang, Despain and DeGroot [2].

### Definition 2.2.3 Coupling dependency:

We say that there exists a ***coupling dependency*** between $q_i$ and $q_j$ if $x \in v(q_i)$, $y \in v(q_j)$, and x and y are output variables and are coupled.

Coupling dependency is a variation of the internal dependency. Coupling dependency cannot be discovered only by checking the given clause, hence it is also called ***hidden internal dependency***.

**Example 2.2.3:**

Consider the program:

C1: $p_0(x) \leftarrow p_1(x,x)$

C2: $p_1(y,z) \leftarrow p_2(y), p_3(z)$

All variables are assumed to be output variables. It is obvious that neither C1 nor C2 has functional or internal dependency. However, when $p_0$ is called, $p_1(x,x)$ is unified with $p_1(y,z)$ and y, z are coupled as a result. Therefore $p_2(y)$ and $p_3(z)$ have a coupling dependency. The coupling effect in C2 is introduced by a higher level goal in C1.

**Definition 2.2.4 Run-time dependency:**

We say there exists a ***run-time dependency*** between $q_i$ and $q_j$ if $x \in v(q_i)$, $y \in v(q_j)$, and x and y are output variables and are coupled at run time by a top level goal.

Run-time dependency is a coupling dependency introduced at run time, due to a goal with coupled terms (eg. p(x, x)). It cannot be detected directly from the program, and hence, is different from coupling dependency.

**Example 2.2.4:**

Consider the program:

$p_1(y,z) \leftarrow p_2(y), p_3(z)$

where the goal is $\leftarrow p_1(x, x)$, and x, y, z are output variables. In this example, $p_2(y)$ and $p_3(z)$ seem to be independent but are coupled at run time by x in the goal $\leftarrow p_1(x, x)$.

This is just a variation of Example 2.2.3 with a higher level goal $\leftarrow p_1(x ,x)$ unspecified at compile time.

Among the four types of data dependencies, internal dependency, coupling dependency and run-time dependency are similar to each other in the sense that all three are defined over *output variables*. They either denote some output variables shared by two subgoals or some output variables coupled with each other, and thus we also call them ***output dependencies*** as opposed to *functional dependencies*, which are defined over both input and output variables.

With the above definitions, it is clear that functional dependency represents inherent sequentiality and implies a partial ordering of subgoals. This partial ordering is "necessary" in the sense of implementation and not the logic. On the other hand, output dependency leads to many possible orderings, one more efficient than another. Together, these dependencies describe the limitation of the AND-parallelism and they are important in detecting the available parallelism. Below, some techniques of detecting such limitation are proposed.

## 3. Automatic Parallelism Detection Techniques

In this section we study techniques of designing a parallel compiler which detects AND-parallelism automatically. This compiler has three phases: in the first phase, it analyzes the input-output mode of arguments in a given program; in the second phase, it finds out the data dependency relations of the subgoals in each clause; and in the third phase, it determines an efficient execution ordering based on the data dependency and other supplemental criteria. This is described in detail below.

### 3.1 Automatic input-output mode analysis

Because the "use" of a relation is not deterministic, the binding transfer direction of a variable in the relation is not always fixed. However, if some binding transfer direction is proven possible (or impossible) in a given program, the binding transfer path can be traced and the binding transfer directions for other variables can be inferred. Several heuristic rules for inferring whether or not a given variable is input are proposed below. In describing the rules, consider a variable X and a subgoal q in either the head or the body of a clause C. By notation "$X \in q$" we mean "X occurs in q", and by "q<*r*>=<input>" we mean the *r*th position of q must be an input position.

The rules are:

1. Arithmetic and logic operations rule: arguments that *must be instantiated* as required by PROLOG systems should be input. For example, both X and Y are inputs in expressions "X * Y" and "X < Y."
2. Body producer-consumer rule: if $X \notin C^+$, $X \in C^-$, and X has two or more occurrences in $C^-$, then at least one position containing X must be input, and at least one must be output.
3. Head producer-consumer rule: if $X \in C^+$, $X \notin C^-$ and X has two or more occurrences in $C^+$, then at least one position containing X must be input.
4. Don't care rule: if a clause D is called by C, and D has a don't care symbol ("_" in PROLOG), then the corresponding position X in C must be input.
5. Variable consistency rule: if $X \in C^+$ and $X \in C^-$, and if all occurrences of X in $C^-$ are *inputs*, then at least one occurrence of X in $C^+$ must be input.

6. Position consistency rule: if predicate q in $C^+$ is also in $C^-$, then C is called a *recursive clause*, and the corresponding position of all q's must be consistent, i.e., if position *r* in any such q is known as input then q<*r*> must be input.
7. OR-bundle rule: if a set of $\{C_i\}$ is an OR-bundle -- i.e., all $C_i^+$ have the same predicate name and the same arity, and if a position in any $C_i^+$ is an input, then the position of all $C_i^+$ must be input.
8. Coupling rule: if variable X occurs in an atom q (either in $C^+$ or $C^-$) more than once and both are output variables, then X is a coupling variable.

The analysis of input-output modes for a given OR-bundle based on these rules is shown in Algorithm 1 below:

### ¶ Input-Output Modes Analysis Algorithm 1:

*Input* : A set of clauses which is an OR-bundle.
*Output* : All possible input-output modes of each arguments in the clauses.
*Method* :

1. Mark any known modes for variables in given clauses.
2. Select any variable with its mode unknown, apply Rules 1 through 8 to this variable. Mark <i> for variables that *must be input.* If there is ***at least one input*** in a set of positions, state "$p_1$<1> or ... or $p_r$<r> = <i>."
3. When no rule is applicable, mark <o> for those ***must be output*** terms and mark <i/o> for the rest.

<End of algorithm 1>

Based on this algorithm, the entire logic program can be analyzed. We may view the logic program as a directed graph, each clause is a node and each node has pointers to the nodes it calls. There is a cycle if a clause calls itself -- i.e. a recursive call. A root node is a clause that is not called by any other clause except the top level goal, and a terminal node is a unit clause that does not call any other clauses. A descendent node of node p is a node pointed to by p, but not p itself. Starting from the terminal nodes, we can analyze the clauses from bottom up until the root is found. This is shown in the next algorithm:

### ¶ Input-Output Modes Analysis Algorithm 2:

*Input* : A logic program.
*Output* : All possible input-output modes of each arguments in the program.
*Method* :

1. Determine the calling sequence of the logic program. Select a clause that does not have any descendent clause unchecked.
2. Use Algorithm 1 to analyze the input-output modes of arguments in that clause.
3. Go to Step 1 if there is any clause remains unchecked, exit otherwise.

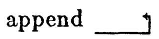
<End of algorithm 2>

**Example 3.1.1 "append":**

C1: append([], L, L)
C2: append([x|L1], L2, [x|L3]) ← append(L1, L2, L3)

**Analysis**:

The calling sequence is trivial:



Apply Rule 3 tod x in C2:
  append<1> or append<3> = <i>
Apply Rule 3 to L in C1:
  append<2> or append<3> = <i>
Apply Rule 7:
  [case i] append<3> = <i>
  [case ii] append<1> = append<2> = <i>
  other cases append<1> = append<3> = <i>
  or append<2> = append<3> = <i>

  are subsumed in [case i].
therefore, append<1,2,3> = <i/o,i/o,i> or <i,i,i/o> and the analysis is done.

**Example 3.1.2 "matrix multiplication":**

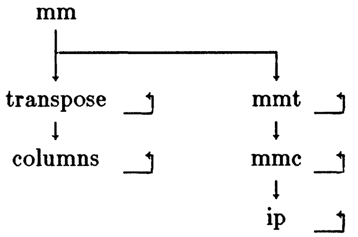
The following matrix multiplication program example is taken from Conery's thesis [5] (pp.122-123 and p.80):

C1: mm(a, b, c) ← transpose(b, bt), mmt(a, bt, c)
C2: mmt([], _ , [])
C3: mmt([a1|an], b, [c1|cn]) ← mmc(a1, b, c1),
                                mmt(an, b, cn)
C4: mmc (_ , [], [])
C5: mmc(a, [b1|bn], [c1|cn]) ← ip(a, b1, c1),
                                mmc(a, bn, cn)

```
C6:   ip([], [], 0)
C7:   ip([a1|an], [b1|bn], c) ← ip(an, bn, x),
                                 c is x + a1 * b1
C8:   transpose([[]|_], [])
C9:   transpose(m, [c1|cn]) ← columns(m, c1, rest),
                                 transpose(rest, cn)
C10:  columns([], [], [])
C11:  columns([[c11|c1n]|c], [c11|x], [c1n|y])
           ← columns(c, x, y).
```

**Analysis**:

the calling sequence is:



we may start from either "ip" or "columns", say "ip":
ip (C6 & C7):
Apply Rule 1 to the last atom of C7,
x, a1 and b1 are *input variables.*
Apply Rule 2, x in "ip" in C7 is *output.*
Apply Rule 6, since
a1 and b1 in $C7^{+}$ are *input variables*, so
an and bn in $C7^{-}$ are *input variables*,
ip<1,2,3> = <i,i,o>
Similarly, we obtain the following step by step:
mmc (C4 & C5): since ip<1,2> = <i,i>,
mmc<1,2,3> = <i,i,i/o>

mmt (C2 & C3): since mmc<1,2> = <i,i>,

mmt<1,2,3> = <i,i,i/o>

columns (C10 & C11):
columns<1,2,3> = <i,i/o,i/o> or <i/o,i,i>

transpose (C8 & C9):
transpose<1,2> =<i,i/o>

mm (C1):
mm<1,2,3> = <i,i,i/o>
and the analysis is done.

The above analysis shows that if "mm" is the top level goal, the first two positions must be input, and the third can be either input (verify if c is the product of a and b) or output (obtain c as the product of a and b). Also, since variable "bt" is an output from "transpose" and an input to "mmc," we know the producer-consumer pair must be <transpose, mmc>. This cannot be inferred by Conery's ordering algorithm. In fact his algorithm will obtain a wrong ordering in this example (see p.122 in [5]).

By performing the input-output analysis we can actually obtain all possible usages allowed in the program. For example, if the "transpose" in Example 3.1.2 is a top level goal, the second position of "transpose" need not be an output as in the case when "mm" is the top level goal. In this case, "transpose" can also be an input such that "transpose(b, bt)" means *verify if* bt *is the transposition of* b. This usage is allowed by the logic of the program.

Although it is difficult to prove that this algorithm guarantees to find all terms which must be input, it is clear that if an input term is erroneously taken as output, the penalty is only a reduced parallelism. The algorithm is very precise when applied to many example programs nevertheless.

### 3.2 Determination of data dependencies

The second phase of parallelism detection is the determination of data dependencies. Since all data dependencies are defined over input and output, once the input-output analysis is done, the detection of data dependencies becomes easy. Even the run-time dependency may be "guessed" at the compile time, although there are certain limitations.

Detection of functional dependency and internal dependency is trivial when the input-output modes are known, because both dependencies can be readily identified by their definitions based on these modes. If the mode of a term is "mixed," then both possibilities of input and output should be considered.

In the case of coupling dependency, a recursive check of subgoal invoking is required to find the "source" of coupling effect. This can be done by adding one procedure to the input-output mode analysis Algorithm 1: mark <c> on each coupled term found by the coupling rule; also mark <c> on each term related to known coupled terms found by the variable consistency rule and the position consistency rule.

In the case of run-time dependency, if there are only a few possible combinations of modes for the top level goal, then the compiler can figure several possible run-time dependencies which can be verified easily at run time. Below, we list some criteria useful in detecting the run-time dependency at compile time. By a *potential output term* is meant a term that is either an output term or a mixed term. The criteria are:

(i). When there are none or one potential output term in a goal, no run-time dependency can exist.

(ii). If there are two potential output terms, two possibilities exist -- one is that they are coupled at run-time and the other is that they are not. Both cases can be analyzed at compile time each leading to one possible ordering selected at run-time.

Since the number of possible mode combinations grows exponentially with the number of potential output terms **r**, this approach is not justified when **r** is large. However, most real world programs seem to have only a few potential output terms in the goal clause, and the compile time analysis is usually sufficient.

### 3.3 Subgoal ordering techniques

In the case of functional dependency, partial ordering is obtained at the time the producer-consumer dependency is detected. But in case of output dependencies, there is flexibility in determining the ordering, and which ordering leads to the highest efficiency is yet to be determined. Therefore we need some supplemental criteria, namely *groundability* and *subtree complexity*, for this purpose.

Consider an output shared variable X in a clause C as shown below:

$$p(X) \leftarrow q_1(X), q_2(X), ..., q_r(X)$$

Because of the output dependency due to X, only one atom can be selected from $q_1(X)$ through $q_r(X)$ for processing. Assume $q_i(X)$, with the substitution $\theta_i = \{X/t\}$, is the one selected.

If $\theta_i$ is a ground substitution, variable X in other $q_j$'s $(1 \leq j \leq r,\ j \neq i)$ is changed from output to input after $q_j$ is executed, and the data dependencies due to X between these $q_j$'s are removed. As a result, $q_j$'s can then be processed in parallel. Conversely, if $\theta_i$ is a non-ground substitution, the remaining $q_i$'s are still data dependent and they cannot be processed in parallel. Therefore, the key to the "efficiency" is to find a $q_i$ such that it has a ground substitution. If no such $q_i$ exists, a $q_i$ with $\theta_i$ that has more ground substitution components should be selected prior to another $q_j$ which has fewer such components. The measure of how close a substitution is to the ground substitution is informally called ***groundability***.

Consider the following clause as an example:

$$grandparent(X,Y) \leftarrow parent(X,Z), parent(Z,Y)$$

with a goal ← grandparent(X,peter). A left-to-right ordering will find all parent-child pairs first, then test for parent(Z,peter) for validity, rather than trying to find the parent of parent of Peter. Obviously, the better ordering is to do parent(Z,peter) first, as implied by its better *groundability*.

In the case that more than one $q_i$ have the same groundability, the ordering of these $q_i$'s might still matter. Assume that both $q_1(x)$ and $q_2(x)$ have ground substitutions, $q_1$ returns the ***binding*** in $k_1$ steps, $q_2$ returns in $k_2$ steps and $k_1 >> k_2$ (i.e., $q_1$ has a much larger computation tree than $q_2$). If $q_1$ is selected first, we can start to process the remaining $q_i$'s in parallel after $k_1$ steps. Assuming that the longest time taken by these $q_j$'s is $k_j$, then the total time is $(k_1 + k_j)$. If $q_2$ is selected first, the total time is then $\max(k_2 + k_j,\ k_2 + k_1)$. Suppose $k_j$ is in the same order as $k_1$, then the time ratio between $q_1$-first computation to $q_2$-first computation is about 2 to 1. Therefore the second measure regarding *how soon can a given subgoal return the answer*, called here ***subtree complexity***, will also be taken into account in our ordering algorithm. Here, *subtree* means the subtree of the AND/OR computation tree, and *complexity* means the inference steps required to solve a subgoal. While there is no easy way to obtain the exact computation complexity of a subtree prior to its execution, the complexity may be estimated for program clauses that have special patterns such as the following:

(a). Tail-recursion:

$$p([H|T], ...) \leftarrow q(H, ...), p(T, ..).$$

assume input list has a length n and q is some terminal node, then the complexity is $O(n)$ when the complexity of q is less than that of p.

(b). Head-tail recursion:

$$p([H|T], ...) \leftarrow p(H, ...), p(T, ..).$$

assume input list has a length n, then the complexity is $\Omega(\log n)$ and $O(n)$.

With the help of the groundability and the subtree complexity techniques, we can obtain an efficient partial ordering of subgoals in the case of output dependency. The subgoal ordering of the whole logic program can then be determined via these techniques as well as the result of phase 2. This is described below:

¶ **Ordering Algorithm**:

*Input* : A logic program.
*Output* : Partial ordering of the subgoals in each clause of the program.
*Method* :

1. Perform input-output mode analysis algorithm for the program.
2. Find functional, internal and coupling dependencies. The partial ordering of subgoals in functional dependency case is determined by producer-consumer order.
3. For other types of dependencies, determine the groundability for each atom. Let $sg_i$ and $sn_i$ be the number of ground and non-ground output variables in atom $q_i$, then:

(a). for those $q_i$ whose $sn_i = 0$, assume there are k1 such $q_i$'s, assign priority number 1 through k1 to them in an order from larger $sg_i$ to smaller $sg_i$, then

(b). for those $q_i$ whose $sn_i = 1$, or $sn_i = 2$ with these two terms coupled, assume there are k2 such $q_i$'s, assign priority (k1+1) through (k1+k2) to them, then

(c). for those $q_i$ whose $sn_i = 2$, assume there are k3 such $q_i$'s, k3 is small (say 1), assign priority (k1+k2+1) through (k1+k2+k3) to them for two cases: two variables never coupled -- ordering $\mathbf{O}_a$, and when coupled at run-time -- ordering $\mathbf{O}_b$, then

(d). for the rest of $q_i$'s, assume there are k4 such $q_i$'s, we only assume a no run-time coupling case as default case, assign priority (k1+k2+k3+1) through (k1+k2+k3+k4) to them.

4. If the same priority numbers result in the above step, perform subtree complexity algorithm-- less complexity receives less priority number (or higher priority).
5. Remove all but the highest priority groundable output variables from atom in step 3. This means if a variable can be instantiated by some atom, the variable becomes input after the execution of that atom and will not count toward dependency). Put all removed atoms as well as atoms that have input terms only into a set called "last set" which is the last to be processed.
   The compile time partial ordering is done.
6. If in step 3, k3≠0, say k3=1, assume x and y are two output variables, we attach a run-time test statement as below:

   (if (x couples y) then ordering $\mathbf{O}_b$
   else orering $\mathbf{O}_a$)
7. If in step 3, k4≠0, assume $x_1$, ..., $x_r$ are involved output variables, then we have:

   (if no coupling among [$x_1$, ..., $x_r$]
   then take the default ordering
   else complete the ordering at run-time)

<End of algorithm>

### 3.4 Performance analysis by example

Consider a logic program that has one non-unit clause:

candidate(X) ← degree(X), engrg(X), exp(X,5),
salary(X,50)

and many unit clauses serve as a database which includes 100 items of "degree(X)," 50 items of "engrg(X)," 10 items of "exp(X,5)" and 25 items of "salary(X,50)," as shown below:

| | | | |
|---|---|---|---|
| degree(John) | engrg(Bob) | exp(John,5) | salary(John,50) |
| degree(Bob) | engrg(Mary) | exp(Mary,3) | salary(Mary,35) |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| degree(Peter) | engrg(Peter) | exp(Peter,5) | salary(Peter,50) |
| <..100 items> | <..50 items> | <..10 items> | <..25items> |

Assume the goal clause is ← candidate(X). Now we employ the ordering technique:

1. X is an output variable, all four subgoals have internal dependency with each other.
2. There is no functional dependency.
3. All four subgoals have $sn_i = 0$, $sg_i = 1$.
4. Subgoals exp(X,5), salary(X,50), engrg(X), degree(X) have priority 1, 2, 3 and 4 respectively.
5. Subgoals salary(X,50), engrg(X) and degree(X) are put into the "last set."
6. Set {exp(X,5)} forms the first set, and the rest form the second set.

According to the ordering, the subgoal exp(X,5) will be selected first. Ten items are retrieved from the data base and are fed into the other three subgoals as their input binding sets. These three subgoals can be started as soon as the first answer from exp(X,5) is available. Assume that there is only one name, "Peter," that matches the goal, although "Bob" matches both "degree(X)" and "engrg(X)." For simplicity, assume all unifications take a time step, the above scheme takes 12 parallel unification steps using up to 100 processing units in this particular case. If we use NSV scheme with arbitrary ordering -- starting from "degree(X)" say, it may take 102 parallel unification steps using the same amount of processing units, much worse than the scheme proposed in this paper.

## 4. Conclusions

We had analyzed the AND-parallelism in logic programming by means of input-output and data dependencies, and we had also devised the automatic input-output mode detection technique which is then used to derive high efficiency subgoal ordering. Because most of the detection works can be done at compilation time, the resulting parallel machine need not be very complicated. Each processing unit in the parallel machine needs to do only unification and simple housekeeping as the run-time tasks are already minimized before a program is executed. Based on this parallelism analysis, a simple and efficient VLSI architecture for parallel logic programming was proposed in [19].

## References

[1] Apt, K.R. and van Emden, M.H. : "Contributions to the Theory of Logic Programming," *Journal of the ACM 29(3)*, pp.841-863, July 1982.

[2] Chang, J.-H., Despain, A.M. and DeGroot D.: "And-parallelism of Logic Programs Based on a Static Data Dependency Analysis," *COMPCON 85*, pp.218-225, February 1985.

[3] Clocksin, W.F. and Mellish, C.S. : *Programming in PROLOG*, Springer-Verlag, 1981.

[4] Conery, J.S. and Kibler, D.F. : "Parallel Interpretation of Logic Program," Technical Report 173a, Department of Information and Computer Science, University of California, Irvine, Irvine, California, August 1981.

[5] Conery, J.S. : "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, Irvine, California, June 1983.

[6] DeGroot, D. : "Restricted And-parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, Japan, pp.471-478, November 1984.

[7] DeGroot, D. : "Alternate Graph Expressions for Restricted And-parallelism," *COMPCON 85*, pp.206-210, February 1985.

[8] Hogger, C.J. : "Concurrent Logic Programming," in Clark, K.L. and Tarnlund, S.-A. (eds.), *Logic Programming*, Academic Press, pp.199-211, 1982.

[9] Ito, N. and Masuda, K. : "Parallel Inference Machine Based on the Data Flow Model," *Proceedings of the International Workshop on High-Level Computer Architecture 84*, pp.4.31-4.40 , May 1984.

[10] Kowalski, R. and Kuehner, D. : "Linear Resolution with Selection Function," *Artificial Intelligence 2*, North-Holland Publishing Company, pp.227-260, 1971.

[11] Kowalski, R. : "Predicate Logic as Programming Language," *Proceedings of IFIP Congress*, North-Holland Publishing Company, Amsterdam, pp.569-574, 1974.

[12] Kowalski, R. : "Algorithm = Logic + Control," *Communications of the ACM*, vol.22, no.7, pp.424-436, July 1979.

[13] Lloyd, J.W. : *Foundations of Logic Programming*, Springer-Verlag, Berlin, German, 1984.

[14] Mellish, C.S. : "The Automatic Generation of Mode Declarations for PROLOG Programs," DAI Research Paper no.163, August 1981.

[15] Pollard, G.H. : "The Design of a Parallel Execution Scheme for PROLOG Program," D.P.E., British Telecom, UK, July 1980.

[16] Pollard, G.H. : "Parallel Execution of Horn clause Programs," Ph.D. dissertation, University of London, Imperial College of science & Technology, UK, 1981.

[17] Robinson, J.A. : "A Machine-oriented Logic based on the Resolution principle," *Journal of the ACM*, vol. 12, pp.23-44. 1965.

[18] Robinson, J.A. : "Logic Programming -- Past, Present and Future," *New Generation Computing*, vol.1, no.2, pp.107-124, 1983.

[19] Tung, Y.-W. : "Parallel Processing Model for Logic Programming," Ph.D. dissertation, Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, California 90089-0781, May 1986.

[20] van Emden, M.H. and Kowalski, R.A. : "The Semantics of Predicate Logic as a Programming Language," *Journal of the ACM* 23, pp.733-742, October 1976.

[21] van Emden, M.H. and de Lucena Filho, G.J. : "Predicate Logic as a Programming Language for Parallel Programming," in Clark, K.L. and Tarnlund, S.-A. (eds.), *Logic Programming*, Academic Press, pp.189-198, 1982.

# HOW GOOD ARE PARALLEL AND ORDERED DEPTH-FIRST SEARCHES?

*Guo-Jie Li and Benjamin W. Wah*
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

**ABSTRACT**

Parallel depth-first searches are widely used to solve combinatorial optimization and decision problems in artificial intelligence and operations research. These problems are represented by OR-trees and AND/OR-trees. The performance of parallel depth-first searches may be difficult to predict due to the nondeterminism and anomalies of parallelism. In this paper we have derived the performance bounds of parallel depth-first searches with respect to optimization problems represented as OR-trees and have verified these bounds by simulations. These bounds provide the theoretical foundation to determine the number of processors to assure a near-linear speedup. The conditions to cope with parallel-to-parallel anomalies are also investigated. For decision problems represented by AND/OR-trees, such as evaluating logic programs, we have studied an ordered depth-first search that rearranges nodes in each level of the AND/OR tree to minimize the expected search cost.

## 1. INTRODUCTION

Combinatorial-search problems can be classified into two types. The first type is decision problems that decide whether at least one solution exists and satisfies a given set of constraints [21]. Theorem-proving, expert systems, and evaluating a logic program belong to this class. The second type is combinatorial extremum-search or optimization problems that are characterized by an objective function to be minimized or maximized and a set of constraints to be satisfied. Practical problems such as finding the shortest path, planning, finding the shortest tour of a traveling salesman, job-shop scheduling, packing a knapsack, vertex cover, and integer programming belong to this class.

The non-terminal nodes in a search tree (or graph) can be classified as AND-nodes and OR-nodes. An *AND-node* represents a problem (or subproblem) that is solved only if all its descendant nodes have been solved, while an *OR-node* represents a problem (or subproblem) that is solved only if any of its immediate descendants is solved. Based on these two kinds of nodes, a combinatorial search can be classified into an AND tree, OR-tree, and AND/OR-tree search [25]. Note that a general dataflow graph contains AND-nodes and OR-nodes that relate the descendant nodes, as well as other nodes that relate the ascendant nodes.

In this paper we will concentrate on evaluating problems that arise in nondeterministic computations, namely, those problems that are represented as OR-trees or AND/OR-trees. As an AND-tree represents deterministic computations and all nodes in it must be evaluated, it will not be discussed here [14]. Due to space limitation, we will only present results on the depth-first search strategy. Results on other strategies with respect to OR-trees can be found elsewhere [15, 16].

An OR-tree is a state-space tree in which all non-terminal nodes are OR-nodes, while an AND/OR-tree is a problem-reduction representation that consists of AND-nodes and OR-nodes. Many AND/OR-tree search procedures, such as AO$^*$, SSS$^*$, and dynamic programming, can be formulated as a general branch-and-bound (B&B) procedure [8, 19], which is a well-known OR-tree search method. Likewise, evaluating a logic program can be represented as an OR-tree or AND/OR-tree search [7].

Both combinatorial OR-tree and AND/OR-tree search procedures can be characterized by four constituents: a branching rule, a selection rule, an elimination rule, and a termination condition. The first two rules are used to decompose problems into simpler subproblems and to appropriately order the search. The last two rules are used to to eliminate unnecessary subproblems. Appropriately ordering the search and restricting the region searched are key ideas behind any search algorithms.

The rules to guide the search and to prune unnecessary searches may differ for optimization and decision problems. In optimization problems, a lower bound of the objective value for each nonterminal node can be used to guide the search and to prune nodes that cannot lead to a better solution. Dominance tests, such as $\alpha-\beta$ pruning, can also be adopted as elimination rules. In decision problems, it was found that the ratio of the success probability of a subproblem to the estimated overhead of evaluating the subproblem is useful to guide the search [21, 1, 12]. The elimination rules are more restricted in decision problems, such as evaluating a logic program. Pruning a subproblem with a smaller success probability or a larger search cost may remove a possible (and possibly a unique) solution. In this case only when a terminal node is found to be true or false, AND-pruning or OR-pruning rules can be applied [12].

There are three basic selection strategies, namely, depth-first, breadth-first, and best-first searches. A generalized heuristic function can be used to unify these three kinds of search strategies and resolve ambiguities in the heuristic function [4, 10]. To resolve the ambiguity on the selection of subproblems, distinct heuristic values must be defined for the nodes to allow ties to be broken. A path number can be used to define an unambiguous heuristic function. The *path number* of a node in a tree is a sequence of (h+1) integers representing the path from the root to this node, where h is the maximum number of levels of the tree [10, 15]. For example, the path numbers of nodes A, B, C, and D in Figure 1c are 0000, 0100, 0200, and 0300, respectively. Note that the nodes having equal path numbers never coexist simultaneously in the search process. For a depth-first search, the generalized heuristic function is defined as

$$h(P_i) = (\text{path number, level number}) \qquad (1)$$

Although a best-first search expands fewer nodes than a depth-first search, it requires a secondary memory to maintain the large number of active nodes, hence the total time, including the time spent on data transfers between the main and secondary memories, to solve a problem may be longer than that required by the depth-first search. Simulations have shown that the best OR-tree search strategy depends on the accuracy of the problem-dependent lower-bound function [24]. Very inaccurate lower bounds are not useful to guide the search, while very

This research was supported by National Science Foundation Grants DMC 85-19649.

International Conference on Parallel Processing, 1986.

accurate lower bounds will prune most unnecessary expansions. In both cases the number of subproblems expanded by depth-first and best-first searches will not differ greatly, and a depth-first search is better as it requires less memory space.

Extensive studies have been conducted on OR-parallelism [2, 12, 15, 18], but very few studies have been done on analyzing the speedups and efficiency of OR-parallelism. Due to the nondeterminism, combinatorial OR-tree and AND/OR-tree searches are quite different from conventional deterministic numerical computations. Simulation results have revealed that using more processors in parallel depth-first searches might degrade the performance, even when the communication overhead is ignored [10]. The prediction of performance and methods to cope with anomalous behavior are important problems to be studied in designing multiprocessors for parallel depth-first searches and will be addressed in this paper.

To take advantage of the search efficiency of best-first searches while avoiding their memory overhead, an informed depth-first search can be used [20]. In this strategy best-first search is performed locally and depth-first search globally. A special case is one in which all sibling nodes are ordered according to heuristic values of the siblings (a more accurate definition will be given in Section 3). We will show that this *ordered depth-first strategy* is very effective to evaluate logic programs represented as AND/OR trees.

## 2. PARALLEL DEPTH-FIRST OR-TREE SEARCHES

To predict the number of processors needed to assure a near-linear speedup in a parallel depth-first search, we will derive the bounds on computational efficiency. The results in this section indicate the relationship among the number of iterations required in a parallel depth-first search, the number of processors used, and the complexity of the problem to be solved.

### 2.1. Model of Efficiency Analysis

In analyzing the performance bounds, a synchronous model is assumed, that is, all processors must finish the current iteration before proceeding to the next iteration. This performance results form a lower bound to that of asynchronous models.

The parallel computational model used here consists of a set of processors connected to a shared memory. In each iteration, multiple subproblems are selected and decomposed. The newly generated subproblems are tested for feasibility, eliminated by (exact or approximate) lower-bound tests and dominance tests,* and inserted into the active list(s) if not eliminated. In this model eliminations are performed after branching instead of after selection as in Ibaraki's algorithm [5] to reduce the memory space required.

We have proved that, for best-first searches, the performance is not largely affected by whether the active subproblems are kept in a single shared list or multiple lists [23,15]. However, for depth-first searches, the performance will be problem-dependent when multiple lists are used. In this paper the performance bounds are derived under the assumption that one list is used and that the nodes with the smallest heuristic values are selected in each iteration.

Since subproblems are decomposed synchronously and the bulk of the overhead is on branching operations, the number of iterations, which is the number of times that subproblems are decomposed in each processor, is an adequate measure in both the serial and parallel models. The *speedup* between using $k_1$ and $k_2$, $k_2 > k_1$, processors is thus measured by the ratio of the number of iterations when $k_1$ processors are used to that when $k_2$ processors are used. Once the optimal solution is found, the time to drain the remaining subproblems from the list(s) is not accounted for, since this overhead is negligible as compared to that of branching operations.

The results proved in this section show the performance bounds of parallel depth-first OR-tree searches for solving optimization problems. The proofs of these theorems require the following definitions on essential nodes. A node expanded in a *serial* depth-first search is called an *essential node*, otherwise it is called a *non-essential node*. The speedup of a parallel depth-first search depends on the number of essential nodes selected in each iteration. An iteration is said to be *perfect* if the number of essential nodes selected is equal to the number of processors, otherwise it is said to be *imperfect*. The *incumbent* at any given time in the search process is the best feasible solution obtained at that time. The incumbent is continuously updated until an optimal solution is found. We denote $T_b(k,\epsilon)$ and $T_d(k,\epsilon)$ as the number of iterations required to find a single optimal (or suboptimal) solution using k, $k \geqslant 1$, processors in a best-first and depth-first search, respectively, where $\epsilon$ is an *allowance function* specifying the allowable deviation of a suboptimal value from the exact optimal value. When an approximate solution is sought, i.e. $\epsilon > 0$, during the search of an OR-tree, an active node $P_i$ is terminated if

$$g(P_i) \geqslant \frac{z}{1+\epsilon} \qquad \epsilon \geqslant 0,\ z \geqslant 0 \qquad (2)$$

where z is an incumbent obtained at that time.

### 2.2. Parallel Depth-First Searches

The following theorem shows that the performance of parallel depth-first searches depends on the problem complexity and the number of distinct incumbents found during the search process.

**Theorem 1**: For a parallel depth-first OR-tree search with k processors, $\epsilon$=0, and a generalized heuristic function of $h(P_i)$ =(path number, level number), then

$$\left( \frac{T_b'(1,0)-1}{k} + 1 \right) \leqslant T_d(k,0) \leqslant \left( \frac{T_d(1,0)}{k} + \frac{k-1}{k}[(c+1) \cdot h - c] \right) \qquad (3)$$

where h is the height of the OR-tree, c is the number of the distinct incumbents obtained during the serial depth-first search, and $T_b'(1,0)$ is the number of essential nodes in a serial best-first search with lower bounds *less than* the optimal-solution value.

*Proof*: The sequence of iterations obtained during a serial depth-first search can be divided into (c+1) subsequences according to the c distinct monotonically decreasing incumbents obtained. Let the c feasible solutions and their corresponding parents be denoted by $F_1, \ldots, F_c$, and $P_1, \ldots, P_c$. Further, assume that $F_1, \ldots, F_c$ are obtained in the $i_1$'th, ..., $i_c$'th iterations, respectively. Hence iterations from 1 to $i_1$ belong to the first subsequence, and iterations from $i_j+1$ to $i_{j+1}$ belong to the (j+1)'th subsequence.

We now consider the j'th $1 \leqslant j \leqslant c$, subsequence. Let $\ell_{min}(x)$ be the level with the minimum level number in which some active essential nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, reside in the x'th iteration. For levels less than $\ell_{min}(x)$, all active nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, are non-essential. We show that Iteration x is imperfect only if all essential nodes, whose heuristic values are between $h(P_{j-1})$ and $h(P_j)$, in $\ell_{min}(x)$ are selected for expansion. Suppose that Iteration x is imperfect, the selected non-essential node must have heuristic value larger than $h(P_j)$, because otherwise this node would have to be eliminated by the feasible solution $F_{j-1}$ ($F_0$ is the initial feasible solution obtained by a heuristic method). Thus after Iteration x is carried out, $\ell_{min}(x)$ must be increased by at least one. Consequently, after at most h imperfect iterations, $F_j$, must be found.

During the last subsequence of iterations, since the optimal solution has been generated, all iterations are imperfect only if less than k nodes are selected in each iteration. In other words, an imperfect iteration implies that all currently active nodes are selected and expanded, and only descendants of these nodes can be active in the next iteration. Hence no active node remains after at most h imperfect iterations in the last subsequence. The previous analysis shows that at most $(c+1) \cdot h$ imperfect

* Dominance tests will not be discussed in this paper due to space limitation.

iterations can appear in a parallel depth-first search. Since at least one node in each iteration in the parallel case belongs to $\Phi^1$, the set of nodes expanded in the serial depth-first search [10, 15], the upper bound of $T_d(k,0)$ can be derived as

$$T_d(k,0) \leqslant \left\lceil \frac{T_d(1,0)-(c+1)\cdot h}{k} + (c+1)\cdot h \right\rceil$$

In the above discussion, the expansion of the root is counted in each of the (c+1) subsequences. Since the root is only expanded once, the above upper bound should be compensated by the additional number of times that the root is expanded (Eq. (3)).

The lower bound on $T_d(k,0)$ can be proved easily because all essential nodes in a serial best-first search with lower bounds less than the optimal solution must be expanded in the parallel depth-first search. □

For problems such as integer programming and 0-1 knapsack problems, all feasible solutions are located in the bottommost level of the OR-tree. In this case the following corollary shows that all essential nodes of a serial depth-first search must be expanded in a parallel depth-first search, and a tighter lower bound is obtained.

**Corollary 1**: In searching an OR-tree using a parallel depth-first search and a heuristic function of (path number, level number), if $\epsilon = 0$ and all feasible solutions are in Level h, then

$$\left\lceil \frac{T_d(1,\epsilon) - 1}{k} + 1 \right\rceil \leqslant T_d(k,\epsilon) \tag{4}$$

where h is the maximum number of levels of the OR-tree.
*Proof*: The proof is omitted due to the space limitation [13, 10].

The bounds in Theorem 1 are tight in the sense that we can construct examples to achieve the lower- and upper-bound of computational times. These degenerate cases occur rarely. Although c, the number of distinct incumbents, is unknown until the solution is found, c is usually small and can be estimated when integral solutions are sought. It has been observed that c is less than 10 for vertex-cover problems with less than 100 vertices. For most integer programming problems, $c \approx 1$. In these cases the range on $T_d(k,0)$ is tight, and a near-linear speedup can be achieved in a large range of k.

Let w be $T_d(1,0)/h$. w can be viewed as the "average width" of an OR-tree, which only consists of essential nodes. Eq. (3) can be rewritten as

$$\frac{T_d(1,0)}{T_d(k,0)} \geqslant \frac{k\cdot w}{w+(c+1)(k-1)} \tag{5}$$

From Eq. (5), it is easy to see that if w >> k and c is small, then the speedup is close to k; whereas if w << k, then the lower-bound speedup is close to w/(c+1).

In Table 1, the theoretical bounds derived above are compared with the simulation results of parallel depth-first searches to solve two 35-object knapsack problems. In generating the knapsack problems, w(i), the weights, were chosen randomly between 0 and 100 with a unform distribution, and the profits were set to be p(i) = (w(i) + 10). This assignment is intended to increase the complexity of the randomly generated problems. The results demonstrate that the bounds on parallel depth-first searches are tight, hence its performance can be predicted quite accurately. Table 1 also shows that the speedup depends strongly on w. In Case 1 $w \approx 2023$, and a near-linear speedup of 0.88k is achieved with 256 processors. In Case 2 $w \approx 188$, and a speedup of 0.29k is obtained with 256 processors. Note that when the number of processors is large, the number of essential nodes in each imperfect iteration of the parallel depth-first search is usually larger than one. In contrast to the upper bound in Eq. (3), which was derived with the assumption of one essential node in each imperfect iteration, $T_d(k,0)$ may be much smaller than the upper bound. Simulations have also revealed that for a number of OR-tree search problems, $T_d(k,0)$ may be very close to $T_b(k,0)$.

| No. of Proc. | Lower bound | No. of iterat. | Upper bound | Speedup |
|---|---|---|---|---|
| Case 1 | | | | |
| 1 | 70790 | 70790 | 70790 | 1.000 |
| 2 | 35395 | 35630 | 35787 | 1.987 |
| 4 | 17698 | 18044 | 18285 | 3.923 |
| 8 | 8849 | 8884 | 9534 | 7.968 |
| 16 | 4425 | 4460 | 5159 | 15.872 |
| 32 | 2213 | 2247 | 2971 | 31.504 |
| 64 | 1107 | 1143 | 1877 | 61.934 |
| 128 | 554 | 592 | 1330 | 119.578 |
| 256 | 277 | 316 | 1057 | 224.019 |
| Case 2 | | | | |
| 1 | 6566 | 6582 | 6582 | 1.000 |
| 2 | 3283 | 3488 | 3513 | 1.887 |
| 4 | 1642 | 1940 | 1978 | 3.393 |
| 8 | 821 | 1161 | 1211 | 5.669 |
| 16 | 411 | 777 | 827 | 8.471 |
| 32 | 206 | 584 | 635 | 11.271 |
| 64 | 103 | 485 | 539 | 13.571 |
| 128 | 52 | 219 | 491 | 30.055 |
| 256 | 26 | 90 | 467 | 73.133 |

Table 1. Comparisons between theoretical bounds and simulation results on parallel depth-first searches for knapsack problems with 35 objects. ($T_b'(1,0)=T_b(1,0)$. During depth-first searches, c=22 in Case 1 and c=12 in Case 2.)

Analogous to the proof of Theorem 1, the upper bound on $T_d(k,\epsilon)$, $\epsilon > 0$, can be derived. To find the lower bound on $T_d(k,\epsilon)$, let $f_o$ be the optimal-solution value and $MINT_b(\epsilon)$ be the minimum number of nodes to be expanded in the approximate best-first search. $MINT_b(\epsilon)$ represents the number of nodes whose lower bounds are less than $f_o/(1+\epsilon)$, since these nodes must be expanded in the best case. $MINT_b(\epsilon)$ may be estimated from the distribution on the number of subproblems with respect to lower bounds. From the above analysis, we get

$$\left\lceil \frac{MINT_b(\epsilon)-1}{k}+1 \right\rceil \leqslant T_d(k,\epsilon) \leqslant \left\lceil \frac{T_d(1,\epsilon)}{k}+\frac{k-1}{k}[(c+1)\cdot h-c] \right\rceil \tag{6}$$

### 2.3. Coping With General Parallel-to-Parallel Anomalies

Some results on coping with serial-to-parallel anomalies have been published elsewhere [10, 11, 15]. We now present results on coping with parallel-to-parallel anomalies of depth-first OR-tree searches based on the performance bounds derived in the last section. When comparing the efficiency between using $k_1$ and $k_2$ processors, $1 \leqslant k_1 < k_2$, a $k_2/k_1$-fold speedup (ratio of the number of iterations in the two cases in our model) is expected. However, simulations have shown that the speedup can be (a) less than one (called a *detrimental anomaly*) [6, 17, 9]; or (b) greater than $k_2/k_1$ (called an *acceleration anomaly*) [6, 9]; or (c) between one and $k_2/k_1$ (called a *deceleration anomaly*) [6, 22, 17, 9]. So far, all known results on parallel OR-tree searches showed a near-linear speedup for only a small number of processors.

Anomalies are studied with respect to the assumption that all idle processors are used to expand active subproblems. In fact, detrimental anomalies cannot happen if some processors can be kept idle in the presence of active subproblems. The number of processors to be kept idle is problem dependent and is very difficult to find without first solving the problem.

Some anomalies on parallel depth-first OR-tree searches are illustrated here. A single list of subproblems is assumed. The behavior of using multiple lists is analogous to that of a centralized list. An example of an acceleration anomaly with an approximate depth-first or best-first search is shown in Figure 1a. When three processors are used, the optimal solution is found in the second iteration, and $P_4$ and $P_5$ are eliminated. If two processors are used, subtrees $T_4$ and $T_5$ have to be expanded. $T(2,0.1)/T(3,0.1)$ will be much larger than 3/2 if $T_4$ and $T_5$ are very large. Figure 1b illustrates a detrimental anomaly under an approximate best-first or depth-first search with $\epsilon=0.1$. When two processors are used, $f(P_8)$, the optimal solution, is found in the fourth iteration. Assuming that the lower bounds of nodes in $T_3$ are between 8.2 and 9, all nodes in $T_3$ will be eliminated by lower-bound tests with $P_8$ since $[9/(1+\epsilon)]<8.2$. When three processors are used, $P_3$ is expanded in the third iteration. $P_5$, $P_6$, and $P_7$ are generated and will be selected in the next iteration. If $T_3$ is large, $T(2,\epsilon) < T(3,\epsilon)$ will occur. A detrimental anomaly may occur even when lower-bound tests are inactive and is illustrated in Figure 1c. A similar example can be derived for acceleration anomalies.

In the last section, we have derived the performance bounds with respect to depth-first OR-tree searches. From these results, we can develop the relative efficiency between using $k_1$ and $k_2$, $1<k_1<k_2$, processors. First, we derive a sufficient condition to assure the monotonic increase in computational efficiency with respect to the number of processors. To simplify the sufficient condition, the following bounds on $T_d(k,0)$ are used.

$$\frac{T_b'(1,0)}{k} \leqslant T_d(k,0) \leqslant \left\lceil \frac{T_d(1,0)}{k} + (c+1)\cdot h \right\rceil .$$

**Corollary 2:**** Let $r_d' = T_b'(1,0)/T_d(1,0) \leqslant 1$. In a parallel depth-first search that satisfies the assumptions of Theorem 1, $T_d(k_2,0) \leqslant T_d(k_1,0)$ when

$$\frac{T_d(1,0)}{h} \geqslant \frac{(c+1)k_1k_2}{r_d'k_2 - k_1} \quad \text{and} \quad r_d' > \frac{k_1}{k_2}, \quad 1<k_1<k_2 \qquad (7)$$

where c is the number of the distinct incumbents obtained during the serial depth-first search.

From Corollary 2, we can conclude that the existence of parallel-to-parallel detrimental anomalies in depth-first searches depends on $T_b'(1,0)$, $r_d'$, and c. If $r_d' \approx 1$, c is small, and $T_b'(1,0)$ is very large, then Eq. (7) will be satisfied. Our simulation results reveal that for some problems, such as the 0-1 knapsack and vertex-cover problems, $T_d(1,0)$ is close to $T_b'(1,0)$, hence $r_d' \approx 1$. Moreover, if the feasible-solution values must be integers, then c is often small. For this kind of problems, detrimental anomalies can be prevented for parallel depth-first searches when $T_b'(1,0)$ is large and $k_2$ is relatively small. However, the range of parallel processing within which no detrimental anomalies occur for depth-first searches is smaller than that for best-first searches [13].

From Theorem 1, we can also derive a necessary condition for acceleration anomalies with respect to $k_1$ and $k_2$ processors.

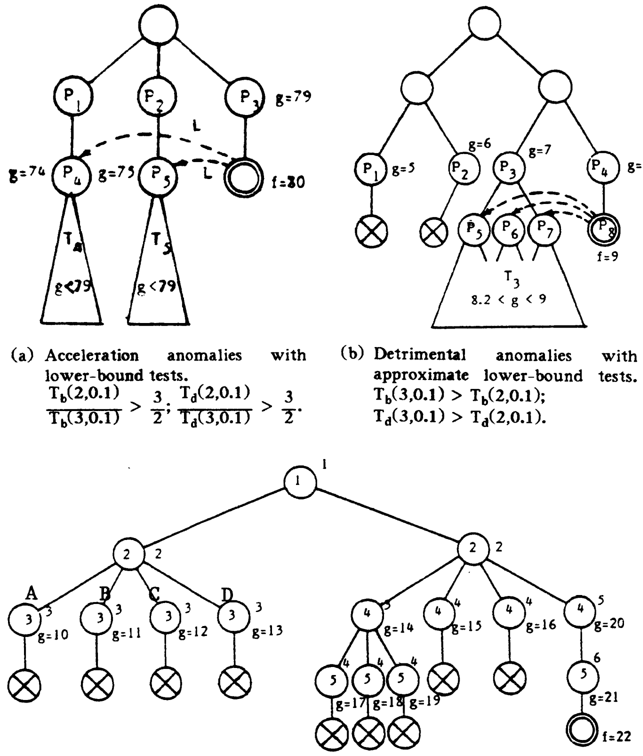
**Corollary 3:**** In a parallel depth-first search that satisfies the assumptions of Theorem 1, $T_d(k_1,0)/T_d(k_2,0) > k_2/k_1$ only if

$$\left\lceil T_d(1,0)-T_b'(1,0) \right\rceil > \left\lceil k_2-1-(k_1-1)[(c+1)\cdot h-c] \right\rceil \quad 1<k_1<k_2 \quad (8)$$

If all solutions are located at the bottommost level of the OR-tree, then the corresponding necessary condition is simplified by Corollary 1 as

$$\left\lceil (c+1)\cdot h - c \right\rceil > \frac{k_2-1}{k_1-1} \qquad 1<k_1<k_2. \qquad (9)$$

Obviously, the necessary condition in Eq. (8) is readily satisfied, and $T_d(k_1,0)/T_d(k_2,0)$ may be much greater than $k_2/k_1$.



(a) Acceleration anomalies with lower-bound tests. $\frac{T_b(2,0.1)}{T_b(3,0.1)} > \frac{3}{2}$; $\frac{T_d(2,0.1)}{T_d(3,0.1)} > \frac{3}{2}$.

(b) Detrimental anomalies with approximate lower-bound tests. $T_b(3,0.1) > T_b(2,0.1)$; $T_d(3,0.1) > T_d(2,0.1)$.

(c) Detrimental anomalies without lower-bound tests in a depth-first or best-first search, $T(4,0)=5$, $T(5,0)=6$.
(Number inside node is the evaluation order using four processors; number outside node is the evaluation order using five processors.)

Figure 1. Examples of anomalies.

Usually, if $k_1$ and $k_2$ are close to each other and h is large, then acceleration anomalies may occur quite often.

When a suboptimal solution is sought, the following corollary shows the required sufficient conditions.

**Corollary 4:**** In parallel depth-first searches that satisfy the assumptions of Theorem 1 with the exception that $\epsilon>0$, $T_d(k_2,\epsilon) \leqslant T_d(k_1,\epsilon)$ when

$$\frac{T_d(1,\epsilon)}{h} \geqslant \frac{(c+1)k_1k_2}{r_dk_2 - k_1} \quad \text{and} \quad r_d > \frac{k_1}{k_2}, \qquad 1<k_1<k_2 \quad (10)$$

where $r_d = \mathrm{MIN}T_b(\epsilon)/T_d(1,\epsilon)$. $T_d(k_1,\epsilon)/T_d(k_2,\epsilon) > k_2/k_1$ when

$$\left\lceil T_d(1,\epsilon)-\mathrm{MIN}T_b(\epsilon) \right\rceil > \left\lceil k_2-1-(c+1)(k_1-1)\cdot h \right\rceil \qquad (11)$$

If all feasible solutions are located at the bottommost level of the OR-tree, the necessary condition to allow acceleration anomalies is the same as that stated in Eq. (9). Further, a weaker sufficient condition to eliminate detrimental anomalies can be derived from Corollary 1.

$$\frac{T_d(1,\epsilon)}{h} > \frac{(c+1)k_1k_2}{k_2 - k_1} \qquad (12)$$

## 3. ORDERED DEPTH-FIRST SEARCH FOR EVALUATING LOGIC PROGRAMS

In our previous paper [12] we have developed an optimal search strategy to evaluate logic programs modeled as AND/OR-trees using the heuristic information p(x), the success probability of a subgoal (or clause) x, and c(x), the estimated overhead of evaluating the subgoal (or clause). The heuristic information to guide the search is defined as follows.

** The proof is omitted due to space limitation and can be found elsewhere [13, 16].

$$\Phi_a(x) = \frac{p(x)}{c(x)} \qquad \text{(x is descendant of an OR—node)} \tag{13}$$

$$\Phi_o(x) = \frac{1 - p(x)}{c(x)} \qquad \text{(x is descendant of an AND—node)} \tag{14}$$

The logic program is first transformed from the AND/OR-tree representation into a two-level AND/OR-tree. The root of the transformed tree is an OR-node and represents the selection of clauses, and its descendants are AND-nodes and represent different solution trees in the logic program. The descendents of the OR-node are ordered according to decreasing values of $\Phi_o$, and the descendants of the AND-nodes are ordered according to decreasing values of $\Phi_a$.

Although the above strategy minimizes the expected search time, there are two implementation problems. First, the transformed AND/OR-tree significantly expands the number of nodes in the original AND/OR-tree. In fact, the number of potential solution trees is a hyper-exponential function of the height of the tree. To apply the above search strategy on the original AND/OR-tree, a global list is required to maintain the order of all possible solution trees, and the storage overhead is prohibitively large [12]. Second, if two solution trees $T_1$ and $T_2$ have nearly equal $\Phi_a$ or $\Phi_o$, then exchanging the search order of $T_1$ and $T_2$ may not significantly improve the expected search overhead. As an example, suppose that the success probabilities and the estimated overheads of all solution trees rooted at a nonterminal node are uniformly distributed between 0.01 and 0.99 and 1 and 10 units of cost, respectively, and that there are a million possible solution trees from this node. Suppose further that two solution trees can be viewed as having nearly equal $\Phi_a$ or $\Phi_o$ if their difference is less than 0.001. Then, approximately, every thousand solution trees have nearly equal $\Phi_a$ or $\Phi_o$. Obviously, it is unnecessary to store the exact order of all solution trees.

In this section we will address two problems. First, given an ordered depth-first search strategy and assuming that all sibling nodes in the AND/OR-tree are independent, what is the order to search the nodes in each level of the AND/OR-tree to minimize the expected search time? Second, for a logic program with shared variables and clauses, how should the subgoals and clauses be ordered to minimize the average search cost of a depth-first search?

### 3.1 Assumptions

In a logic program, if there are n clauses whose heads match (sub-)goal A, then they can be ordered according to the given heuristic values. Likewise, if there are m subgoals in the body of a clause B, $B :- B_1, ..., B_m$, then the m subgoals can also be ordered.

The assumptions made in the search strategy are described here.

(1) For a given representation of the AND/OR-tree, a depth-first search is used. When nodes in each level are ordered according to the heuristic values, the search is called an *ordered depth-first search.*

(2) A *producer-consumer* model is used to bind values to variables. A variable is a producer if it has not been bound to any value, otherwise, it is a consumer. For each variable not defined in the head, only its leftmost occurence can be the producer, as a depth-first search is used. All other occurences of this variable in this clause are consumers. For example, in the clause A(x,y):-B(x,z)C(z,y)D(x,y), variable z in subgoal B must be a producer, while variable z in subgoal C is a consumer. Depending on whether a variable defined in the head is a producer or a consumer, the variable in the corresponding subgoal will be a producer or a consumer. For example, if x is a producer in A, then x in B is a producer, while x in D is a consumer. We use a subscript '+' to indicate that the mode of a variable is a producer and a '—' to indicate that its mode is a consumer. As an example, $A(x_+,y_-) :- B(x_+,z_+)C(z_-,y_-)D(x_-,y_-)$. When a variable in a subgoal is a consumer, it is necessary to verify in this subgoal whether the subgoal is TRUE or FALSE for such a binding of value. In contrast, when a variable in a subgoal is a producer, it is necessary to find a binding of value to the variable such that this subgoal is TRUE.

(3) The probability of a subgoal to return TRUE and the average minimum overhead to determine whether a subgoal is TRUE or FALSE are independent of the bound values.

(4) The overhead to test whether a subgoal in a clause is TRUE or FALSE for a given binding of values to variables or to generate a binding of values to variables is assumed to be independent of other subgoals in this clause, provided that the modes of its variables are unchanged. Likewise, the overhead to verify the head of a clause is independent of other clauses with the same head when the modes of its variables are unchanged. These assumptions are valid when results in one subgoal or clause are passed to other subgoals or clauses through the binding of values to variables.

(5) The probability that a subgoal in a clause is TRUE for a given binding of values to variables is assumed to be independent of other subgoals in this clause. Similarly, the probability that the head of a clause is TRUE is independent of other clauses with the same head. These assumptions are not valid in general logic programs because subgoals have shared clauses and variables, but are made here to simplify the model.

### 3.2. Optimal Ordering of Depth-First Searches in AND/OR-Trees

In this section we discuss a special case in the optimal ordering of depth-first searches for AND/OR-trees, assuming that the success probabilities and expected overheads of all nodes are independent of each other, and that a node, once evaluated, will not be evaluated again. This special case exists in a logic program when it does not have any logic variables and shared clauses. For each node in the AND/OR-tree, suppose that it has n descendent nodes, then there are n! possible evaluation orders for a depth-first search. Our objective is to select the optimal order of descendents for each node in the AND/OR-tree such that the average overhead to verify the root to be TRUE or FALSE is minimized.

Various heuristic functions can be used to arrange the order of descendent nodes. Examples include the success probability, the lower bound on cost, and the number of immediate descendents. The following theorem shows that $\Phi_a$ and $\Phi_o$ (for AND-nodes and OR-nodes, respectively) are the heuristic functions to order the search such that the expected search cost is minimized.

**Lemma 1**: Suppose that node K is an OR-node (resp. AND-node) with n (resp. m) immediate descendent AND-nodes (resp. OR-node) ordered as $K_1, ..., K_n$ (resp. $K_1, ..., K_m$), and $K_i$ is searched before $K_{i+1}$ in a depth-first search. Let $p_i$ and $c_i$ be the success probability and search cost of node $K_i$, and $q_i=(1-p_i)$. If all $p_i$s and $c_i$s are independent of each other and $p_i/c_i < p_{i+1}/c_{i+1}$ (resp. $q_i/c_i < q_{i+1}/c_{i+1}$), $1 \leqslant i \leqslant n$ , then the expected search cost can be reduced when $K_{i+1}$ is searched before $K_i$.

*Proof*: Let C and C' be the expected costs of searching the descendents of node K in the order $K_1, ..., K_n$ and that in the order with $K_i$ and $K_{i+1}$ interchanged. Assume that node K is an AND node. Then

$$C = \sum_{k=1}^{n} \left( \prod_{j=1}^{k-1} q_j \right) \cdot c_k \qquad \text{and} \tag{15}$$

$$C' = \sum_{k=1}^{i-1} \left( \prod_{j=1}^{k-1} q_j \right) \cdot c_k + \left( \prod_{j=1}^{i-1} q_j \right) \cdot (c_{i+1} + q_{i+1}c_i) + \sum_{k=i+1}^{n} \left( \prod_{j=1}^{k-1} q_j \right) \cdot c_k \tag{16}$$

Subtracting Eq. (15) from (16) yields

$$C' - C = \left( \prod_{k=1}^{i-1} q_k \right) \cdot (p_i c_{i+1} - p_{i+1} c_i) > 0$$

The proof when node K is an OR-node is analogous. □

Some special cases of this ordering strategy have been observed by Simon and others [21, 3, 1].

**Theorem 2**: Assume that a depth-first search is used to search an AND/OR-tree, that the probabilities of success and search costs of all sibling nodes are independent of each other, and that a node, once evaluated, will not be evaluated again. The ordered sequence in which all OR-nodes, $x_i$s, are ordered by decreasing $p(x_i)/c(x_i)$ and all AND-nodes, $y_i$s, are ordered by decreasing $q(y_i)/c(y_i)$ will minimize the expected search cost over all possible ordered sequences, where $p(x)$, $q(x)$, and $c(x)$ are the success and failure probabilities and average search cost for node x.

*Proof*: Without loss of the generality, assume that the root (in Level 0) is an OR-node and that each OR-node (resp. AND-node) has n (resp. m) immediate descendent AND-nodes (resp. OR-nodes). For the n AND-nodes (resp. m OR-nodes), there are n! (resp. m!) possible oredered sequences, $s_1, \ldots, s_{n!}$ (resp. $s_1, \ldots, s_{m!}$). Let $c^i_{j,AND}$ (resp. $c^i_{j,OR}$) be the minimum expected cost of the j'th AND-node (resp. OR-node) in sequence $s_i$ over all possible ordered sequences of descendents of this node. Let $c_{r,OR}$ be the minimum expected cost of a depth-first search of the root over all possible ordered depth-first searches of the given AND/OR-tree. Since the expected search cost of a node is the cost of searching the subtree rooted at this node to return TRUE or FALSE, it is independent of the search order of other sibling nodes. Hence, if all nodes in the k'th level have been ordered optimally, then this optimal order remains unchanged when determining the optimal order in levels smaller than k. That is, the principle of optimality is satisfied. The minimum expected cost of the root r can be found from a dynamic programming formulation.

$$c_{r,OR} = \min_{s_i \in \{s_1,\ldots,s_{n!}\}} \left[ \sum_{j=1}^{n} \left( \prod_{k=1}^{j-1} q^i_k \right) \cdot c^i_{j,AND} \right] \quad \text{where} \tag{17}$$

$$c^i_{j,AND} = \min_{s_u \in \{s_1,\ldots,s_{m!}\}} \left[ \sum_{v=1}^{m} \left( \prod_{w=1}^{v-1} p^u_w \right) c^u_{v,OR} \right] \tag{18}$$

where $p^i_k$ and $q^i_k$ are, respectively, the success and failure probabilities of the k'th node in the i'th ordered sequence $s_i$. $c^u_{v,OR}$ can be evaluated in a similar fashion as in Eq. (17). Eq's (17) and (18) can be solved by a bottom-up evaluation.

For any nonterminal OR-node (resp. AND-node), K, since all its immediate descendents $K_1, \ldots, K_n$ (resp. $K_1, \ldots, K_m$) are independent of each other, then from Lemma 1 and applying adjacent pairwise interchanges, the optimal search order should satisfy $p(K_i)/c(K_i) > p(K_{i+1})/c(K_{i+1})$ (resp. $q(K_i)/c(K_i) > q(K_{i+1})/c(K_{i+1})$). □

The above ordering strategy only holds when all nodes are independent. In general, a logic program has shared variables and shared clauses. Hence, the subgoals and clauses have dependent search costs and success probabilities. Moreover, a subgoal may be searched more than once because a given binding of values to variables may succeed with this subgoal but fail with other subgoals. In the next section, we will discuss a heuristic method to find an efficient search order.

### 3.3. Ordered Depth-First Search of Logic Programs

To find an appropriate order of depth-first search in a logic program, the main problem is to develop a function to compute the expected search cost and success probability of a clause or a subgoal, assuming that the costs and success probabilities of all its immediate descendents in the AND/OR-tree representation are known. The difficulty lies in the shared variables and clauses in different subgoals of a logic program. The search cost of a subgoal may depend on the modes of its variables and cannot be evaluated as in Eq. (15). For a subgoal with a producer variable, it is necessary to generate one (or all) binding of value for the given variable; whereas a subgoal with a consumer variable has to test whether the given binding is TRUE. The latter cost is usually larger than the former one. The cost functions are more complicated when there are multiple variables. Here, a subgoal can have a combination of producer and consumer variables.

Owing to the distinction between producers and consumers and that a clause may be used with their variables set in different modes, the success probabilities and costs must be defined for all combinations of modes of variables. For example, there are four success probabilities and four expected search costs for clause with head A(x,y), namely, $p_A(x_+,y_+)$, $p_A(x_+,y_-)$, $p_A(x_-,y_+)$, $p_A(x_-,y_-)$, $c_A(x_+,y_+)$, $c_A(x_+,y_-)$, $c_A(x_-,y_+)$, and $c_A(x_-,y_-)$, where a subscript '+' indicates that a variable is a producer, and '−' indicates that it is a consumer. Let **L** be the set of variables in a subgoal, and $\mathbf{L}_+$ and $\mathbf{L}_-$ be the subsets of producer and consumer variables. For a clause with head $A(\mathbf{L}_+,\mathbf{L}_-)$, all variables in $\mathbf{L}_-$ have been bound (called a ***binding-set***) before this clause is searched, whereas all variables in $\mathbf{L}_+$ must be bound after the subtree rooted at clause A has been searched.

In Figure 2 we have shown a Prolog program to query granddaughter(•,•). In Table 2 the average search costs for various modes of variables X and Y in granddaughter are shown. For different modes, the orders in which the depth-first search should be performed may be different. We have shown the order that minimizes the search cost for two of these combinations of modes. The structures for the other two combinations are different. The values in Table 2 illustrate that the difference in costs between the best and the worst orders can be a factor of one to seven.

For node A(**L**), $p_A(\mathbf{L}_+,\mathbf{L}_-)$ is defined as the probability to successfully generate a binding-set of $\mathbf{L}_+$ under the condition that the given binding-set of $\mathbf{L}_-$ is TRUE, namely,

$$p_A(\mathbf{L}_+,\mathbf{L}_-) = p_A(\mathbf{L}_-)\, p_A(\mathbf{L}_+ \mid \mathbf{L}_-) \tag{19}$$

```
mother(theresa,martha).
mother(jane,martha).
mother(michael,mary).
mother(susan,jane).
mother(edward,jane).
wife(john,martha).
wife(paul,mary).
wife(michael,jane).
female(theresa).
female(susan).
female(X):-wife(_,X).
father(X,Y):-mother(X,Z),wife(Y,Z).
parent(X,Y):-mother(X,Y).
parent(X,Y):-father(X,Y).
grandparent(X,Y):-parent(Z,Y),parent(X,Z).
granddaughter(X,Y):-female(Y),grandparent(Y,X).
```

Figure 2. Minimum-cost Prolog program on family tree with granddaughter(−,+) or granddaughter(−,−) as the goal.

| *Modes of X,Y in granddaughter* | *Minimum Avg. Cost* | *Maximum Avg. Cost* | *Mean Avg. Cost* | *Standard Deviation* |
|---|---|---|---|---|
| −,− | 20.6 | 97.8 | 46.9 | 28.9 |
| −,+ | 14.1 | 97.9 | 47.4 | 31.6 |
| +,− | 20.1 | 130.8 | 81.6 | 36.0 |
| +,+ | 11.5 | 20.6 | 16.6 | 4.4 |

Table 2. Average Costs of evaluating granddaughter(X,Y) in Figure 2 for all combinations of bindings of variables and all possible solutions returned. (Each traversal of a subgoal or clause has unit cost. Each producer variable only produces one binding at a time.)

$p_A(L_-)$ is the probability that the binding-set of $L_-$ on A is true. $p_A(L_+ | L_-)$ is defined as n/(n+1), where n is expected number of binding-sets of $L_+$ in subgoal A for a given binding-set of $L_-$. In this case we are approximating the distribution on the number of distinct binding-sets of $L_+$ for a given binding-set of $L_-$ as a geometric distribution with parameter p. For such a geometric distribution, its expected value is p/(1−p), which implies that p=n/(n+1). In the special case when all variables in L are producers, then $p_A(L_+)$ = m/(m+1), where m is the total number of generated binding-sets.

For $A(L_+,L_-)$, its expected cost, $c_A(L_+,L_-)$, is defined as the expected cost of generating a successful binding of variables in $L_+$, given the binding of variables in $L_-$. If all variables in L are consumers, then $c_A(L_-)$ is the expected cost of testing whether a binding-set is TURE.

For clarity, we illustrate a heuristic method to compute the various costs. In this method all probabilities are assumed to be independent. For a clause A(x,y) :− B(x,z),C(z,y) with known costs and probabilities for subgoals B and C, the expected cost of A can be computed by modeling the test process as an absorbing Markov chain [26]. If *one solution is sought*, then the absorbing Markov chain in Figure 3a is used. The two sink nodes ($s_0$ and $s_1$) represent the states of success and failure. After a finite number of steps, the process must enter one of these absorbing states. To find the expected cost, we need to calculate the expected number of times that the process is in transient states $s_2$ and $s_3$. In this example P, the transition matrix, Q, one of its submatrices denoting the process in the transient states, and R, another submatrix denoting the transitions from the transient states to the absorbing states, are

$$P = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & q_2 & 0 & p_2 \\ p_3 & 0 & q_3 & 0 \end{vmatrix} = \begin{vmatrix} I & 0 \\ R & Q \end{vmatrix}; \; Q = \begin{vmatrix} 0 & p_2 \\ q_3 & 0 \end{vmatrix}; \; R = \begin{vmatrix} 0 & q_2 \\ p_3 & 0 \end{vmatrix} \qquad (20)$$

Let $n_j$ be the expected total number of times that the process is in state $s_j$, and $M_i[n_j]$ be the mean of $n_j$ when the chain is started in $s_i$. From the theory of absorbing Markov chains [26], $N = \{M_i[n_j]\} = (I - Q)^{-1}$. N is called the *fundamental matrix*. In our example

$$N = \begin{vmatrix} \frac{1}{1-p_2q_3} & \frac{p_2}{1-p_2q_3} \\ \frac{q_3}{1-p_2q_3} & \frac{1}{1-p_2q_3} \end{vmatrix} \qquad (21)$$

As a result, the expected cost is $(c_2+p_2c_3)/(1-p_2q_3)$, where $c_i$ is the cost associated with state $s_i$. If B is searched before C, then A has expected cost

$$c_A(x_+,y_+) = \frac{c_B(x_+,z_+) + p_B(x_+,z_+) \cdot c_C(z_-,y_+)}{1 - p_B(x_+,z_+) \cdot q_C(z_-,y_+)} \qquad (22)$$

If subgoal C is searched before B, then A has expected cost

$$c_A'(x_+,y_+) = \frac{c_C(z_+,y_+) + p_C(z_+,y_+) \cdot c_B(x_+,z_-)}{1 - p_C(z_+,y_+) \cdot q_B(x_+,z_-)} \qquad (23)$$

Comparing $c_A$ and $c_A'$, the order with the smaller cost is used. Expected costs of clause A with variables in other modes can be computed similarly.

When *all solutions* in a subgoal have to be found, the process can also be modeled as an absorbing Markov chain. Figure 3b shows the absorbing Markov chain for the above example.

To compute the success probability of a clause, if $b_{i,j}$ is the probability that the process starting in transient state $s_i$ ends up in absorbing state $s_j$, then from the theory of absorbing Markov chains, $\{b_{i,j}\} = B = N \times R$. In our example, $b_{2,0} = p_2p_3/(1-p_2q_3)$, and $b_{2,1} = q_2/(1-p_2q_3)$. If subgoal B is searched first, then the success probability of node A is

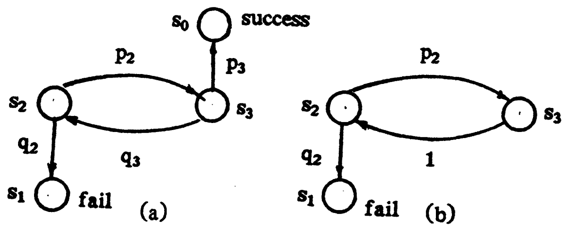


Figure 3. Example to compute the search cost and probability using an absorbing Markov chain.

$$p_A(x_+,y_+) = \frac{p_B(x_+,z_+)p_C(z_-,y_+)}{1 - p_B(x_+,z_+)q_C(z_-,y_+)} \qquad (24)$$

In general, if a subgoal has k variables, then $2^k$ combinations of probabilities and costs corresponding to all combinations of modes of the variables have to be found.

The above example illustrates the use of an absorbing Markov chain to order the search of descendents of an AND-node, which represents the evaluation of a clause. In contrast, to order the descendents of an OR-node, which represents the selection of multiple clauses with the same head, it is observed that once a descendent of an OR-node has been searched for a given binding-set, it will not be searched again. Unlike descendents of AND-nodes, there is no backtracking involved for a given binding-set. According to Theorem 2, descendents of an OR-node should, therefore, be searched in decreasing order of ratios of success probability to cost. The cost and probability of an OR-node can be computed in a similar fashion as in Eq. (15) when it has at least one consumer variable. When all its variables are producers, the average cost is taken as the average cost of each of its descendents weighted by the fraction of the total number of binding-sets that can be generated.

The basic idea in a systematic method to determinate an appropriate ordering of the subgoals and clauses is to associate with each subgoal and clause a table of the expected costs and probabilities for all combinations of modes of variables, and to use the appropriate costs and probabilities depending on the modes set for the variables. The best order with the minimum expected cost is chosen from all possible permutations of descendents. The number of permutations may be large. In this case heuristic information, such as the number of variables in a subgoal, can be used to eliminate inefficient candidate permutations. Note that the cost of each node in the AND/OR-tree representation depends only on the costs and probabilities of its descendents, provided that the descendents only depends on each other through shared variables. Here, the selection of the best order in a given level does not influence the computation of costs in levels above. That is, the computation of the minimum cost satisfies the principle of optimality, and the optimal order can be found by dynamic programming. In practice, subgoals are generally dependent on each other through shared clauses, which results in over-estimation of the costs. The proposed scheme is still applicable as a heuristic method to arrange the order in the search process. Statistic sampling has to be used to estimate the cost and probability of a node after the order of its descendents is determined. This reduces the accumulation of errors as nodes in higher levels of the AND/OR-tree are ordered.

A final point on the ordering of nodes in the AND/OR-tree representation of logic programs is that different orders may be found depending on the modes of the variables. Either an 'average' order may be used or multiple program statements may be generated for different cases to reflect the preferred order.

## 4. CONCLUSIONS

In this paper we have studied the computational efficiency of parallel and ordered depth-first searches to solve optimization and decision problems. The performance bounds and conditions to cope with anomalies in searching optimization problems represented as OR-trees have been derived and verified by simulations. Speedups have been found to be related to the problem complexity and the number of incumbents obtained during the search process. For a problem with a high complexity and a small number of incumbents, such as integer programming problems, a near-linear speedups can be achieved with respect to a large number of processors.

An ordered depth-first search strategy has been studied with respect to decision problems represented as AND/OR-trees. When the success probabilities and costs of sibling nodes are independent of each other, and a node, once searched, will not be searched again, the sibling nodes should be ordered according to ratios of probability and cost to minimize the expected total search cost. Due to shared clauses and variables in a Prolog program and that backtracking is allowed, it is difficult to find the optimal depth-first search order. An absorbing Markov chain to model the effects of backtracking and a dynamic programming method to order the search have been developed.

## ACKNOWLEDGEMENTS

The authors would like to thank Mr. Mark Gooley for generating the results in Figure 2 and Table 2.

## REFERENCES

[1] J. Barnett, "Optimal Searching from AND Nodes," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 786-788, William Kaufman, Inc., Los Altos, CA, 1983.

[2] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 551-560, ICOT and North-Holland, 1984.

[3] M. Garey, "Optimal Task Sequencing with Precedence Constraints," *Discrete Mathematics*, vol. 4, no. 1, pp. 37-56, 1973.

[4] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms," *Int'l J. of Computer and Information Sciences*, vol. 5, no. 4, pp. 315-343, Plenum Press, 1976.

[5] T. Ibaraki, "Computational Efficiency of Approximate Branch-and-Bound Algorithms," *Math. of Oper. Research*, vol. 1, no. 3, pp. 287-298, Inst. of Management Sciences, 1976.

[6] M. Imai and T. Fukumura, "A Parallelized Branch-and-Bound Algorithm Implementation and Efficiency," *Systems, Computers, Controls*, vol. 10, no. 3, pp. 62-70, Scripta Publishing, June 1979.

[7] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[8] V. Kumar and L. N. Kanal, "A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures," *Artificial Intelligence*, vol. 21, no. 1-2, pp. 179-198, North-Holland, 1983.

[9] T. H. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Comm. of the ACM*, vol. 27, no. 6, pp. 594-602, ACM, June 1984.

[10] G.-J. Li and B. W. Wah, "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms," *Proc. Int'l Conf. on Parallel Processing*, pp. 473-480, IEEE, 1984.

[11] G.-J. Li and B. W. Wah, "How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms," *Proc. National Conf. on Artificial Intelligence*, pp. 212-215, AAAI, 1984.

[12] G.-J. Li and B. W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *Proc. Int'l Conf. on Parallel Processing*, pp. 123-130, IEEE, June 1985.

[13] G.-J. Li, *Parallel Processing of Combinatorial Search Problems*, Ph.D. Dissertation, Purdue University, W. Lafayette, IN, Dec. 1985.

[14] G.-J. Li and B. W. Wah, "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. 1986 Fall Joint Computer Conference*, ACM-IEEE, Nov. 1986.

[15] G.-J. Li and B. W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *Trans. on Computers*, vol. C-35, no. 6, pp. 568-573, IEEE, June 1986.

[16] G.-J. Li and B. W. Wah, *Computational Efficiency of Combinatorial OR-Tree Searches*, to appear in IEEE Trans. on Software Engineering, 1986.

[17] J. Mohan, "Experience with Two Parallel Programs Solving the Traveling-Salesman Problem," *Proc. 1983 Int'l Conf. on Parallel Processing*, pp. 191-193, 1983.

[18] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The Architecture of a Parallel Inference Engine (PIE)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479-488, ICOT and North-Holland, 1984.

[19] D. Nau, V. Kumar, and L. Kanal, "General Branch and Bound and its Relation to A$^*$ and AO$^*$," *Artificial Intelligence*, vol. 23, no. 1, pp. 29-58, North-Holland, 1984.

[20] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[21] H. A. Simon and J. B. Kadane, "Optimal Problem-Solving Search: All-or-None Solutions," *Artificial Intelligence*, vol. 6, no. 3, pp. 235-247, North-Holland, 1975.

[22] B. W. Wah and Y. W. E. Ma, "MANIP--A Parallel Computer System for Implementing Branch and Bound Algorithms," *Proc. 8th Annual Symp. on Computer Architecture*, pp. 239-262, ACM, 1981.

[23] B. W. Wah and Y. W. E. Ma, "MANIP--A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *Trans. on Computers*, vol. C-33, no. 5, pp. 377-390, IEEE, May 1984.

[24] B. W. Wah and C. F. Yu, "Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search," *Trans. on Software Engineering*, vol. SE-11, no. 9, pp. 922-934, IEEE, Sept. 1985.

[25] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *Computer*, vol. 18, no. 6, pp. 93-108, IEEE, June 1985.

[26] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, D. Van Nostrand Company, Inc., New York, NY, 1965.

# Counting and Packing in Parallel

Joseph Gil and Larry Rudolph

Department of Computer Science, Hebrew University
Jerusalem, Israel

ARPAnet Address: rudolph%hujics.bitnet@wiscvm

## ABSTRACT

Given an array of size $n$ with entries either marked or unmarked, we compute the number of marked entries and assign to each (marked entry) a unique index between one and the number of marked entries. If there are $k$ marked entries, then our algorithms require $O(\log k)$ time, which is independent of $n$, using $n$ processors and a CRCW model.

## 1. Introduction

As the study of parallel algorithms matures, it is important to develop efficient basic techniques that can be used as building blocks for more complex algorithms. One basic algorithm is to compute the size of a subset and pack it into an array. That is, given an array with entries either marked or unmarked, compute the number of marked entries and assign to each (marked entry) a unique index between one and the number of marked entries.

Assume that there are $k$ marked entries in an array of size $n$. We are given $n$ processors and a CRCW model of parallel computation. That is, each processor can read or write any cell in shared memory in a single cycle. If two or more processors try to write to the same cell during the same cycle, then one arbitrarily wins. Actually, our algorithms require a slightly weaker model in which two processors are allowed concurrent write of common values only.

Using a simple binary summation tree, it is easy to solve this problem in $O(\log n)$ time. On the other hand, using a simple locking mechanism, it is also easy to solve the problem in time $O(k)$; each processor is assigned a unique cell of the array and if the entry is a one, then the processor locks a counter variable. When the lock is granted, the processor just increments the counter by one and unlocks the variable.

We desire a solution whose running time is independent of $n$ and is logarithmic in $k$. Such a solution already exists, however, it uses randomness and assumes the stronger model of CRCW. Rudolph and Steiger [2] give a parallel algorithm that terminates in $O(\log k)$ time with high probability. Moreover, they use only $k$ processors. We present a deterministic solution to this problem with the same time bounds. Note that a trivial consequence of our result yields an algorithm for computing parity.

The outline of the paper is as follows. We first describe an algorithm for computing the the "prefix maxima" of an array. This is then used to develop an $O(\log\log n + \log k)$ time algorithm. For $k \geq \log n$ this algorithm gives the desired result. We then present a second algorithm for the case of $k < \log n$. Running the two algorithms in tandem gives the final result.

## 2. Prefix Maxima

Given an array $A[1..n]$ of numbers, we define prefix maxima, $MAX_i(A)$, to be a set of $n$ values, the i-th of which is the maximum of the first $i$ elements of $A$:

$$MAX_i(A) = \text{Maximum}\{A[1], ..., A[i]\} : 1 \leq i \leq n.$$

A simple sequential scan algorithm solves this problem in $O(n)$ steps; this section gives a parallel algorithm using $n$ processors and terminates within $O(\log\log n)$ parallel steps.

**Lemma 1:** The prefix maxima vector can be found in $O(1)$ parallel steps using $n(n^2-1)/6$ processors and a CRCW model of parallel computation.

**Proof:** Note that we can compute all $MAX_i(A)$ in parallel. For the i-th prefix of $A$ we do the following:

First, $i(i-1)/2$ processors perform all distinct comparisons between $A[x]$ and $A[y]$, $1 \leq x < y \leq i$, in one parallel step. Next, the same number of processors check, for each $j$, $1 \leq j \leq i$, whether $A[j] = \text{Maximum}\{A[1], ..., A[i]\}$ by parallel computation of an *AND* function. Writing the correct value of $MAX_i(A)$ requires one more step.

The above procedure can be performed in parallel for all $i$, $1 \leq i \leq n$, with $F(n)$ processors, where

$$F(n) = \sum_{i=1}^{n} i(i-1)/2 = n(n^2-1)/6 .$$

If, on the other hand, there are fewer than a cubic number of processors, but at least a linear number, i.e. $n \leq p \leq F(n)$, an adaptation of the max computation of Valiant [4] (and as implemented by Shiloach and Vishkin [3]) can be used. The following algorithm evaluates the array $M[1..n]$ for the input $A$ such that $M[i] = MAX_i(A)$:

0190-3918/86/0000/1000 $01.00 © 1986 IEEE

```
Procedure $PreMax(p, A, n, M)$
    Step 1: If $p \geq F(n)$ then use lemma 1 to compute $M[1..n]$ in one parallel step and return.
    Step 2: Divide the array $A[1..n]$ into minimal number $l$ of distinct segments $S_1, .. S_l$ which satisfy the following conditions [3]:
        $(i)$ $|S_i| - |S_j| \leq 1$ for $1 \leq i \leq j \leq l$
        $(ii)$ $\sum_{i=1}^{l} F(|S_i|) \leq p$
    Step 3: Let $b_i$, $e_i$, $n_i$, be the indices of the beginning, ending, and size of each segment $S_i$.
    Step 4: For each $i$ in $[1..l]$ do in parallel
            $PreMax(F(n_i), A[b_i .. e_i], n_i, M[b_i .. e_i])$
    Step 5: Let $A'[1 .. l]$ be the maximum value in each segment:
        For each $i$ in $[1..l]$ do in parallel:        $A'[i] = M[e_i]$
    Step 6: Apply the algorithm recursively on $A'$:
            $PreMax(p, A', l, M')$
    Step 7: Merge results from steps 4 and 6:
        For each $i$ in $[2..l]$ and $j$ in $[b_i .. e_i]$ do in parallel
            $M[j] := \text{Maximum}(M[j], M'[i-1])$
```

It is not hard to see that this algorithm terminates in $O(\log\log n)$ parallel time: Step 1 requires constant time as shown in lemma 1. Finding $l$ and then computing $b_i$ and $e_i$ for all $i$: $1 \leq i \leq l$ in steps 2 and 3 require $O(1)$, i.e. constant, parallel time. The recursive call in step 4 is terminated in constant time since there are enough processors for each segment to compute all the prefix maxima in constant parallel time as in step 1. The exact allocation of processors in step 4 may require some computational work, but it can be done by the first $l$ processors in constant time. Steps 5 and 7 again take constant parallel time.

Note that step 7 correctly evaluates $M[1..n]$: At the end of step 6, each $M[j]$ contains the maximal value of $A[b_j]..A[e_j]$. $M'[j-1]$ is the largest of $M[e_1]..M[e_{j-1}]$. Since these $M$ values are the maximal in their segment, then at the end of step 7, $M[j]$ is the biggest of $A[1..j]$.

The algorithm requires constant parallel time except for the recursive call in step 6 on the auxiliary array $A'$. We need to bound the number of recursions. The recursion terminates when the number of processors, $p$, is larger than the number of items cubed. Let $T(\alpha,n)$ denote the parallel time for running the algorithm on $n$ values using $\alpha n$ processors. From the definitions it is easy to check that

$$\sum_{j=1}^{l} F(|S_j|) \leq n^3/l^2 \text{ for all } l \leq n.$$

We can bound the value of $l \leq l_0$ where $n^3/l_0^2 = p$ Thus, $l$, the size of the array in the next iteration would be smaller than $n/\alpha^{½}$. Since, the number of items drops by a factor of $\alpha^{-1/2}$ and the number of processors remains constant, then the new ratio between processors and items will be $\alpha^{3/2}$. Thus we get:

$$T(\alpha,n) = 1 + T(\alpha^{3/2}, n/\alpha^2)$$

The recursion is guaranteed to terminate when $\alpha$ is bigger than $n^2$, and this will occur within

$$\frac{(\log\log n^2 - \log\log\alpha)}{\log\log(3/2)} = O(\log\log n) \text{ steps}.$$

If $1 \leq \alpha < 2$, the above equation is ill defined. We can force $\alpha$ to be 2 by using virtual processors: some or all of the processors will act like 2 virtual processors and thereby raising the processor to item ratio and only doubling the time. A more careful and much more tedious counting shows that the number of steps required even in the case of $\alpha=1$ is $O(\log\log n)$.

For $p \leq n$ divide the array into $p$ almost equal segments. Assign one processor to each segment and, in parallel, compute the prefix maxima on each segment sequentially. Then compute the prefix maxima of the maximum values of each segment; there are $p$ of them. Computing the prefix maxima of the full array can be done as in step 7 in the previous algorithm. The running time is $O(n/p) + O(\log\log p)$.

## 3. Computing the Subset Size in Parallel

The Subset size problem is to find the number of elements in a set which satisfy some predicate. This can be formulated as follows: Given an array $A[1..n]$ with binary values, compute how many entries contain the value 1. The subset will be denoted by $S$ and its size by $k$. In addition, each entry with a value of 1 is to be assigned a unique integer in the range 1 to $n$.

We present two algorithms for solving this problem. One is efficient when the number of set elements is large (SubSet_I) and the other (SubSet_II) is efficient when it is small. Both these algorithms can be executed in parallel to ensure an efficient algorithm for any set cardinality.

### 3.1. Algorithm Subset_I

As usual we will assume $p \geq n$. First we reduce the problem of finding subset size to the prefix maxima problem. Then, we form a linked list of the subset elements. Finally, we apply recursive doubling to compute the size and assign ordinal numbers to each element.

The following algorithm outputs $k$, and the array $R[1..k]$ where where $D[R[i]]$ is the i-th element of $D$ that contains a 1. The arrays $A[1..n+1]$ and $P[1..n+1]$ are used as temporary storage.

```
Procedure $SubSet\_I(p, D, n, R, k)$.
    Step 1: Compute the auxiliary array $A[1..n+1]$ as follows:
        For each $i$ in $[1..n+1]$ do in parallel:
            If $i > 1$ and $D[i-1]=1$ Then $A[i] := i-1$
            Else $A[i] := 0$
    Step 2: Apply the previous algorithm to produce the linked list
            $PreMax(p, A, n, P)$
        Note that $P[i]$ will come to contain the max value of all the elements in cells $A[j]$, where $j < i$. This value is also the index of the first cell containing a 1 to the left (smaller index) of $D[i]$.
    Step 3: Apply recursive doubling to the linked list represented by the array $P[1..n]$. The head of the list is stored in $P[n+1]$. This will result in the array $R$ of the requested form and will yield the value $k$.
```

It is easy to see that $SubSet\_I$ requires $O(\log\log n + \log k)$ time using $n$ processors: Step 1 requires $O(1)$ time, step 2 requires $O(\log\log n)$ time as previously shown, and step 3 can be done in logarithmic time of the length of the linked list, i.e. $O(\log k)$.

### 3.2. Algorithm SubSet_II

When $k \le \log n$ the $\log\log n$ term in running time of *SubSet_I* dominates. Fortunately, when $k$ is this small, we can create the linked list faster. The basic idea is to compact the set elements into a smaller array so that more processors can be used to quickly solve the problem. Algorithm *SubSet_II* gives the precise details of the computation of $k$ and $R[1..k]$ as before. The algorithm uses the temporary arrays: $D', D'', R', R''$, size of which will be determined at run time.

```
Procedure SubSet_II $(p, D, n, R, k)$
    Step 1: If $p \ge F(n)$ then apply SubSet_I to D:
                $SubSet\_I(p, D, n, P, k)$
    Step 2: Find the maximal $l$ for which $p \ge F(l)$.
    Step 3: Split the array $D[1..n]$ into $l$ disjoint consecutive segments $S$ which may vary in size by no more than 1.
    Step 4: Compute the array $D'[1..l]$ as follows:
        for each $j$ in $[1..l]$ and $i$ in $[1..|S_j|]$ do in parallel:
            If $S_j[i] = 1$ Then $D'[j] := 1$ Else $D'[j] := 0$
    Step 5: Apply algorithm SubSet_I to $D'$:
                $SubSet\_I(p, D', n, P', k')$
    Step 6: Compress the array $D$ into $D''$, using the ordinal number of non empty segments, as found in step 5. $D''[1..k' * \lceil (n/l) \rceil]$, will contain only the segments $S_j$ of $D$ before step 6, that have at least one element set to 1 in them.
    Step 7: Apply SubSet_II recursively to $D''$:
                $SubSet\_II(p, D'', k' * \lceil (n/l) \rceil, R'', k)$
    Step 8: Map $R''$ into $R$ to fit the original indexes of $D$ (before the compression in step 6.
```

As before steps 2 and 3 take a constant parallel time. Step 4 is really a computation of an OR function which can be done in parallel in constant time. In step 6 and 8 we make a compressed copy of an array and compute the opposite transformation. Since the ordinal number of each segment copied is given in the array $P'$, those steps can be done in parallel using $n$ processors in constant time.

In step 1 and 5 we apply algorithm *SubSet_I*, there are always enough processors to do the linking phase in it in a constant time. The counting in *SubSet_I* takes $O(\log k)$ in step 1, and $O(\log k')$ in step 5. Since $k' < k$ we can conclude that the total running time of *SubSet_II* is $O(\log k)$ except for the recursive call in step 7.

Since $F(l) \le l^3$ and $p \ge n$ we have $l < n^{1/3}$ at step 2. The algorithm is applied recursively in step 7 to an array sized $k' \lceil (n/l) \rceil$. Denote the size of the array after the $i^{th}$ recursive iteration by $n_i$. It is clear that $n_i \le n$ for all $i > 0$, now $n_0 = n$, $n_1 \le k n_0^{2/3}$, $n_2 \le k n_1^{2/3}$. A simple induction gives:

$$n_i \le k^3 n^{(2/3)^i}$$

The recursive calls are guaranteed to stop if $n_i \le n_0^3 \le p$. For $k \le n^\beta < n^3$ the recursion will terminate in the $\log(1/3 - 3\beta) - \log(2/3)$ step. The total running time of the algorithm is therefore $O(C_\beta \log k)$ were $C_\beta$ is a constant depending only on $\beta$. For example taking $\beta$=0.07 would give $C_\beta$=5.

Note that asymptoticly $\log n < n^\beta$ for any $\beta$. This algorithm can be used in those application where the $k$ is known in advance to be smaller than $\log n$. When no estimate can be made about $k$, one would run the two previous algorithms in parallel, and terminate as soon as one of them terminates, requiring a total time of $O(\log k)$ for any size $k$.

For $p < n$ creating the linked list and computing the size of the subset can be done by dividing the array into $p$ segments. Each processor will make a linked list and will count the number of elements in his segment. After that Use the *SubSet_I* and *SubSet_II* algorithms for chaining the linked lists of all segments which contain one or more elements from the subset. The total running time is therefor $O(n/p) + O(\log k)$.

### 4. Conclusion

We view the algorithms presented as building blocks to more complex ones. One application is when there is a multiway predicate, and subsets of the processors are to go off and solve independent subproblems. Before they can work together, however, they must know how many processors are cooperating, it is advantageous if each has a unique index.

The algorithm can be improved somewhat, but only by a constant factor, by using a technique to compute the maximum of $n$ items, all in the range 1 to $n$ with $n$ processors in constant time. (Note that in this paper we require about $n^2$ processors to compute the max.) This result can be found in a fuller version of this paper [1].

### References

[1] Gil, J. and L. Rudolph, "Counting, Packing, and Maximum in Parallel," Technical Report, Department of Computer Science, The Hebrew University, Jerusalem, Israel, 1986.

[2] Rudolph, L. and W. Steiger, "Subset Size in Parallel," *1985 International Conference on Parallel Processing*, pp. 11-13.

[3] Shiloach, Y. and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computational Model," *Journal of Algorithms* vol. 2, no. 1, 1981, pp. 88-102.

[4] Valiant, L. "Parallelism in Comparison Problems," *SIAM Journal of Computing* vol. 4, no 3, 1075, pp. 348-355

# REDUCTION TECHNIQUES FOR SELECTION IN DISTRIBUTED FILES

**Nicola Santoro and Ed Suen**

Distributed Computing Group
Carleton University
Ottawa, Ontario, K1S 5B6
Canada

## Abstract

The problem of selecting the K-th smallest element of a set of N elements distributed among d sites of a communication network is examined. A distributed reduction technique for selection is a distributed algorithm which tranforms this problem to an equivalent one where either K or N (or both) are reduced. A collection of distributed reduction techniques is presented; the combined use of these algorithms yield new solutions for the selection problem for both point-to-point and shout-echo networks. In particular, $O(d + \delta \log(\kappa/d))$ and $O(\log \kappa)$ upper-bounds on the distributed selection problem are derived for point-to-point networks having a spanning star graph (e.g., complete networks) and shout-echo networks, respectively, where $\kappa \le \Delta = \mathrm{Min}\{K,N-K+1\} \le K$ and $\delta \le d$. These results represent an improvement on the existing bounds for those networks; furthermore the existing selection algorithms for other point-to-point networks (e.g., rings, meshes, trees) can be made more efficient by using one of the proposed reduction techniques as a pre-processing phase.

## 1. Introduction

The classical problem of selecting the K-th smallest element of a set F drawn from a totally ordered set has been extensively studied in serial and parallel environments. In a distributed context, it has different formulations and complexity measures.

A ***communication network*** of size d is a set $S=\{S_1, \dots, S_d\}$ of ***sites***, where each site has a local non-shared memory as well as processing and communication capabilities.

A ***file*** of cardinality N is a set $F=\{f_1,\dots,f_N\}$ of records, where each record $f \in F$ contains a unique key k(f) drawn from a totally ordered set; for convenience, $k(f_i)<k(f_j)$ shall be denoted by $f_i<f_j$.

A ***distribution*** of F on S is a d-tuple $X=<X_1,\dots,X_d>$ where $X_i \subseteq F$ is a subfile stored at site $S_i$, $X_i \cap X_j= \emptyset$ for $i \neq j$, and $\cup_i X_i = F$.

Order-statistics queries about F can be originated at any site and will activate a query resolution process at that site. Since only a subset of F is available at each site, the resolution of a query will in general require the cooperation of several (possibly all) sites according to some predetermined algorithm. Since local processing time is usually negligible when compared with transmission and queueing delays, the goal is to design resolution algorithms which minimize the amount of communication activity rather than the amount of processing activity.

The **distributed selection problem** is the general problem of resolving a query for locating the K-th smallest element of F. The tuple <N, K, N[1], ..., N[d]> is called the problem ***configuration***, and $\Delta = \mathrm{Min}\{K,N-K+1\}$ is called the problem ***size***, where N[i] is the cardinality of $X_i$. Any efficient solution to this problem can be employed as a building block for a distributed sorting algorithm [9].

The complexity of this problem (i.e., the number of communication activities required to resolve an order-statistics query) depends on many parameters, including the number of sites, d, the size N of the file, the number N[i] of elements stored at site $S_i$, the rank K of the element being sought, and the topology and type of the network.

Solutions to this problem have been developed for both the point-to-point and the shout-echo models of computation.

In ***point-to-point*** networks, associated with S is a set $L \subseteq S \times S$ of direct communication lines between sites; if $(S_i,S_j) \in L$, $S_i$ and $S_j$ are said to be ***neighbours***. Sites communicate by sending messages; a message can only be sent to and received from a neighbour. The couple G=(S,L) can be thought of as an undirected graph; hence, graph-theoretical notation can be employed in the design and analysis of distributed algorithms in the point-to-point model. Different solutions and bounds exist for the distributed selection problem in the point-to-point network depending on the topology of the network [2,6,15]. In particular,

0190-3918/86/0000/1003 $01.00 © 1986 IEEE

Frederickson [2] presented algorithms which require O(d $(\log d)^2 \log N$) messages in a *ring*, O(d $(\log d)^{1/2} \log N$) messages in a *mesh*, and O(d log(2N/d)) messages in a *complete binary tree*. All these bounds apply to the worst case; the analysis of the expected communication complexity has been carried out in [11,15].

In ***shout-echo*** networks, each site can broadcast a message and reply to a received message; the process of broadcasting a message and receiving a (possibly distinct) reply from all sites constitutes a basic communication activity called shout-echo. Several papers investigating the selection problem in this network have appeared [1,5,7,8,10-14]; an O(log Δ) bound has been established by Marberg and Gafni [5] who also prove a $\Omega(\log n_2)$ lower bound if the messages are constrained to carry only file elements, where $n_2$ is the second largest number of elements stored at the sites.

In this paper, a selection procedure based on various reduction techniques is proposed, and used to obtain efficient solutions both in the shout-echo and in the point-to-point models. The algorithm is based on the serial technique developed by Frederickson and Johnson for selection in an array with sorted columns [3], and employs some of the existing distributed selection algorithms as subroutines. Using the proposed algorithm, the following upper-bounds on the distributed selection problem are obtained:

1) 9.64 δ log(κ/d) + O(d) messages in networks having a spanning star graph (e.g., star networks, complete networks);
2) 6.82 logκ + O(1) basic communication activities in a shout-echo network;

where $\kappa \le \Delta$ and $\delta \le p = \mathrm{Min}\{\Delta, d\}$ are, respectively, the rank of the desired element and the number of sites under consideration in the problem configuration produced from the execution of Procedure SET-UP (see section 2.1). These results improve the existing bounds of 19.28 p log Δ - 9.64 p log d + O(d) and 10.64 log Δ + O(1) for these networks, respectively. Note that, while always $\kappa \le \Delta$, sometimes $\kappa << \Delta$. Consider for example the configuration <10032,5937,10000,20,5,5,2>; in this case κ=33 while Δ=Min{K,N-K+1}=4096.

Furthermore, it is observed that selective application of the proposed reduction techniques can also yield slight improvements to the existing bounds for rings, meshes, and complete binary trees established in [2].

The paper is organized as follows. In the next section a collection of reduction techniques is presented and analysed; the combined use of these techniques yields a new selection procedure. In section 3, this procedure is employed to obtain the new bounds in the shout-echo networks. Finally, in section 4, the application of some of the reduction techniques to point-to-point networks is discussed.

## 2. A selection procedure

In this section, a selection algorithm is proposed which can be used to obtain efficient solutions both in the shout-echo and in the point-to-point models.

The algorithm consists of four phases: **set-up, cut, filtration**, and **termination**; the first three phases are reduction techniques. The set-up phase (procedure SET-UP) enforces a set of constraints on the problem configuration; this might require a modification of the entire problem configuration. The cut phase (procedures CUT1 and CUT2) has the overall effect of further reducing the problem configuration so that the cardinality of the set of elements among which the sought element is to be found is linear in the rank of this element. Finally, the filtration phase (procedure FILTER) iteratively reduces the problem under consideration to one that can be efficiently solved at just one site by the termination phase. Assume, without loss of generality, that the original problem is finding the K-th smallest element (a "min" problem). The symmetrical problem of finding the K-th largest (a "max" problem) can be treated in a similar manner.

### 2.1 Set-up phase

A problem configuration <N, K, N[1],...,N[d]> is said to be **regular** if N[i]≤Δ for 1≤i≤d, where Δ=Min{K,N-K+1} is the problem size. The goal of the SET-UP procedure is to tranform the initial problem configuration to a regular one.

Let C=<N,K, N[1],..., N[d]> be the initial problem configuration and, without loss of generality, let Δ = K. If this configuration is not regular, then for all sites $S_i$ with N[i]>K the N[i]-K largest elements can obviously be removed from consideration without altering the problem integrity (i.e., the K-th smallest element will have the same rank among the remaining elements); this observation has also been made in [3,5]. After this simplification, the resulting configuration will be C'=<N',K, N'[1],..., N'[d]> where N'[i]=Min{N[i],K} and $N'=\sum_i N'[i]$.

It is possible, however, that this new configuration is not regular once we change the problem type (i.e., select the

$(N'-K+1)$-th largest instead of the $K$-th smallest element). For example, it may occur that $\Delta'=\mathrm{Min}\{K,N'-K+1\}<K$ and that $N'[i]>\Delta'$ for some i; in this case, the $N'[i]-\Delta'$ *smallest* elements can still be removed from consideration from that set without altering the problem integrity (i.e., the $(N-K+1)$-th largest among the original element will be the $\Delta'$-th largest of the remaining elements). Notice that such a modification will change the value of $N'$ and thus possibly $K'$; hence, the new configuration might again be not regular.

In the following, the variable $T_j$ will represent the type of the problem at iteration j: $T_j$="max" ("min") will indicate that the $K_j$ largest (smallest) element is to be determined; the function *other* on problem types is defined as follows: *other*["max"]="min" and *other*["min"]="max".

```
PROCEDURE SET-UP
initialization:
        $N_0:= N; K_0:=K; N_{i,0}:=N[i]; T_0=$"min";
j-th iteration ($j\geq1$):
        $K_j := \mathrm{Min}\{N_{j-1} - K_{j-1}+1, K_{j-1}\}$;
        if $K_j=K_{j-1}$ then $T_j:=T_{j-1}$ else $T_j:=$other$[T_{j-1}]$;
        $N_{i,j}:= \mathrm{Min}\{N_{i,j-1}, K_j\}$;
                <Meaning: remove from consideration
                the $K_j-N_{i,j-1}$ largest (if $T_j$="min") or
                smallest (if $T_j$="max") elements still
                under consideration at site $S_i$>
        $N_j:= \Sigma_i N_{i,j}$;
        $\beta_j=|\{ N_{i,j-1}>K_j\}|$;
        if $\beta_j>0$ perform the next iteration, otherwise
        execute the termination step.
termination:
        $d:= \mathrm{Min}\{d,K_j\}$.
                <Meaning: If $d>K_j$ and $T_j$="min"
                ("max"), consider the smallest
                (largest) element still under
                consideration at each site; among these
                elements, identify the $d-K_j$ largest
                (smallest) and remove from
                consideration the corresponding sites>
END SET-UP
```

It will now be shown that the procedure will iterate a constant number of times. The j-th iteration will be called a **flip** if $K_j=N_{j-1}- K_{j-1}+1<K_{j-1}$; i.e, if the type of the problem is changed.

**Property 1** Let $j\geq1$. If the $(j+1)$-th iteration is not a flip, the procedure terminates at that iteration.

Proof. If the $(j+1)$-th iteration is not a flip, $K_{j+1}=K_j\geq N_{i,j}$; hence $\beta_{j+1}=0$ and the algorithm terminates. ♦

**Property 2** Let $j>1$. If the $(j+1)$-th iteration is a flip, then $\beta_j=1$.

Proof. Let the $(j+1)$-th iteration be a flip; then $\beta_j>0$, otherwise the algorithm would have terminated at the j-th iteration. Let $\beta_j>1$, and let $S_a$ and $S_b$ be two sites with $N_{a,j-1}>K_j$ and $N_{b,j-1}>K_j$; then, $N_j\geq N_{a,j} + N_{b,j}= 2 K_j$. That is, $N_j-K_j+1\geq K_j+1>K_j$ which would imply $K_{j+1}=\mathrm{Min}\{K_j, N_j-K_j+1\} = K_j$ contradicting the fact that the $(j+1)$-th iteration was a flip. Thus, $\beta_j=1$. ♦

**Property 3** Let $j>1$. Then the j-th and the $(j+1)$th iterations cannot both be flips

Proof. By contradiction, let both the j-th and the $(j+1)$th iterations be flips; that is

$$K_j = N_{j-1}-K_{j-1}+1 < K_{j-1} \qquad (1)$$

and

$$K_{j+1} = N_j-K_j+1 < K_j \qquad (2)$$

By property 2, $\beta_{j-1}=1$; that is, only one site, say $S_a$, had been reduced in the $(j-1)$-th iteration. Since $N_{a,j-1}=K_{j-1}$, site $S_a$ will be reduced also in the j-th iteration; and since iteration $(j+1)$ is a flip, then (by property 2) $\beta_j=1$; that is, $S_a$ will be the *only* site to be reduced in the j-th iteration. Thus, $N_j=N_{j-1}-(N_{a,j-1}-K_j)= N_{j-1}-K_{j-1}+K_j$; by substituting this expression for $N_j$ in inequality (2), it follows $K_j > N_j -K_j +1= N_{j-1}-K_{j-1}+K_j -K_j +1 =N_{j-1}-K_{j-1}+1$ contradicting relation (1). ♦

**Theorem 1** Procedure SET-UP performs at most three iterations.

Proof. Let it perform at least two iterations. If the second iteration is not a flip, (by property 1) the algorithm terminates at the end of this iteration; on the other hand, if the second iteration is a flip then (by property 3) the third iteration is not a flip and (by property 1) the procedure terminates after the third iteration. ♦

**Theorem 2** Let $\kappa$ and $\delta$ be, respectively, the rank of the desired element and the number of sites under consideration in the problem configuration produced by the Procedure SET-UP. Then, there will be at most $\kappa\delta \leq \kappa^2$ elements still left under consideration.

Proof. The configuration obtained from Procedure SET-UP is regular; i.e., at most $\kappa$ elements are still under considerations at each site. Since in the termination step of the procedure $\delta \leq \kappa$ sites are kept under consideration, the theorem follows. ♦

### 2.2 Cut phase

Let $<N, K, N[1], ..., N[d]>$ be the problem configuration. If this configuration is the result of the SET-UP phase, then $N[i] \leq K = \kappa$, $1 \leq i \leq d$, and $d = \delta \leq \kappa$. The CUT phase is the distributed translation of the CUT procedure of the algorithm by Frederickson and Johnson for selection in matrices with sorted columns [2]. This phase is composed of two sub-phases. The initial sub-phase CUT1 reduces the number of elements under consideration to at most $\mathrm{Min}\{K \log d, K \log n\}$, where $n = \mathrm{Max}\{N[i] \mid 1 \leq i \leq d\}$; sub-phase CUT2 guarantees that at most 4.5K elements remain.

The j-th iteration of procedure CUT1 is applied only to a subset, $\xi^j$, of the sites. The $i_j = \lceil 2^j(K+1)/d \rceil$ - th smallest element from each of these sites is collected. If a site has less than $i_j$ elements then $+\infty$ will be used. Among these values, the median is found and all values at all sites greater than or equal to the median are removed from consideration. The next iteration is then applied to half the sites participating in this iteration.

Using the values $i_j$ produced by CUT1 ($0 \leq j \leq q$), each subfile $X_m$ of site $S_m$ can be broken down into the subsections $\Gamma_{m,j} = \{x \mid x \in X_m, S_m[i_j] \leq x < S_m[i_{j+1}]\}$, where $S_k[w]$ denotes the w-th smallest element at $S_k$ and $i_0=1$. Assign the weight $i_{j+1}$-$i_j$ to the initial $S_m[i_j]$ subsection element. Procedure CUT2 operates on this set of initial subsection elements $\{S_m[i_j] \mid 1\leq m \leq d, 0 \leq j \leq q\}$; among these, it finds an element $x^*$ whose overall rank is at least K and at most 4.5K, using a weighted selection algorithm (e.g., see [4]). Finally all elements larger than $x^*$ are removed from consideration.

**PROCEDURE CUT1**
(to be performed only if N>4.5K)

```
1. Initialization
        $\xi^0 := S$; $i_0 := 1$.

2. Stage j  ($1 \leq j \leq q$)
    2.0  IF $\lceil 2^j(K + 1)/d \rceil > n$ or $|\xi^{j-1}| \leq 1$ THEN
             go to the termination step.
    2.1  $i_j := \lceil 2^j(K+1)/d \rceil$;
    2.2  Let $V^j$ denote the set of all $S_i[i_j]$, the $i_j$th
         ranked element of site $S_i$, where $S_i \in \xi^{j-1}$.
         Find the median v of $V^j$
    2.3  Send the median v to all sites.
         Upon receipt of this median v, each site
         eliminates all elements not smaller than v.
    2.4  Let $\xi^j = \{S_i \mid S_i \in \xi^{j-1}$ and $s_i[i_j]<v\}$.
         Perform the next iteration by repeating step 2.

3.Termination
        Let $i_{q+1} = n + 1$

END CUT1
```

**Property 4.** The K-th element of the original problem is also the K-th element of the result of CUT1.

Proof. At step j, $V^j$ contains at least $d/2^{j-1}$ elements, each having rank at least $2^j(K+1)/d$. Thus the median of $V^j$ has overall rank at least $(2^j(K+1)/d)(d/2^j) = (K+1)$; that is, the K-th smallest element must be smaller than the median of $V^j$. Since we only remove elements which are greater or equal than the median, the lemma follows. ♦

**Property 5** The number of stages of procedure CUT1 is at most $\mathrm{Min}(\lceil \log d \rceil, \lceil \log n \rceil)$.

Proof. There are two conditions for termination in step 2.0. The first condition, $\lceil 2^j(k+1)/d \rceil > n$, guarantees that at most $j < \lceil \log n \rceil$ iterations are performed. Since $|\xi^j| \leq \lfloor |\xi^{j-1}|/2 \rfloor$, for $j \geq 1$, the second condition, $|\xi^{j-1}| \leq 1$, ensures at most $\lceil \log d \rceil$ iterations. ♦

**PROCEDURE CUT2**
(to be performed only if N>4.5K)
Let $V^0$ denote the $i_0$th element from every site $S_i \in \xi^0$.

```
1    Select $x^*$ from the elements $\cup_{j=0,q} V^j$ by a 4K-th
     weighted selection, where an element $x \in V^j$
     has weight $i_{j+1} - i_j$.

2    Send the element $x^*$ to all sites. Upon receipt of
     this element $x^*$, each site eliminates all elements
     larger than $x^*$.

END CUT2
```

**Property 6** If K is the rank of the element to be selected, then CUT2 retains at most

4.5K elements, and the Kth element among these elements is also the Kth element of the input problem to CUT2.

Proof. By Lemma 2 in [3] ♦

### 2.3 Filtration phase

The filtration stage is the last reduction technique used by our algorithm. It comprises of a sequence of iterations; each iteration reduces the number of elements under consideration by a constant factor until the number of candidate elements is less than or equal to a desired number, $N_f$. The choice of $N_f$ depends on the type of network under consideration. Let <N, K, N[1], ..., N[d]> be the problem configuration. If this configuration is the result of the CUT phase, then $N \leq 4.5K$.

The chosen filtration procedure is the technique developed in [5]. It should be noted that the distributed translation of the serial procedure REDUCE in [3] could also be used as a filtration technique; the choice here of the former technique rests on the fact that it guarantees a larger fraction of elements to be removed at each iteration. A full description and analysis of this technique can be found in [5]; an informal description is provided below.

```
PROCEDURE FILTER
(Iterate until either the sought element is found or $N \leq N_f$)

    1.  Among the medians of the elements still
        under consideration at each site, find the
        (N/2)-th weighted element x using
        weighted selection.

    2.  Determine the overall rank r of x. If r= K
        then terminate (x is the sought element).
        Otherwise, eliminate from consideration
        the elements not greater than x (if r<K) or
        not smaller than x (if r>K), and adjust K
        (if r<K) and N. If $N \leq N_f$, terminate,
        otherwise proceed with the next iteration.
END FILTER
```

**Property 7** The number of iterations required by procedure FILTER to reduce N elements to $N_f$ elements is $2.41\log(N/N_f)$.

Proof. Since at least 1/4 of the elements are removed from consideration after each iteration (see [5]), the property follows. ♦

### 2.4 Termination phase

Once the filtration phase is concluded, if the sought element has not yet been found, the $N_f$ elements still under consideration can be collected at a site where the final selection is performed.

To perform this phase efficiently, the actual value $N_f$ (a parameter in the FILTER procedure) will depend on the type of network under consideration. As shown later, the choice for shout-echo is log $\kappa$ (where $\kappa$ is the value determined after the set-up phase), and for point-to-point networks the choice is d, where d is the original number of sites.

## 3. Shout-echo networks

In shout-echo networks, each site can broadcast a message and reply to a received message; the process of a site broadcasting a message (the "shout") and receiving a reply from all other sites (the "echo") constitutes a primitive communication activity, called a shout-echo. Marberg and Gafni presented a selection algorithm (procedure FILTER in this paper); they also assume that a preliminary reduction (equivalent to the first iteration of the set-up phase followed by the termination stage) has already been performed (i.e., $N \leq \Delta^2$). Their algorithm, in the worst case, requires $4.82\log\Delta^2 + \log \Delta = 10.64 \log \Delta$ shout-echos [5].

An improvement in this bound for selection in shout-echo networks can be obtained by using the selection procedure described in the previous section. All phases can be easily implemented using shout-echo primitives.The procedure can actually be simplified in this implementation; in fact, it is not necessary to perform the specific element elimination steps of CUT1 (steps 2.2 - 2.4), in the shout-echo model: by taking $\xi^j = S$, for all j, CUT2 may be performed immediately. However, an increase of local storage space for file elements from O(d) to O(d log d) is needed at the initiating site, where d is the number of sites still under consideration when CUT is executed. By choosing $N_f = \log\kappa$ in the FILTER procedure, the following result yields.

**Theorem 3** The number of shout-echoes used in the selection algorithm is at most $6.82\log\kappa + O(1)$, where $\kappa \leq \Delta$ is the problem size obtained after the set-up phase.

Proof. The initialization and termination stages, as well as each iteration of procedure SET-UP can be implemented using only a constant number of shout-echoes. Since the procedure SET-UP can be performed in a constant number of iterations (by Theorem 1), SET-UP requres O(1) shout-echoes. After this procedure is executed, the size of the problem is $\kappa \le K$; the number of sites still under consideration is $\delta \le \kappa$; and the maximum number of elements at any one site is $n \le \kappa$.

During each iteration of CUT1, the collection of the elements $V^j$, $1 \le j \le q$, and the broadcast of the median can be done with one shout-echo. Since the number of iterations in CUT1 is bounded by $\mathrm{Min}\{\lceil \log\delta \rceil, \lceil \log n \rceil\} \le \log\kappa$, the total number of shout-echoes needed to perform CUT1 is at most $\log\kappa + O(1)$. Since the only communication activities needed in CUT2 are the collection of the elements $V^0$ and the broadcast of $x^*$, only a constant number of shout-echoes are needed to perform this procedure. Therefore at most $\log\kappa + O(1)$ shout-echoes are needed to perform the procedure CUT. By Property 6, the CUT procedure allows at most $4.5\kappa$ elements to remain. By Property 7, the number of iterations used by the FILTER procedure is at most $2.41\log\kappa + O(1)$. Since each iteration uses 2 shout echoes, the number of shout echoes used by the FILTER procedure is $4.82 \log\kappa$. Since CUT takes at most $\log\kappa + O(1)$ shout echoes, and transferring the remaining $\log\kappa$ elements will take no more than $\log\kappa$ shout echoes, therefore the number of shout echoes used in the selection algorithm is at most $6.82 \log\kappa + O(1)$. ♦

## 4. Point-to-point networks

In sec. 4.1, a point-to-point selection algorithm for the star graph shall be presented; this algorithm can obviously be employed in any other network having a spanning star graph (e.g., complete networks). In sec. 4.2, the application of the SET-UP procedure to the existing algorithms for rings, meshes, and (complete binary) trees shall then be examined.

### 4.1 Star networks

In point-to-point networks, the communication primitive is the transmission of a message to a neighbouring site. For networks whose topology is a star (or contains a spanning star subgraph), the shout-echo algorithm by Marberg and Gafni [5] can be transformed so to employ $19.28p \log \Delta - 9.64p \log d + O(d)$ messages, where $p = \mathrm{Min}\{K,d\}$. An improvement in this bound is obtained by executing all four phases (as for shout-echo networks) and choosing $N_f = d$ in the filtration phase, where d is the original number of sites.

**Theorem 4** The number of messages used in the selection algorithm in a star network is at most $9.64\ \delta \log(\kappa/d) + O(d)$ where $\kappa \le \Delta$ is the rank of the desired element and $\delta \le p$ is the number of remaining sites, respectively, in the problem configuration produced from the execution of the Procedure SET-UP.

Proof. Since the procedure SET-UP can be performed in a constant number of iterations (by Theorem 1), SET-UP can be performed in O(d) messages. During each iteration of CUT1, the collection of the elements $V^j$, $1 \le j \le q$, and the broadcast of the median can be done with $2|\xi^{j-1}|$ messages. Since $|\xi^j| < |\xi^{j-1}|/2$ by step 2.4 and $|\xi^0| = \delta \le d$ by step 1, CUT1 uses at most $O(\delta)$ messages. Furthermore, since the only communication activities in CUT2 are the collection of the elements of set $V^0$ and the broadcast of $x^*$, at most $O(\delta)$ messages are needed for procedure CUT.

By Property 6, the CUT procedure allows at most $4.5\kappa$ elements to remain. Therefore by Property 7, the number of iterations used by the FILTER procedure is at most $2.41\log(\kappa/d) + O(1)$. Since each iteration of FILTER uses $4\delta$ messages, the number of messages used by the FILTER procedure is bounded by $9.64\delta\log(\kappa/d)$. As the transmission of the remaining d file elements in the termination phase requires at most O(d) messages, the theorem follows. ♦

### 4.2 Other topologies

The existing algorithms for selection in other networks [2,6] can be made more efficient if the SET-UP procedure is used prior to their execution.

The initialization and each iteration step of procedure SET-UP can be easily implemented in rings, meshes and complete binary trees using O(d) messages; the termination step, however, should be implemented using the existing topology-dependent algorithms for selection when there is only one element at each site (e.g., [2]).

This implementation of procedure SET-UP will thus require $O(d\ (\log d)^3)$ messages in the *ring* topology to reduce the number of elements, N, to at most $\kappa\delta \le \kappa^2$. If the existing ring selection algorithm is then applied to the remaining elements, then at most $O(d(\log d)^2\log\kappa)$ messages are needed. The resulting $O(d\ (\log d)^2\log(\mathrm{Max}\{\kappa,d\})$ improving the existing bound of $O(d(\log d)^2\log N)$.

Similarly in the *mesh*, SET-UP will take O(d(logd)$^{3/2}$) messages to execute, thus enabling the existing algorithm to terminate in O(d(logd)$^{1/2}$logκ) messages; the resulting O(d (log d)$^{1/2}$log(Max{κ,d}) improves on the existing O(d(logd)$^{1/2}$logN) bound.

In the *complete binary tree*, SET-UP will require O(d) messages to reduce N to at most κδ, whereupon the exsting algorithm is called to complete the selection in O(dlogκ).The resulting O(dlogκ) bound improves on the existing O(d log(2N/d)) bound whenever κ<N/d.

## 5. Conclusions

The problem of selecting the K-th smallest element of a set of N elements distributed among d sites of a communication network has been examined. A collection of distributed reduction techniques has been presented; the combined use of these algorithms has been shown to yield new solutions for the selection problem for both point-to-point and shout-echo networks. The complexity of these solutions has been analysed and it has been shown to represent an improvement on the existing bounds.

There are still many open problems. For example, an efficient algorithm for arbitrary tree networks has not yet been developed. Another important open problem is to determine a lower bound for the distributed selection problem in point-to-point networks; the only existing lower bound is for complete binary trees [2].

**Acknowledgement**

This work has been supported in part by the National Sciences and Engineering Research Council (of Canada) under grant A2415.

## References

[1] F. Chin, H.F. Ting, "A near-optimal algorithm for finding the median distributively", Proc. 5th IEEE Conf. on Distributed Computing Systems, Denver, (May, 1985).

[2] G.N. Frederickson, "Tradeoffs for selection in distributed networks", Proc. 2nd ACM Symp. on Principles of Distributed Computing, Montreal, (Aug., 1983), pp. 154-160.

[3] G.N. Frederickson and D.B.Johnson, "The complexity of selection and ranking in X+Y and matrices with sorted columns", J. Computer and System Science 24, 2, (April, 1982), 197-208.

[4] D.B. Johnson and T. Mizoguchi, "Selecting the Kth element in X + Y and $X_1 + X_2 + ... + X_m$", SIAM J. Comput. 7 (1978), pp 556-570

[5] J.M. Marberg, E. Gafni, "An optimal shout-echo algorithm for selection in distributed sets", Proc. 23rd Allerton Conf. on Communication, Control and Computing, Monticello, (Oct., 1985).

[6] T.A. Matsushita, Distributed algorithms for selection, Master Thesis, Department of Computer Science, University of Illinois, Urbana, (July, 1983).

[7] M. Rodeh, "Finding the median distributively", J. Computer and System Science 24, 2, (April, 1982), pp. 162-167.

[8] D. Rotem, N. Santoro, J.B. Sidney, "A shout-echo algorithm for finding the median of a distributed set", Proc. 14th SE Conf. on Combinatorics, Graph Theory and Computing, Boca Raton, (Feb., 1983), pp. 311-318.

[9] D. Rotem, N. Santoro, J.B. Sidney, "Distributed sorting", IEEE Transactions on Computers C-34, 4, (April, 1985), pp. 372-376.

[10] D. Rotem, N. Santoro, J.B. Sidney, "Shout-echo selection in distributed files", Networks, to appear.

[11] N. Santoro, M. Scheutzow, J.B. Sidney, "New bounds on the communication complexity of distributed selection", J. Parallel and Distributed Computing, to appear.

[12] N. Santoro, J.B. Sidney, "Order statistics on distributed sets", Proc. 20th Allerton Conf. on Communication, Control and Computing, Monicello, (Oct., 1982), pp. 251-256.

[13] N. Santoro, J.B. Sidney, Communication bounds for distributed selection SCS-TR-10, Carleton University, Ottawa, (Sept., 1982).

[14] N. Santoro, J.B. Sidney, S.J. Sidney, On the expected communication complexity of distributed selection, SCS-TR-69, Carleton University, Ottawa, (Feb., 1985).

[15] L. Shrira, N. Francez, M. Rodeh, "Distributed k-selection: from a sequential to a distributed algorithm", Proc. 2nd ACM Symp. on Principles of Distributed Computing, Montreal, (Aug., 1983), pp. 143 - 153.

# Optimal Parallel Algorithms for Constructing a Balanced m-way Search Tree

Eliezer Dekel and Shietung Peng
University of Texas at Dallas
and
S.Sitharama Iyengar
Lousiana State University

Abstract : We present parallel algorithms for constructing balanced m-way search trees. These parallel algorithms have time complexity O(1) for an n processors configuration.

## 1. INTRODUCTION

The idea of using tree structure to represent symbol tables, dictionaries or directories has been extensively studied[K]. In all these structures we have a collection of records that are to be manipulated with regard to a certain key field in the record. Common operations on these structures are SEARCH, INSERT and DELETE. The tree structure supports efficient INSERT, SEARCH and DELETE operations. In some implementations the operations are designed so that a balanced tree is maintained through the process. Another approach is to periodically rebalance the tree. In our discussion we refer to these structures as dictionaries. The operations INSERT,SEARCH and DELETE will be referred to as basic dictionary operations.

Many types of balanced m-way search trees are reported in the literature [K,HS]. In our discussion we refer to the following:

Def 1.1 : An m-way search tree, T, is a tree in which all internal nodes are of degree $\le$ m. If T is empty then T is an m-way search tree. When T is not empty it has the following properties: (1) T is a node of type $A_0, (K_1,A_1), (K_2,A_2), \ldots,$ $(K_{m-1},A_{m-1})$ where the $A_i$, $0 \le i < m$ are pointers to the subtree of T and the $K_i$, $1 \le i < m$ are key values. (2) $K_i < K_{i+1}$, $1 \le i < m-1$ (3) All key values in subtree $A_i$ are less than value $K_{i+1}$ $0 \le i < m-1$ (4) All key values in the subtree $A_{m-1}$ are greater than $K_{m-1}$. (5) The subtrees $A_i, 0 \le i \le m-1$ are also m-way search trees.

Def 1.2 : A balanced m-way search tree is an m-way search tree with minimal height.

While many parallel architectures have been proposed and studied ,we deal directly with only the MIMD model. We assume there is a large common memory that is shared by all processors. Any processor can access any word in common memory, but access the same memory word simultaneously is not allowed.

Recently, Moitra and Iyengar[MI] explored a technique of transforming a sequential algorithm for balancing a binary search tree into an efficient parallel algorithm. Furthermore, they have shown that the resulting parallel algorithm has a time complexity O(1) when a tree with N elements is balanced with N PEs. An O(log N) time set up overhead is incurred when a new N is considered. In this paper, we generalized the results of [MI] to a general m-way search tree. We also improve the computation such that no setup overhead is required.

This paper is organized as follows. In section 2, we review some properties of m-way search trees. In section 3, we develop the general m-way search tree rebalancing/construction algorithm. In section 4, we discuss the implementation of these algorithms on an MIMD machine.

## 2. M-WAY SEARCH TREES

Dictionary searches are more efficient when they are done on a balanced tree. In order to keep the tree balanced, the insertion and deletion algorithms are designed to leave the tree balanced. Because of this requirement it is more convenient to consider balanced m-way search trees that are not necessarily of minimal height (e.g. B -trees). These trees lend themselves to easier splitting and combining. While this is true in the serial case we show that in the parallel case the complete m-way tree proves to be an efficient choice.

Def 2.1 : A level labeling of an m-way search tree is a labeling in which nodes are numbered serially, top down left to right. The root node is always labeled 1.

Lemma 2.1 : When the nodes of a complete m-way tree are labeled by level labeling then the m children of node i (if they exist) are labeled $(i-1)*m + 2 + j$ $,0 \le j \le m-1$.

Def 2.2 : In an m-way tree T, the (i,j)'th node refers to the jth node from the left on the ith level, if it exists. We refer to this indexing method for m-way search trees as two dimensional indexing.

Def 2.3 : Inorder traversal of an m-way search tree is defined by the following recursive procedure:

```
procedure MINORDER (T)
{* T is an m-way tree as defined in def.1.1 *}
if T <> nil
then  call MINORDER ($A_0$);
      for I= 1 to M
```

0190-3918/86/0000/1010 $01.00 © 1986 IEEE

```
        begin
        if no $K_i$ key then exit;

        visit($K_i$);

        MINORDER ($A_i$)
        end;
end. {* MINORDER *}
```

Def 2.4 : An index can be associated with each key in an m-way search tree. If this index corresponds to the order in which MINORDER visits the keys we call the indexing, Inorder indexing.

Lemma 2.2 Suppose node I is at level r of a full m-way search tree of height n. Let the inorder index associated with key $K_i$ in I be $X_i$. Then $X_{i+1}-X_i=m^{n-r}$ for $1 \leq i < m-1$.

Lemma 2.3 a) Let nodes I and J be two adjacent sibling nodes at level r of a full m-way search tree of height n. Let $I.X_{m-1}$ be the inorder index of the last key in node I and $J.X_1$ the inorder index of the first key in node J. Then

$$J.X_1 - I.X_{m-1} = 2 * m^{(n-r)}.$$

b) The above claim is true for any two adjacent nodes at level r.

Theorem 2.1 Consider a full m-way search tree of height n. The inorder indexes for keys in node i at level r (node (r,i)) are: $(i-1)*m^{(n-r+1)}+j*m^{(n-r)}$, where $1 \leq i \leq m^{(r-1)}$; $1 \leq j < m$

Corollary 2.1 A key with inorder index t is at level r of a full m-way search tree if and only if $t \bmod m^{(n-r+1)} \neq 0$ and $t \bmod m^{(n-r)} = 0$.

Corollary 2.2 Let t be the inorder index associated with a key at level r of an m-way tree of height n and let $q = \lfloor t/m^{(n-r+1)} \rfloor$. Keys with the same r value (Cor.2.1) and q value are in the same node, in the m-way search tree. Moreover the two dimensional indexing of this node will be (r,q+1).

Corollary 2.3 Let t be the inorder index associated with a key at level r of an m-way tree of height n. The position of the key within the node is given by s,

$$s = \lfloor (t \bmod m^{(n-r+1)}) / m^{(n-r)} \rfloor,$$

$1 \leq s \leq m-1$.

Lemma 2.4 Let n be highest level in a complete m-way tree. The inorder indexes associated with keys in node (n,p) are $(p-1)*m + j$, where $1 \leq j < u$, (n,v) is the last node in this level, $v \leq m^{(n-1)}$. $u = m-1$ for all nodes except (n,v).

## 3. ALGORITHMS FOR REBALANCING AN M-WAY SEARCH TREE.

We associate a PE with each node in the m-way balanced search tree.Let each PE calculate the inorder indexes for keys that should reside in the node. We consider a full m-way search tree and assume that each PE knows the height and degree of the tree.

ALGORITHM 1

(* Assume that there are $(m^h-1)/(m-1)$ PEs. The number of keys that are to be associated with this tree are $m^h-1$.*)

(I): (* each PE computes the two dimensional index of the node it is associated with, i.e. there is a mapping from i, the PE index, to (q,r),$1\leq i \leq \lceil n/(m-1) \rceil$, $1 \leq q \leq m^{(h-1)}$, $1 \leq r \leq h$ *)

```
 for each PE i do
 begin
 j ← $\lfloor \log_m i \rfloor$ (* i is the PE index*)

 if i > $(m^{(j+1)}-1)/(m-1)$
 then  r ← j+2;
       q ← i-$(m^{(j+1)}-1)/(m-1)$;
 else  r ← j+1;
       q ← i-$(m^j-1)/(m-1)$;
 end.
```

(II): (* Each PE represents a node (r,q). Node (r,q) has m-1 inorder indexes associated with it. Each inorder index is associated with a unique key. The inorder index for key $K_s$ in node (r,q) is held in $X_s$, $1 \leq s \leq m-1$. *)

```
 for each node (r,q) do
    for  s ← 1 to m-1 do
    $X_s$ ← $((q-1)*m+s) * m^{(h-r)}$;
```

(III): (* The m pointers for node (r,q) are stored at $A_s$, $0 \leq s \leq m-1$ *)

```
 for each node (r,q) do
    for  s ← 0 to m-1 do
    if r < h then
          $A_s$ ← node ((q-1)*m+s+1,r+1)

          else
          $A_s$ ← null
```

Theorem 3.1 Algorithm 1 correctly constructs the required full m-way search tree within O(1) time if O(n) PEs are available.

We now consider the general case where the number of keys can be any positive integer. Algorithm 2 will produce a complete m-way search tree for the given number of keys. Only the last node in the tree (greatest level label) can have less then m-1 keys associated with it. We assume that each PE knows the number of keys and the degree of the search tree. These values might be passed to the PE as procedure parameters.

### ALGORITHM 2

```
(* The input keys have inorder indexes in
the range [1 : n], n can be any positive
integer *)
(I): Compute the two dimensional index of
each node as in (I) of algorithm 1.
(II): (* Compute the parameters,u,v and
w, of the tree. They stand for the
number of keys in the last node, the
number of the full nodes in the last
level and the inorder index of the right
most key in the last level. *)
 for each PE i do
 begin
 $h \leftarrow \lfloor \log_m n \rfloor$ ;
 if $n = m^{(h+1)}-1$ then (* full tree *)
 execute II and III of algorithm 1 and
 stop ;
 $c \leftarrow n-(m^h-1)$ ;
 $u \leftarrow c \bmod (m-1)$ ;
 $v \leftarrow \lfloor c/(m-1) \rfloor$ ;
 if $u = 0$ then $w \leftarrow v*m -1$ else $w \leftarrow$
 $v*m+u$ ;
 end.
(III): (* Compute the inorder indexes
that are directly influenced by the last
level. These will be associated with
nodes in the left part of the tree *)
 for each node (q,r) do
 begin
 $s \leftarrow 1$ ;
 loop
    $temp \leftarrow ((q-1)*m+s) * m^{(h-r+1)}$ ;
    if $temp > w$ or $s = m$ then exit ;
    $X_s \leftarrow temp$ ;
    $s \leftarrow s+1$;
 forever
 end
(IV): (* Compute the inorder indexes for
the rest of the tree *)
 for each node (q,r) with $r < h+1$ do
 begin
 $s \leftarrow m-1$ ;
 loop
    $temp \leftarrow ((q-1)*m+s) * m^{(h-r)} + c$ ;
    if $temp \le w$ or $s = 0$ then exit ;
    $X_s \leftarrow temp$ ;
    $s \leftarrow s-1$ ;
 forever
 end.
(V): (* Compute pointer values*)
 for each node (q,r) do
 begin
 if $u = 0$ then $q_1 \leftarrow v$ else $q_1 \leftarrow v+1$ ;
 (* node $(q_1,h+1)$ contains w *)
 $q_2 \leftarrow \lfloor (q_1-1)/m \rfloor + 1$;
 (* $(q_2,h)$ is the parent of $(q_1,h+1)$*)
 $j \leftarrow (q_1-1) \bmod m$ ;
 (* $A_j$ of $(q_2,h)$ points to $(q_1,h+1)$ *)
 for $s \leftarrow 0$ to $m-1$ do
 if $r < h$ or ($r = h$ and $q < q_2$)
 or ($r = h$ and $q = q_2$ and $i \le j$)
 then $A_s \leftarrow$ node $((q-1)*m+s+1,r+1)$
 else $A_s \leftarrow$ null ;
 end.
```

Lemma 3.1 The following are true for a complete m-way search tree : (i) The right most node of level $h+1$ is node$(v,h+1)$ when $u = 0$ and node$(v+1,h+1)$ otherwise. (ii) The last element of the right most node at level $h+1$ is w.

Theorem 3.2 Algorithm 2 correctly constructs the required complete m-way search tree within O(1) time if O(n) PEs are available.

## 4. IMPLEMENTATION AND CONCLUSIONS

In section 3, we have presented an optimal parallel algorithms for rebalancing or constructing balanced m-way search trees. If n keys are to be associated with the tree then the construction can be carried out in O(1) time using O(n) PEs. A more detailed discussion can be found in [DPI]. When the dictionary is stored in external memory. The storage structure is chosen so that the number of I/O operations are minimized. An m- way search tree is a popular choice. m is selected to fit the physical characteristics of the external storage [HS]

The above observations can be translated quite effectively to practice in our MIMD environment. The system can initiate any number of searches in a "pipelined" fashion. Each search is conducted using only one PE. leaving one machine cycle between consecutive requests. Search results can be obtained in a pipeline interval of O(1). While some PEs are conducting searches other PEs are free to perform other tasks.

Our solution is applicable for a general purpose machine environment. The m-way search tree is kept in external storage. At any point of time k PEs are available, $0 \le k \le P$ (P is the maximal number of PEs available on the a machine.). In such a machine the operating system can be instructed to allocate only one processor for a search operation and as many PEs as available or required ( whichever is the minimum), in case a new tree has to be constructed or an existing tree rebalanced.

## BIBLIOGRAPHY

[DPI] E. Dekel, S. Peng and S. Iyengar, " Optimal parallel algorithms for constructing and maintaining a balanced m -way search tree," Technical Report CS-209, University of Texas at Dallas,1985.

[HS] E. Horowitz and S. Sahni, "Fundamentals of computer algorithms," Computer Science Press, Inc, 1984.

[K] D.E. Knuth, Art of Computer Programming, Vol. 1 : Fundamental algorithms, Addison-Wesley, Reading, Ma., 2nd Ed. 1973.

[MI] A. Moitra and S.S. Iyengar, "A maximally parallel balancing algorithm for obtaining complete balanced binary trees," IEEE Trans. on Comput. , Vol. C-34, No. 6 pp.563-565, June 1985.

# A STREAM-ORIENTED PARALLEL PROCESSING SCHEME FOR RELATIONAL DATABASE OPERATIONS

**Yasushi Kiyoki**
Institute of Information Sciences and Electronics
University of Tsukuba
Sakura, Niihari, Ibaraki 305, Japan

**Ryuzo Hasegawa and Makoto Amamiya**
NTT Electrical Communication Laboratories
Musashino, Tokyo 180, Japan

## ABSTRACT

This paper presents a stream-oriented parallel processing scheme for relational database operations (relational operations). By combining the stream processing principle with demand-driven control, this scheme exploits the pipeline concurrency inherent in a query and makes it possible to execute relational operations with limited resources. This paper presents a basic algorithm for this stream-oriented parallel processing scheme and discusses its features and effectiveness.

## 1. Introduction

The relational data model [1] proposed by E. F. Codd has received much attention as a practical data model based on a set theory. However, many problems remain to be solved in applying the model to practical applications. In particular, improvement of the performance in executing relational operations is essential in implementing a relational database system.

Most conventional parallel processing schemes for relational operations were designed to improve the execution performance of an individual relational operation, such as the join, selection or projection operations ([4],[6],[7],[8] et al.).

This paper presents a novel parallel processing scheme called the stream-oriented parallel processing scheme (stream scheme). This scheme is based on the stream processing principle [2], and exploits the parallelism inherent in a query. The stream scheme is attractive not only for the design of a dedicated database machine but also for the design of a distributed query processing system [10]. In particular, when memory and processor resources are limited, this scheme acts to the best of its ability making efficient use of those resources. The conventional schemes do not alleviate problems due to the complexity of resource management, such as memory overflow when executing relational operations. For example, processing performance decreases drastically when the amount of main memory available is less than that required to process the operand relations. By combining the stream processing principle with demand-driven control, the stream scheme enables relational operations to be executed with limited available memory and to effectively exploit the parallelism of pipeline processing.

This scheme can also be applied to query processing in a sequential machine. In such a case, relational operations are described as concurrent processes, and each relational operation for query processing is executed in quasi-parallel.

In the stream scheme, since the scheme to execute an individual relational operation is not restricted, efficient conventional schemes can be employed as the internal processing scheme for a relational operation.

The main features of the stream scheme are as follows:

(1) Intermediate data does not cause memory overflow even in query processing in which the join operation, Cartesian-product operation or union operation creates a large intermediate relation.

(2) Concurrent pipeline processing is performed between relational operations in a query. The scheme exploits parallelism by adapting to existing system resources.

(3) This scheme enables a relational operation to be activated when a subset (several tuples) of each operand relation is ready to be processed. As a result, the response time required to obtain the earlier parts of resulting tuples of a query can be shorten.

## 2. An Algorithm for Relational Operations

A query can be thought of as a tree whose nodes represent a set of relational operations. The leaves of the tree only reference source relations of the database. In the stream scheme, a query is executed as shown in Fig.1. We define the consuming node of intermediate tuples produced by a node as the "upper node"(upper relational operation node), and the producing node of intermediate tuples as the "lower node"(lower relational operation node). Each node has one or two input buffers. When a relational operation node receives a demand from the upper node, it accesses a page in its input buffer and then executes the relational operation until it completes the production of one resulting page of tuples in the output buffer. The output buffer is then treated as the input buffer for the upper node. Here, "access a page" means that a node is ready to perform a relational operation on a page in its input buffer.

0190-3918/86/0000/1013 $01.00 © 1986 IEEE

Individual relational operation nodes do not create a whole intermediate relation for a single demand. Each one creates only one page of tuples for a single demand. Individual buffers do not require the capacity to store an entire intermediate relation. The size of each page is set to half of the corresponding buffer size, that is, each buffer has two storage areas so that two pages can be stored. As the sizes of buffers assigned to relational operation nodes are different, the page sizes are also different. Immediately after a node begins to access a page in one of two areas of its input buffer, it sends a demand to the lower node to refill the other area of the input buffer with the subsequent page. In this algorithm, it is assumed that the double buffering mechanism is supported in every buffer. That is, while the node accesses a page in one area of its input buffer to execute a relational operation, the lower node can store a page in the other area of the same buffer at the same time.

As a result, concurrent pipeline processing is performed between relational operation nodes. By using demand-driven control, unary relational operations (the selection, restriction and projection operations), and binary relational operations (the join, union, intersection, difference and Cartesian-product operations) can be concurrently executed with limited memory resources. In particular, this algorithm shows attractive advantages in executing the join, union and Cartesian-product operations, which are the most time- and resource-consuming operations.

## 2.1 Unary Relational Operations

A unary relational operation node has one input buffer and one output buffer. Unary relational operations are executed by the following algorithm.

Step (1) When the unary relational operation node receives a demand from the upper node, it repeatedly executes Steps (2) and (3) until one page made up of the resulting tuples is created and stored in the output buffer.

Step (2) A single page is accessed in one area of the input buffer. At this time, the other area of this input buffer becomes available and a demand is pre-issued to the lower node to refill the other area of that buffer with the subsequent page. As a result, pipeline concurrent processing is implemented between this unary relational operation node and the lower node.

Step (3) The relational operation is executed on the page that has just been accessed in Step (2), and the resulting tuples are stored in the output buffer. Once a page in the input buffer is manipulated in this step, the page is deleted from the buffer. If the output buffer is filled with a page of resulting tuples, the execution is suspended at this point and the node waits for the next demand from the upper node. Otherwise, Steps (2) and (3) are executed repeatedly. If the page being manipulated is the last one of the operand relation, the execution of this relational operation is terminated.

For the projection operation, if the primary key attribute is not included as an operand attribute, it is necessary to eliminate duplicate tuples. Therefore, the resulting tuples in the output buffer of the projection node must not be deleted even after the upper node have completed manipulating those tuples. That is, the projection node must reserve the output buffer area to provide storage for all of the tuples resulting from the projection operation.

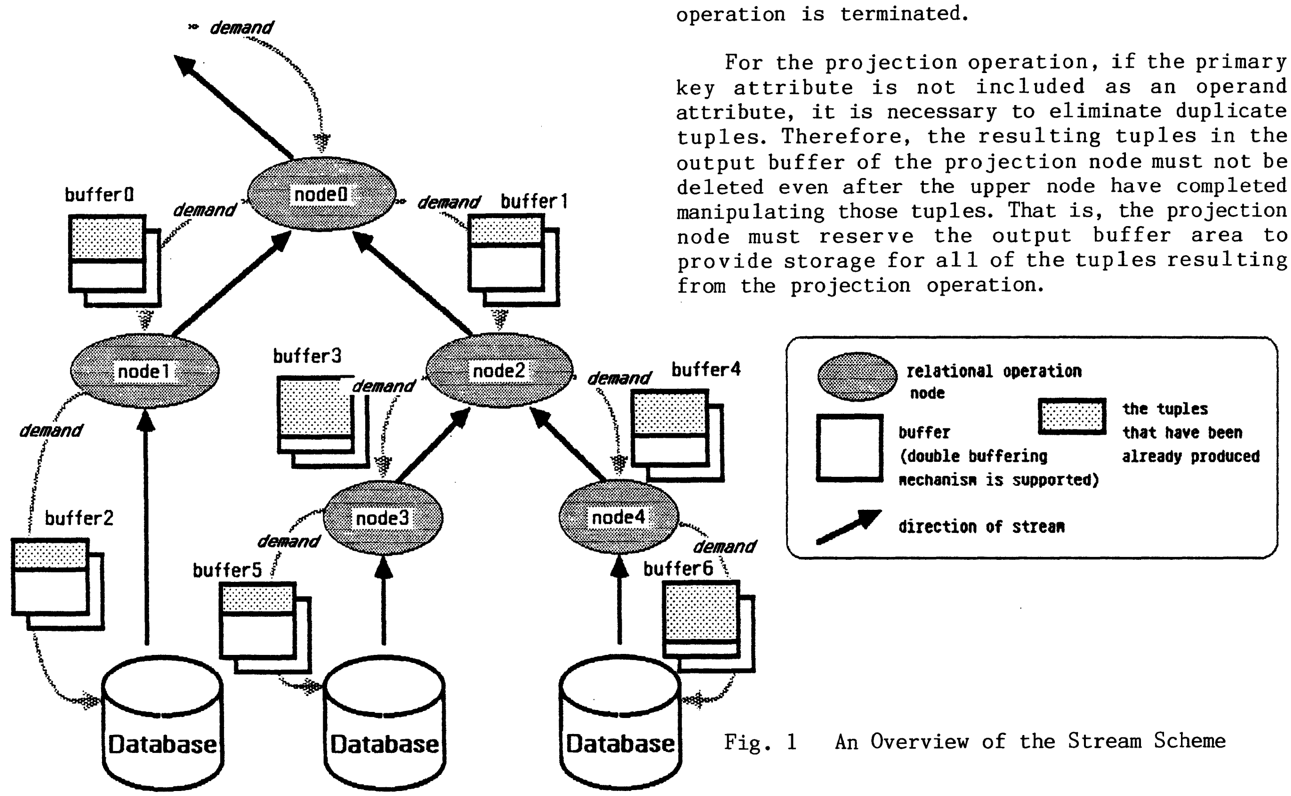


Fig. 1 An Overview of the Stream Scheme

### 2.2 Binary Relational Operations

In binary relational operation nodes, the operation is carried out by comparing each page of the outer-relation with all of the pages of the inner-relation. That is, the nested-loop algorithm is applied to the page-level granularity. The comparison algorithm applied to the tuple-level granularity within a page is not restricted.

The sort-merge, nested-loop or hashing algorithms can be used to compare the tuples of two different pages. Even dedicated multiprocessors can also be utilized to compare two operand pages. Each binary relational operation node has two input buffers and one output buffer. One of the input buffers is used for storing outer-relation pages and the other for storing inner-relation pages. Binary relational operations are executed by the following algorithm.

Step (1) When the binary relational operation node receives a demand from the upper node, it executes Steps (2), (3) and (4).

Step (2) One outer-relation page is accessed in one of two areas of the input-buffer with the subsequent outer-relation page. Then, a demand is pre-issued to the lower node which produces outer-relation pages to refill the other area of that buffer. By issuing the demand in advance before executing the relational operation, the production of the following operand page in the lower node overlaps with the execution in this relational operation node.

Step (3) One inner-relation page is accessed in one of two areas of the input-buffer. Then, a demand is pre-issued to the lower node to refill the other area of the input-buffer.

Step (4) The relational operation is executed by comparing the outer-relation page accessed in (2) with the inner-relation page accessed in (3). The resulting tuples are stored in the output buffer. When one of two areas of the output-buffer is filled as the result of the demand received in (1), the execution is suspended until the next demand is issued from the upper node. Otherwise, (3) and (4) are executed repeatedly. If the inner-relation page being compared with the outer-relation page is the last page and the outer-relation page is not the last one, then the lower node which creates inner-relation pages is initialized and control returns to (2). (This means the reproduction of the inner-relation by the lower node. The inner-relation is reproduced in order to compare all inner-relation pages with each outer-relation page. If all of the outer-relation pages are stored in the input buffer, this reproduction is not required.) If both of the pages being compared with each other are last pages, the execution of the relational operation is terminated.

## 3. Design Considerations

### 3.1 Stream Processing

This section discusses the basic concepts and properties of the stream scheme. In Fig.1, the relational operation nodes (node-1, node-3 and node-4) for a query can be executed in parallel because these relational operations are independent. Furthermore, the concurrency among relational operation nodes which are vertically connected can also be exploited in the stream scheme.

A stream is an ordered sequence of data which are arranged according to the order of production. In this scheme, each element of a stream corresponds to a tuple or a page in a relation. Therefore, the ordered sequence of tuples or pages is manipulated as a stream. In processing relational operations, it is not necessary to wait for the production of complete operand relations. Processing of relational operations can start immediately after an earlier part of the stream is constructed.

Criteria for exploiting parallelism are given here. We will consider a subquery consisting of two join operation nodes as shown in Fig.2. It is assumed that the two operations are executed in different processors and each operation is executed by using the nested-loop algorithm for tuple-level granularity in comparing two operand pages. For the sake of simplicity, it is also assumed that communication overhead between processors does not exist. This is a reasonable assumption because execution process of a relational operation can overlap the communication process. Furthermore, the execution time is much longer than communication time when a single processor is used in each node. If a multiprocessor is allocated to each node, the communication overhead may affect the parallelism of stream processing.

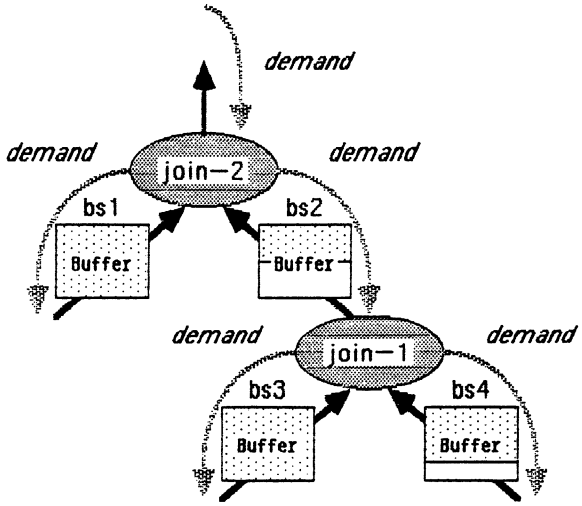


Fig. 2 Pipeline Concurrency in a Subquery

If the join-1 node completes producing the next operand page while join-2 node executes the comparison of two operand pages currently stored in the input buffers, pipeline delay will be eliminated. That is, in the join-2 node, the suspension of the relational operation execution which occurs due to the absence of the next inner-relation page causes pipeline delay. The criterion is given by the following formula. "jsf1" represents the join selectivity factor of the join-1 node. The number of tuples of an intermediate relation is jsf1*(the number of tuples in the outer-relation)*(the number of tuples in the inner-relation). "bs1" and "bs2" are the sizes (the number of tuples) of buffers to store an outer-relation page and an inner-relation page in the join-2 node.

1) The number of times the tuples in an inner-relation page and the tuples in an outer-relation page are compared in the join-2 node is

$$bs1*bs2. \qquad (1)$$

2) The number of times of comparisons in the join-1 node to produce an inner-relation page for the join-2 node using the nested-loop (1/jsf1 is the average number of times of comaparisons to generate one tuple by the nested-loop) is

$$bs2/jsf1. \qquad (2)$$

3) The condition to continue processing in the join-2 node without suspension is

$$(1) = (2), \text{ that is,}$$
$$bs1 = 1/jsf1. \qquad (3)$$

In general, the criterion to exploit the maximum parallelism between two relational operation nodes is

buffer size for an outer-relation page
= 1/join selectivity factor of the lower node. (4)

If this criterion is satisfied between two nodes in two different processors, highly concurrent processing can be realized.

If the sort-merge algorithm or the hashing algorithm is used in the comparison between an inner-relation page and an outer-relation page, approximately the same criterion as that used for the nested-loop algorithm is given. If a multi-processor is used to implement the nested-loop algorithm in each node and the communication overhead among processors is not considered, the criterion is

$$bs1 = P2/(jsf1*P1). \qquad (5)$$

(P1 and P2 are the numbers of processors allocated to the join-1 and join-2 nodes, respectively.)

### 3.2 Demand-driven Control

The stream constructed by the pipeline processing can be controlled according to the buffer capacity by demand-driven control. The stream flowing on the pipeline is limited and the computation on the stream can be carried out in a limited resource environment. That is, buffer overflow does not occur even if the join, Cartesian-product or union operations create a large intermediate relation. However, if a whole outer-relation is not stored in the input buffer in processing a binary relational operation, the inner-relation must be reproduced as described in Subsection 2.2. In this case, a source relation, or an intermediate relation produced by applying selection and projection operations prior to the binary relation, must be re-accessed. Therefore, when the reproduction of an intermediate relation causes a heavy overhead, the input buffer for outer-relation pages should have enough capacity to store the whole outer-relation. In such a case, we may change Step (4) of the algorithm described in Subsection 2.2 as follows:

To avoid having to reproduce the inner-relation, all of the outer relation pages are produced all at once in Step (2). At this time, the outer-relation pages overflowing from the input buffer are stored in secondary storage. In Step (4), each of the outer-relation pages is compared with the inner-relation page, which was accessed in Step (3), until the output-buffer is filled. When the output buffer is full, the execution is suspended until the next demand is issued from an upper node. If the outer-relation page being compared is the last page, control returns to Step (3). If both pages being compared are last pages, the execution of the relational operation is terminated.

### 3.3 Design Approaches

Two approaches to the design of a relational database system based on the stream scheme can be considered. One approach is based on developing a distributed query processing system on conventional computers connected to a local area network. In general, in the local area network environment, the number of processors used for database processing is restricted. Also, the capacity of main memory allocated to each processor is restricted. The stream scheme is attractive in such a resource-limited environment. This approach is presented in [10].

The other approach is based on developing a dedicated machine for the stream scheme. One candidate for a dedicated machine is to design a dataflow machine with demand-driven control mechanism. It is well known that the dataflow computation is suitable for the exploitation of the parallelism inherent in a functional program. Relational operations can be described in a functional programming language providing the specifications of eager and lazy evaluation mechanisms[9]. Furthermore, the eager and lazy evaluation mechanisms are useful for realizing pipeline processing and demand-driven control[11]. Therefore, by specifying these mechanisms in describing the programs of relational operations, the stream scheme is implemented on a dataflow machine supporting eager and lazy evaluation mechanisms.

## 4. Performance Considerations

### 4.1 Effects of Stream Processing

The stream scheme was experimentally examined under the UNIX 4.2BSD on the Sun-2 workstation[14]. The experimental environment for stream processing is summarized as follows:

(1) Each relational operation node is described as a coroutine, and pipeline processing among relational operation nodes is simulated.
(2) A whole outer-relation is retained in an input buffer, that is, an inner-relation do not have to be reproduced.
(3) In the join node, the sort-merge algorithm is used to compare an inner-relation page with an outer-relation page. An inner-relation page and an outer-relation page are first sorted on joining attributes, respectively. After that, these pages are joined by using the merge operation. It is assumed that the sort-merge algorithm is implemented by a sequential process.
(4) In this environment, the situation in which the demand-driven control is not employed can also be simulated. We call this situation "non-demand-driven case". In the non-demand-driven case, though relational operation nodes are activated on the basis of the page-level granularity in the same manner as the demand-driven case, the production of pages is not controlled by the demand-driven mechanism. Therefore, although pipeline concurrency can be exploited, intermediate pages may cause buffer overflow.
(5) The communication time between processors is set to 16.6 msec/2k bytes. It is assumed that communications are performed exclusively. That is, when two processors are communicating, no other processors can communicate with each other.

The sample query shown in Fig.3 is chosen from the benchmarks described in [13]. This query consists of two join operations and two selection operations. The parameter settings of source relation sizes, join selectivity factors (jsf), selection selectivity factors (ssf) and buffer sizes are shown in Table 1. Tuples in all relations are 64 bytes long, each including two 4-byte integer attributes as joining attributes. All integer attributes have uniformly distributed values, but the range of their distributions varies in order to provide different join selectivity factors.

The experimental results are shown in Table 2. As the join selectivity factors (jsf1 and jsf2) of the join nodes(join-1, join-2) increase, the first response time (the time to obtain the first resulting page of a query) becomes shorter in the stream scheme. On the other hand, the response time (the time to obtain all of the resulting pages) becomes longer as the increase of the jsf because the number of tuples in the intermediate relation manipulated by the join-1 node becomes larger. However, in comparing the case of jsf1=jsf2=0.002 with the case of jsf1=jsf2=0.001, though the number of tuples in the intermediate relation manipulated by the join-1 node doubles, the increase of the response time is smaller. This is because join-1 and join-2 processings overlap due to pipeline effect.

In comparing the experiment-1 (buffer1 = buffer3 = buffer5 = 10(tuples)) with the experiment-2 (buffer1 = buffer3 = buffer5 = 100(tuples)), the first response time in the experiment-1 is shorter than that in the experiment-2. This is because the time to activate the join-2 and join-1 nodes can be shorten in the small buffer case. On the other hand, the response

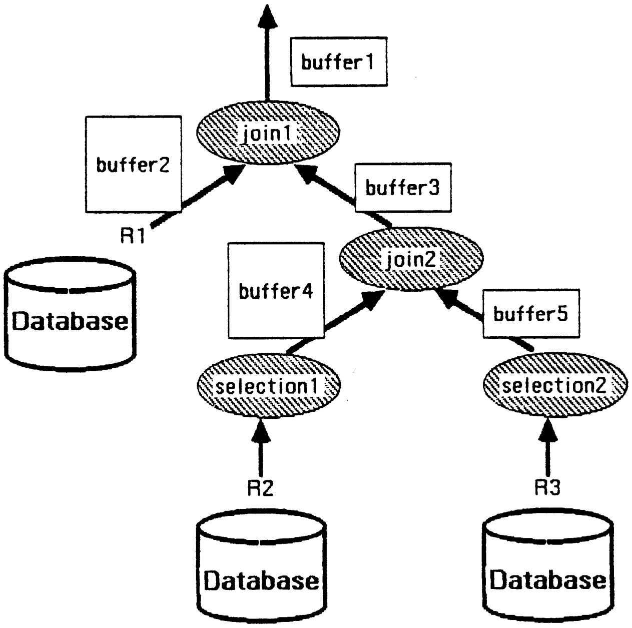


Fig. 3 Sample Query

Table 1 Parameter Settings

source relation size : R1 = 1000(tuples),
R2 = R3 = 10000(tuples)
tuple size : 64 bytes
buffer size : buffer2 = buffer4 = 1000(tuples)
buffer1 = buffer3 = buffer5 = 10(tuples)
(experiment-1)
buffer1 = buffer3 = buffer5 = 100(tuples)
(experiment-2)
selection selectivity factor(ssf) :
ssf1(selection1) =ssf2(selection2) = 0.1
((intermediate relation size) = ssf*(relation size))
join selectivity factor(jsf) : jsf1(join1) = jsf2(join2)

| jsf<br>(jsf1=jsf2) | intermediate relation size<br>(result relation size of each operation) | | | |
|---|---|---|---|---|
| | selection1 | selection2 | join2 | join1 |
| 0.0001 | 1000 | 1000 | 100 | 10 |
| 0.001 | 1000 | 1000 | 1000 | 1000 |
| 0.002 | 1000 | 1000 | 2000 | 4000 |

Table 2 Response Time

| jsf<br>(jsf1<br>=jsf2) | Experiment-1<br>(buffer-1,3,5 = 10(tuples)) | | | | Experiment-2<br>(buffer-1,3,5 = 100(tuples)) | | | |
|---|---|---|---|---|---|---|---|---|
| | Demand-driven | | Non-demand-driven | | Demand-driven | | Non-demand-driven | |
| | FR Time | R Time | FR Time | R Time | FR Time | R Time | FR Time | R Time |
| 0.0001 | 12.1 | 13.8 | 12.1 | 13.8 | 5.8 | 5.8 | 5.7 | 5.7 |
| 0.001 | 3.5 | 16.5 | 3.5 | 16.3 | 4.4 | 7.4 | 4.4 | 7.4 |
| 0.002 | 3.5 | 28.3 | 3.5 | 28.0 | 3.8 | 11.0 | 3.8 | 11.0 |

FR Time : First Response Time (sec)
R Time : Response Time (sec)

time in the experiment-1 becomes longer than that in the experimental-2. This is because the communication overhead between two nodes cause overhead in the smaller granularity case. Furthermore, since the sort-merge algorithm is used in comparing two operand pages, the total number of times of comparisons increases in the smaller granularity case.

In comapring the response time in the demand-driven case with that in the non-demand-driven case in Table 2, the response times of both cases are almost the same. This result shows that the overhead due to the transfers of demands did not affect to the response time. In general, the time to transfer a demand between processors is much shorter than the time to transfer a page. Therefore, the transfers of demands do not cause heavy overhead.

Another major interest is in comparing the memory requirement in the demand-driven case with that in the non-demand-driven case. Fig.4 shows the memory requirement for the inner-relation pages of each join node in the experiment-1, and Fig.5 shows that in the experiment-2. In the demand-driven case, a query can be executed with the fixed sizes of buffers. On the other hand, in the non-demand-driven case, the memory requirement for the buffer increases as increasing the join selectivity factor. Though the response times in both cases are approximately the same, the memory requirement in demand-driven case is much smaller than that in the non-demand-driven case. In particular, in the non-demand-driven case, when the time to produce a page in the lower node is much longer than the time to consume the page in the upper node, the input buffer of the upper node must have an enough capacity to store the large amount of the intermediate pages. Since the selection operation node (selection-2) which is the lower node producing the inner-relation pages of the join operation (join-2 node) produces pages very fast, the input-buffer(buffer-5) of the join-2 node must store many inner-relation pages. In the join-1 node, as the join-2 node produces the intermediate pages of the inner-relation of the join-1 node faster as increasing the join selectivity factor, the memory requirement becomes

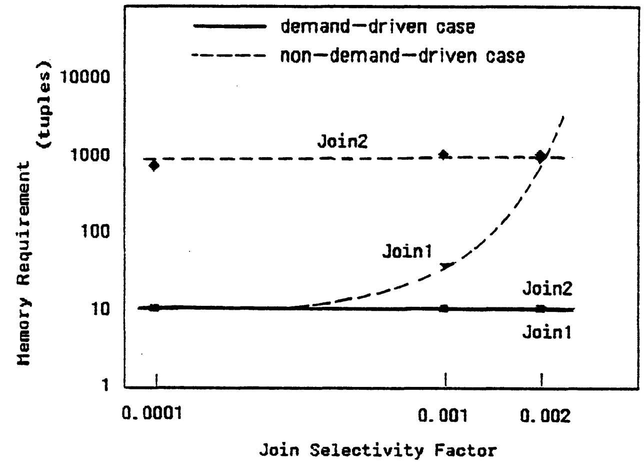


Fig. 4 Memory Requirement (Experiment-1)

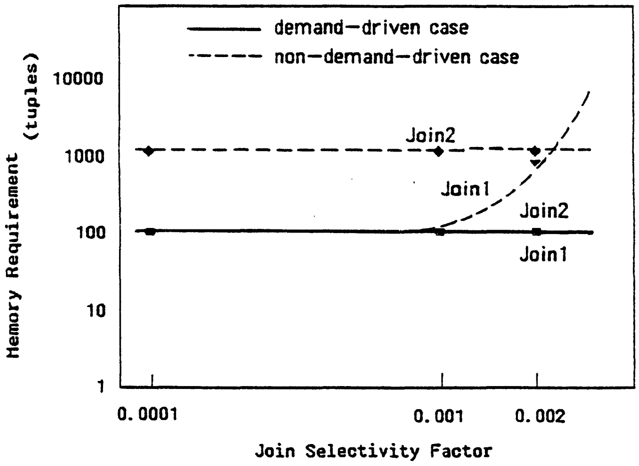


Fig. 5 Memory Requirement (Experiment-2)

larger. In those situations, the superiority of the stream scheme is distinguished.

## 4.2 Comparisons with Conventional Schemes

In this evaluation, it is assumed that each relational operation node is constructed with a multiprocessor. It is also assumed that the source relations, or intermediate relations whose sizes were reduced by applying selection or projection operations prior to the join operation are retained in the main memory. The performance of the stream scheme in executing the join operation is represented as follows: T1 and T2 are the numbers of tuples in the outer-relation and the inner-relation of the join node, respectively; Im is the number of tuples in the outer-relation of the upper relational operation node if the upper node is a binary relational operation. Otherwise, Im is set to B; B is the number of tuples that can be stored in a buffer(It is assumed that the size of each buffer is the same.); "jsf" is the join selectivity factor of the join node being evaluated; "jsf2" is the join selectivity factor of the lower join node; P1 is the number of processors allocated to the join node being evaluated; P2 is the number of processors allocated to the lower join node; P is P1+P2; Cm is the time required to broadcast a tuple to the processors; Ro is the comparison time between two tuples; Sw is the swapping time of a tuple; and Wr is the storing time of a resulting tuple.

The total number of times tuples are compared is

$$T1*T2*(Im/B).$$

The memory requirement is

$$3*B.$$

The swapping time of tuples in operand intermediate relations is

$$0.$$

The processing time of the join in the multiprocessor node is

$$Cm*B + Ro*B/(jsf2*P2) + Wr*B/P2 \\ +(T1/B)*(T2/B)*\max(Cm*B+Ro*(B*B/P1)+Wr*jsf*B*B/P1, \\ Cm*B+Ro*B/(jsf2*P2)+Wr*B/P2).$$

In most conventional schemes for relational operations, the granularity for the operand data of each relational operation is set to a relation. In this case, after the relational operation node completely produces a whole intermediate relation, then the subsequent relational operation is activated. Therefore, parallelism of pipeline processing between relational operations is not exploited, and furthermore memory swapping is necessary between main storage and secondary storage when memory overflow occurs due to intermediate relations. If the granularity of data in activating a relational operation is set to a page, the advantages of pipeline concurrency inherent in a query can be utilized[3],[12]. An execution scheme that implements this pipeline processing by using the data-driven control mechanism has been proposed[5]. However, pipeline processing using data-driven control may cause heavy overhead when executing a query.

A binary relational operation node is required to compare each outer-relation page with all of the inner-relation pages. Therefore, even if only one inner-relation page and only one outer-relation page remain to be produced, all of the other produced outer-relation pages and all of other produced inner-relation pages must be kept in storage. As a result, every node of a query may be forced to keep large amounts of intermediate data in storage.

We compare the stream scheme with a typical conventional scheme which employs the relation-level granularity when activating a relational operation. We assume this conventional scheme implements a relational operation using parallel nested-loop algorithm[4]. This algorithm is employed in many database machines. If it is employed, each tuple of the outer-relation is compared with every tuple of the inner-relation. In the conventional scheme, the total number of times tuples are compared is

$$T1*T2.$$

The memory requirement is

$$T1+T2+jsf*T1*T2.$$

The swapping time of tuples in intermediate relations is

$$Sw*((T1-B)+(T1/B)*(T2-B) + (jsf*T1*T2-B)).$$

The processing time of the join operation in a multiprocessor is

$$Cm*T1 + Ro*T1*T2/P + Wr*jsf*T1*T2/P.$$

The total number of times tuples are compared in the conventional scheme is O(T1*T2) and that in the stream scheme is O(T1*T2*Im/B). "Im/B" indicates the number of times the inner-relation is reproduced when the whole outer-relation is not retained in the input buffer. The stream scheme overcomes this overhead because of utilization of pipeline processing, and the prevention of memory overflow.

As described in Section 2, memory overflow due to intermediate relations does not occur in the stream scheme. In the conventional scheme, on the other hand, if the intermediate relation becomes larger than the buffer size, memory swapping must be done between secondary storage and the buffer. The O(T1*(T2-B)/B) number of tuple swapping times is required. The number of swapping times is almost the same as the number of times tuples are compared in the nested-loop algorithm. In general, since the tuple swapping time is much longer than the tuple comparison time, the memory overflow causes heavy overhead. That is, the superiority of the stream scheme is distinguished when memory overflow occurs in the conventional scheme.

The advantage of the stream scheme increases in proportion to the increase of the join selectivity factor. This is because the stream of tuples continuously flows through the pipeline among the relational operation nodes in the larger jsf case. On the other hand, in the conventional scheme, the size of the intermediate relation increases according to the increase of the join selectivity factor. As the result, the subsequent relational operation node must munipulate a larger intermediate relation, and the processing time of the subsequent node becomes longer. Therefore, when the join selectivity factor is large, the stream scheme is more effective than other schemes. Furthermore, even when the join selectivity factor is small, the processing time required in the stream scheme to execute a query is almost the same as in the conventional scheme. In such a case, the number of reproduction times(Im/B) becomes less because Im becomes smaller as the join selectivity factor decreases. That is, when the join selectivity factor is small, the stream scheme executes a query in almost the same fashion as the conventional scheme.

## 5. Conclusions

We have presented a stream oriented execution scheme for relational database operations. This scheme is based on the stream processing principle. By combining stream processing with demand-driven control, this scheme exploits pipeline concurrency and makes it possible to execute relational operations with limited resources while avoiding the complexity of resource management. In this paper, we have presented a basic algorithm for the stream-oriented parallel processing scheme and have also discussed the features and the effectiveness of the stream scheme.

As mentioned in Subsection 3.3, two implementation approaches for the stream scheme can be considered. A distributed query processing system based on the stream scheme is currently being developed in a local area network environment[10].

We believe that concepts included in the proposed stream-oriented parallel processing scheme will contribute to the development of relational database systems.

## Acknowledgements

The authors would like to thank Professor Takashi Masuda (Univ. of Tsukuba) for valuable discussions and suggestions. Special thanks are due to Kazuhiko Kato (Univ. of Tsukuba) for his valuable discussions and support in evaluating the performance of the stream scheme.

## References

[1] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," Comm. ACM, Vol.13, No. 6, pp377-397, 1970.

[2] G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," Proc. IFIP '77, pp. 993-998, 1977.

[3] J.M. Smith and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," Comm. ACM, Vol.18, No. 10, pp.568-579, 1975.

[4] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan and K.C. Smith, "RAP.2 - An Associative Processor for Database and Its Applications," IEEE Trans. on Compt., Vol.c-28, No. 6, pp.446-457, 1979.

[5] H. Boral and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," ACM Trans. on Database Systems, Vol. 6, No. 2, pp.227-256, 1981.

[6] D. Bitton, H. Boral, D.J. DeWitt and W.K. Wilkinson, "Parllel Algorithms for the Execution of Relational Database Operations," ACM Trans. on Database Systems, Vol. 8, No. 3, pp.324-353, 1983.

[7] Y. Kiyoki, K. Tanaka, N. Kamibayashi and H. Aiso, "Design and Evaluation of a Relational Database Machine Employing Advanced Data Structures and Algorithms," Proc. 8th International Symposium on Computer Architecture, pp.407-423, 1981.

[8] Y. Kiyoki, M. Isoda, K. Kojima, K. Tanaka, A. Minematsu and H. Aiso, "Performance Analysis for Parallel Processing Schemes of Relational Operations and a Relational Database Machine Architecture with Optimal Scheme Selection Mechanism," Proc. 3rd International Conference on Distributed Computing Systems, 1982.

[9] Y. Kiyoki, R. Hasegawa and M. Amamiya, "An Execution Scheme for Relational Database Operations with Eager and Lazy Evaluations," Trans. of IPSJ, Vol. 26, No. 4, pp.685-695, July 1985 (in Japanese).

[10] Y. Kiyoki, K. Kato, and T. Masuda, "A Stream-oriented Approach to Distributed Query Processing in a Local Area Network, Research Report, Institute of Information Sciences and Electronics, Univ. of Tsukuba, March 1986.

[11] M. Amamiya and R. Hasegawa, "Dataflow Computing and Eager and Lazy Evaluation," New Generation Computing, Vol. 2, No. 2, 1984.

[12] S.B. Yao, "Optimization of Query Evaluation Algorithms," ACM Trans. on Database Systems, Vol. 4, No. 2, pp.133-155, June 1979.

[13] D. Bitton, D.J. DeWitt, and C. Turbyfill, "BenchmarkingDatabase Systems, A Systematic Approach," Proc. of 9th International Conference on VLDB, Nov. 1983.

[14] Programmers Reference Manual for the Sun Workstation, Sun Micro System, Inc., 1982.

# PARTITIONING RELATION FOR PARALLEL PROCESSING IN FAST LOCAL NETWORKS

C. T. Yu*, K. C. Guh*, D. Brill+ and A. L. P. Chen+

*Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60680

+System Development Corporation
2500 Colorado Avenue
Santa Monica, California 90406

## Abstract

A partition and replicate strategy for processing distributed queries referencing no fragmented relation is sketched. An optimal algorithm is given to decide which relation is to be partitioned into fragments, which copy of the relation is to be used, how the relation is to be partitioned and where the fragments are to be sent for processing. The time complexity of the algorithm is $O(N_d\ m \log m + r \log r)$, where $N_d$ is the total number of copies of the relations referenced by the query, m is the number of sites and r is the number of relations. Under the situations that all sites have the same processing speed and the cost functions for data transmission and local processing cost are linear, the time complexity of the algorithm is reduced to $O(r\ |AS(Q)| \log |AS(Q)| + r \log r)$, where $|AS(Q)|$ is the number of sites having a copy of a relation referenced by the query.

## 1. Introduction

Distributed query processing is an important factor for the performance of a system with the databases distributed in a network. Many distributed query processing algorithms [ApHY, BeCh, BeGo, BlLu, CDFG, ChBH, Chan1, Chan2, ChHo, ChHu ChLi1, ChLi2, ChLi3, EpSW, GoSh, HeYa, KeYa, Luk2, SDD1V1, SDD1V2, Woka, Wong, WOHL, Yaos, YCTB, YLCC, YuOz] have been proposed. Most of these algorithms assume that the data communication cost is dominant and make use of semijoins to reduce the amount of data transfer. While such an assumption is reasonable for long-haul networks where data communication costs are high, it may not be valid for fast local networks. In contrast the "fragment and replicate" (a fragmented relation remains to be fragmented, while other relations are replicated at the sites of the fragmented relation) query processing strategy was used in distributed Ingres [EpSW]. Its main feature is to allow parallel processing. However, for many queries, substantial data transfer is required before parallel processing can take place unless many relations are duplicated in many sites.

We have developed two algorithms. The first algorithm, the semijoin algorithm [YCTB, YCTBL], is an extension to the SDD-1 algorithm [SDD1V2] which assumes that the most important cost is the number of bytes transmitted between network nodes. The algorithm was extended to support fragmented and replicated relations. The other algorithm, the replicate algorithm [BrTY, YGCC] is derived from distributed Ingres [EpSW]. It assumes that local processing cost dominates network costs and uses fragmented relations to maximize the amount of parallelism in operations.

We have done the performance analysis for these two algorithms [BrTY, TBCD]. We found that the replicate algorithm outperforms the semijoin algorithm in a fast local network environment. Thus, in a fast local network, it is preferable to use a fragment and replicate algorithm. However, if no relation referenced by a query is fragmented, it is necessary to decide the relation and its copy to be partitioned into fragments, how the relation is to be partitioned and where the fragments are to be sent for processing. This paper provides answers to the above questions and gives an optimal algorithm in time $O(N_d\ m \log m + r \log r)$, where $N_d$ is the total number of copies of the relations referenced by the query, m is the number of sites and r is the number of relations. Under the special case that all sites have the same processing speed and the cost functions for data transmission and local processing cost are linear, the time complexity reduces to $O(r\ |AS(Q)| \log |AS(Q)| + r \log r)$, where $|AS(Q)|$ is the number of sites having a copy of a relation referenced by the query.

## 2. Problem definition and notations

A relation $R_i$ is partitioned into a number of fragments $\{F_{ij}\}$ if $|R_i| = \sum_j |F_{ij}|$.

Suppose $\{R_1, R_2, \ldots R_r\}$ are the relations to be joined as specified in a query and all of them are not fragmented. One of them, say $R_f$, will be chosen to be partitioned into d fragments, for some integer d. Then, the fragments will be assigned to a set of d sites. The other relations, if not present at each of the d sites, will be replicated at those sites. The answer is the union of the joins of the fragments with the other realtions at the d sites. The problem is to decide (i) the relation to be partitioned i.e. $R_f$ (ii)

0190-3918/86/0000/1021 $01.00 © 1986 IEEE

the number of fragments the relation is to be partitioned i.e. d (iii) the size of each of the d fragments (iv) the sites where the fragments are to be assigned (v) the copy of the partitioned relation to be used, if multiple copies exist, such that the response time (in terms of both local processing cost and communication cost) is minimized.

The following notations are used with respect to the given query Q.

AS(Q): Accessed Sites; the set of sites containing at least a copy of a relation referenced by Q.

NAS(Q): Non-Accessed Sites; the set of sites containing no relations referenced by Q.

$CS(R_f)$: The set of sites each containing a copy of the relation $R_f$ to be partitioned.

As in [YGCC, YGZT], the cost of joining relations $R_1,...R_n$ is given by $h(R_1) + ... + h(R_n)$, where

$$h(X) = \begin{cases} f(X) & \text{if there exists a fast access path(e.g. index) for the joining operation involving X} \\ n(X) & \text{otherwise,} \end{cases}$$

with $f(X) << n(X)$.

This is a rough approximation, with the implicit assumption that the join is restrictive.

$W(R_f,p)$: The weight of site p. This includes the cost to transmit required data into the site from other sites and the processing cost at the site but not including the costs associated with relation $R_f$, which is chosen to be partitioned. The weight

$$= \sum_{D_{ij} \text{ in } p, i \neq f} h(D_{ij})/Sp(p) + \sum_{D_{ij} \text{ not in } p, i \neq f} [t(D_{ij})/Tspeed + n(D_{ij})/Sp(p)]$$

where $D_{ij}$ is a copy of relation i at site j and Tspeed is the communication speed of network, and Sp(p) is the processing speed of site p; t() is the cost function for data transfer. The first term represents the local processing cost for those data already at site p, taking into consideration the processing speed of the site. The second term gives the local processing cost and the data trasmission cost for the data to be transferred into site p.

d-partition: A relation is partitioned into d fragments.

$par(R_f)$: The cost function to partition $R_f$. This is assumed to depend only on the size of the relation $R_f$.

$PS(R_f,d)$: The set of the processing sites each of which is assigned one fragment of $R_f$, if the partition is a d-partition, i.e., $|PS(R_f,d)| = d$.

$PT(PS(R_f,d),s)$: A d-partition of $R_f$ by using the copy of $R_f$ at site s (if multiple copies of the relation exist) and the processing sites $PS(R_f,d)$.

$F(R_f,d,p)$: The size of the fragment that will be assigned to site p for a d-partition of $R_f$.

$Assign(R_f,d,p,s)$: The cost to assign and to process a fragment of size $F(R_f,d,p)$ at site p, $p \in PS(R_f,d)$ by using the copy of $R_f$ at site s for partition.

$$= \begin{cases} t(F(R_f,d,p))/Tspeed + n(F(R_f,d,p))/Sp(p) & \text{if } p \neq s \\ n(F(R_f,d,p))/Sp(p) & \text{if } p=s. \end{cases}$$

It is assumed that after the partitioning process, fast access paths, if exist earlier, will be lost. Thus, the cost function for not having a fast access path, n(), is used.

$cost(R_f,d,p,s)$: The total ( processing and communication ) cost at site p when the copy of $R_f$ at site s is chosen to be partitioned into d fragments, and the fragment of size $F(R_f,d,p)$ is assigned to site p.

$= W(R_f,p) + Assign(R_f,d,p,s) + par(R_f)/Sp(s)$.

Note that since each processing site has to wait for the completion of the partitioning of $R_f$, $par(R_f)/Sp(s)$ is incurred at each processing site.

All cost functions t(), h(), f() and par() are monotonically increasing functions i.e. if the relation/fragment is larger, a higher cost will be incurred for local processing and/or data transfer.

$Res(R_f,d,s)$: The response time obtained at d-partition when the copy of $R_f$ at site s is chosen to be partitioned into d fragments and the processing sites are $PS(R_f,d)$.

$$= \max_{p \in PS(R_f,d)} \{ cost(R_f,d,p,s) \}$$

We are interested in minimizing $Res(R_f,d,s)$.

## 3. Partition strategy

### 3.1 Determine the optimal processing sites

Given a relation and its copy to be partitioned, we want to determine (1) the processing sites (and therefore the number of partitions) and (2) how the relation is to be partitioned and assigned to the processing sites.

Proposition 1 below shows that an optimal way to assign fragments to processing sites is in such a way that each processing site will have the same total cost. Thus, it remains to determine the processing sites. We will show in Propositions 3 and 4 that (i) if a site is not used as a processing site, then those sites with weights higher than or equal to that of the site should not be used as processing sites and (ii) if a site is used as a processing site, then those sites with weights smaller than or equal to that of the site should be used as processing sites. Based on results (i) and (ii), all sites are arranged in ascending order of weight. Consider site t. If site t is not a processing site, then sites (t+1) to the last site need not be considered; otherwise, site 1 to site t and possibly the next few sites will be processing sites.

Proposition 1: Let $Res(R_f,d,s)$ and $Res_1(R_f,d,s)$ be the response times obtained by the d-partitions $PT(PS(R_f,d),s)$ and $PT_1(PS(R_f,d),s)$, respectively. If the d-partition $PT(PS(R_f,d),s)$ makes $cost(R_f,d,p,s) = cost(R_f,d,j,s)$ for every pair p,j $\notin PS(R_f,d)$ then $Res(R_f,d,s) \leq Res_1(R_f,d,s)$.

Definition: S is an optimal set of processing sites for the copy of relation $R_f$ at site t if the copy is partitioned into fragments and assigned for processing at each of the sites in S to yield the optimal response time. Each site s in S is an optimal processing site.

Lemma 2: Let $Res(R_f,d,s)$ be the response time obtained by d-partition $PT(PS(R_f,d),s)$.
If $W(R_f,j) + par(R_f)/Sp(s) < Res(R_f,d,s)$, j not in $PS(R_f,d)$, then there exists a (d+1)-partition, $PT(PS(R_f,d+1),s)$, such that $Res(R_f,d+1,s) < Res(R_f,d,s)$, where $Res(R_f,d+1,s)$ is the response time obtained by including site j as a processing site.

Proposition 3: If site p is an optimal processing site and site j satisfies $W(R_f,j) \leq W(R_f,p)$ then site j is also an optimal processing site.

Proposition 4: If site p is not an optimal processing site and site j satisfies $W(R_f,j) \geq W(R_f,p)$ then site j is also not an optimal processing site.

Suppose $R_f$ is the relation to be partitioned. After the sites are arranged in ascending order of weights, the best solution is obtained by assigning the fragments of $R_f$ using Proposition 1 to the first d sites, for some d. We now seek to determine d.

One simple-minded way is to compute the response time for a given d by assigning the fragments of $R_f$ to the first d sites using Proposition 1. Then d is increased by 1 and the process is repeated until an increase in the number of sites yields an increase in the response time or the total number of sites, m, is reached. For a given d, finding the response time for the first d sites takes O(d) time. Since the process may be repeated up to m times, the time complexity is $O(m^2)$.

A more efficient process is as follows. We first consider the middle site, the m/2-th site. If this is not an optimal processing site ( determined by Proposition 5), then it is sufficient to consider the first (m/2 - 1) sites. We then repeat the process by checking the m/4-th site. If the m/2-th site is an optimal processing site, then the 3m/4-th site will be examined. In other words, a binary search is performed on the set of sites. The process of determining whether the i-th site is an optimal processing site will be shown to take at most O(i) time. Since the binary search process is an O(log m) process, the time complexity for determining the optimal number of sites is at most O(m log m). It remains to determine whether a given site is an optimal processing site.

Suppose the given site is the i-th site. Even if this given site is assigned a fragment of size zero (i.e. the site is not a processing site), the time incurred at this site is $T= W(R_f,i) + par(R_f)/Sp(s)$ ( the copy of $R_f$ at site s is partitioned). Suppose each of the preceding (i-1) sites is assigned some part of $R_f$ (the parts of $R_f$ assigned to the sites need not be disjoint) such that its response time is T. The next proposition states that site i is not an optimal processing site if and only if the sum of the sizes of the fragments of $R_f$ assigned to the first (i-1) sites is greater than or equal to the size of $R_f$.

Proposition 5: Suppose size(j) is the size of the part of $R_f$ assigned to site j such that the total cost of site j, $cost(R_f,i-1,j,s) = W(R_f,i) + par(R_f)/Sp(s)$ for each site $1 \leq j \leq i-1$. Then site i is not an optimal processing site iff

$$size = \sum_{j=1}^{i-1} size(j) \geq size(R_f) \qquad (1)$$

where $size(R_f)$ is the size of relation $R_f$.

The process of determining the last processing site is given as follows.

```
BINSEARCH(low,high,result)
/* The relation to be partitioned is $R_f$; the copy
of $R_f$ to be partitioned is at site s. The optimal
set of processing sites is 1 to result. */
    If ( low > high ) then result = high
    else {
         MID = ( low + high )/2 ; MID1 = MID - 1
         For i := 1 to MID1
         {  Assign part of $R_f$ to site i such that
            $cost(R_f,MID1,i,s)$
            $= W(R_f,MID) + par(R_f)/Sp(s)$ }
         If $\sum_{i=1}^{MID1} size(i) \geq size(R_f)$
         then BINSEARCH(MID+1,high,result)
                 /* search right half */
         else BINSEARCH(low,MID-1,result)
                 /* search left half */
    }
```

### 3.2 Determine the copy of the relation to be partitioned

When there are more than one copy of the relation to be partitioned, we need to determine which copy should be used to obtain the best response time. Suppose we use the copy $CO_u$ at the site u that has the fastest processing speed among the sites that have a copy of the relation. Let the set of the optimal processing sites and the optimal response time for this copy be $PS_u$ and $OP_u$ respectively. Proposition 7 below states that any other copy which is not in one of the optimal processing sites, $PS_u$, can not yield a better response time than $OP_u$ and should not be chosen for partitioning.

Suppose there is a copy $CO_v$ at site v which is within $PS_u$. If the set of the optimal processing sites obtained for $CO_v$ is not a subset of $PS_u$ then the response time obtained by using $CO_v$ can not be less than $OP_u$. This is shown in Lemma 6. Thus, we do not need to consider any site outside $PS_u$ as a processing site for copy $CO_v$.

Lemma 6: Let $Res(R_f,d,u)$ and $Res(R_f,d',v)$ be the optimal response times obtained by the optimal partitions $PT(PS(R_f,d),u)$ and $PT(PS(R_f,d'),v)$, respectively, by using the copies $CO_u$ and $CO_v$ of $R_f$ at sites u and v, respectively.

If $Sp(u) \geq Sp(v)$ and $PS(R_f,d') \not\subseteq PS(R_f,d)$ then $Res(R_f,d',v) > Res(R_f,d,u)$.

Proposition 7: Let $Res(R_f,d,u)$ and $Res(R_f,d',v)$ be the optimal response times obtained by the optimal partitions $PT(PS(R_f,d),u)$ and $PT(PS(R_f,d'),v)$, respectively, by using the copies $CO_u$ and $CO_v$ of $R_f$ at sites u and v, respectively.

If $Sp(u) \geq Sp(v)$ and $v \notin PS(R_f,d)$
then $Res(R_f,d,u) \leq Res(R_f,d',v)$.

When there are more than one copy of the relation to be partitioned, one method is to first use the copy at the site that has the fastest processing speed among the sites that have a copy of the relation, then discard some unnecessary copies by applying Proposition 7. The remaining copies are processed in descending order of processing speed of their residing sites. For each such copy, the processing sites will be restricted to be the intersection of the processing sites of the previous copies since, by Lemma 6, no copy can yield a better response time by using any site outside the processing sites of a previous copy. We then compare all such cases and the copy yielding the least response time is chosen. Suppose instead we choose the copy $CO_u$ at the site u that has the fastest processing speed among the sites that contain a copy of the relation to be partitioned, and obtain the response time for this copy. Let the size of the fragment assigned to site j, where a copy $CO_j$ of the relation exists, be $F_j$. (If no copy of the relation exists in a processing site of $CO_u$, then, by Proposition 7, the optimal solution for the relation is obtained.) If we choose $CO_j$ to be partitioned, we incur $par(R_f)/Sp(j) - par(R_f)/Sp(u)$ additional time at each processing site, but we save transmission cost of $t(F_j)/Tspeed$ for site j. If the copy $CO_j$ is chosen to be partitioned, we can reduce the response time by at most $t(F_j)/Tspeed - ( par(R_f)/Sp(j) - par(R_f)/Sp(u) )$. In fast local area networks, the transmission cost is supposed to be small in comparision with processing cost. Thus, the copy at the site with the fastest processing speed should be a good choice.

### 3.3 Discard of certain non-accessed sites

The following result (Proposition 9) says that if a site j is an non-accessed site and the weight of this site for partitioning relation $R_f$ is greater than or equal to the optimal response time obtained by partitioning $R_f$, then this site should not be considered as one of the processing sites for any partitioned relation $R_r$, where $|R_r| \leq |R_f|$, since it will result in higher response time. Thus, if we order the relations in descending order of size, a non-accessed site discarded in the processing of a relation is automatically discarded when a later relation is processed. Suppose site j is discarded. Any non-accessed site whose weight is greater than or equal to site j with respect to relation $R_f$ satisfies the same condition as site j. Thus, it can also be discarded.

Lemma 8: If $j \notin NAS(Q)$ and $|R_f| \geq |R_r|$
then $W(R_f,j) \leq W(R_r,j)$.

Proposition 9: Let $Res(R_f,d,u)$ and $Res(R_r,d',v)$ be the optimal response times obtained by partitioning relations $R_f$ and $R_r$, respectively. Let $PS(R_r,d')$ be the set of the optimal processing sites for partitioning relation $R_r$.
If $j \notin NAS(Q)$, $j \in PS(R_r,d')$, $|R_f| \geq |R_r|$ and $W(R_f,j) \geq Res(R_f,d,u)$
then $Res(R_r,d',v) \geq Res(R_f,d,u)$

### 3.4 An optimal algorithm

We now give an algorithm to find an optimal partition strategy.
Let R(Q) be the set of relations referenced by query Q.
The set of sites $S = AS(Q) \cup NAS(Q)$ i.e. the union of the accessed and the non-accessed sites of Q.
Among the non-accessed sites in NAS(Q), let s be the site having the fastest processing speed.

ALGORITHM PARTITION

```
(1) Estimate the total time if the query is pro-
    cessed in a single site t, $t \notin AS(Q) \cup \{s\}$.
    Set Bound = the best estimated response time
    obtained.
(2) Arrange the relations in R(Q) in descending
    order of size. Let the arranged relations be
    $R_i$, $1 \leq i \leq n$.
(3) For i = 1 to n /* from the largest relation to
    the smallest */
  3.1 From the remaining non-accessed sites, dis-
      card those sites p with
      $W(R_i,p) \geq Bound$.   /* Once a non-accessed
                     site is discarded for a re-
```

```
                    lation, it is discarded for
                    all smaller relations by
                    Proposition 9 */
3.2 Arrange all remaining sites p in ascending
    order of weight $W(R_i,p)$
3.3 Discard those sites p with weight
    $W(R_i,p) + par(R_i)/Sp(u) \geq Bound$
    where u is the site having the fastest pro-
    cessing speed among the sites that contain a
    copy of $R_i$.
            /* Unlike 3.1 this discard is for
            the relation $R_i$ only */
3.4 Use binary search to determine the optimal
    processing sites for the copy of $R_i$ at site
    u.
    low = 1; high= the number of sites remained
    after step 3.3.
    BINSEARCH(low,high,result)
3.5 Compute response time Res using the process-
    ing sites 1 to result.
3.6 Order the remaining copies of $R_i$ that reside
    in a site j between 1 and result in descend-
    ing order of processing speed of their
    residing sites. (Let $k_i'$ be the number of
    these remaining copies). Let the arranged
    copies be $CO_t$, $1 \leq t \leq k_i'$. $result_0$ = result.
    For t=1 to $k_i'$
    { low = 1; high = $result_{t-1}$
      If( $1 \leq site(CO_t) \leq high$) ) /* By Proposi-
                 tion 7, if $site(CO_t)$, the site
                 where $CO_t$ resides, is beyond
                 high, it can not yield a
                 response time better than $res_{t-1}$
                 */
        { BINSEARCH(low, high, $result_t$)
          Compute the response time $res_t$ for the
          processing sites 1 to $result_t$.
          res = MIN { res, $res_t$ }
          }
      }
3.7 Bound = MIN { Bound, Res }.          []
```

### 3.5 Given the relation $R_f$,and its copy at site s, to be partitioned ~~and the optimal processing sites~~, determine the sizes of the fragments to be assigned to the sites.

By Proposition 1, the optimal assignment is to make each processing site have the same total cost. Thus, for each optimal processing site i,

$W(R_f,i) + par(R_f)/Sp(s) + assign(R_f,d,i,s) = c$

where c is the total cost (and hence the response time) to be determined and $F(R_f,d,i)$ is the fragment of $R_f$ to be assigned to site i.

$F(R_f,d,i)$

$= assign^{-1}( c - W(R_f,i) - par(R_f)/Sp(s))$ (2)

where the assign function is in terms of the transmission cost function t() and the join processing function n().

Since $\sum_i F(r_f,d,i) = |R_f|$,

we can solve for c by substituting (2) into it and therefore $F(R_f,d,i)$ can be solved. We can solve Equation (2) for $F(R_f,d,i)$ as a function of c in constant time. We need to repeat this for d fragments. Thus, c can be solved in O(d) if the relation is partitioned into d fragments and the sizes of fragments can also be computed in O(d).

## 4. Time complexity

In the algorithm PARTITION, it takes $O(N_d)$ to obtain the best response time for Step (1)[YGZT], where $N_d$ is the total number of the copies of the relations. At Step (2), it takes O(r log r) to arrange the r relations. We know that the higher the processing speed of a non-accessed site the smaller the weight of the site. Thus, we can pre-arrange all sites in ascending order of weight ( or descending order of processing speed) as if they are non-accessed sites. For a given site, check if the site is non-accessed or not and if it is, determine whether its weight is greater than or equal to the Bound. A binary search of sites can be employed here, since the sites are in ascending order of weight. Thus, the non-accessed sites with weight ≥ Bound can be determined in O(log m) in Step 3.1, where m is the number of sites. For step 3.2, it takes $O(|AS(Q)| \log |AS(Q)|)$ to order the accessed sites and then takes $O(|AS(Q)|+|NAS(Q)|)$ to merge the non-accessed and the accessed sites since the non-accessed sites are arranged in ascending order of weight. At Step 3.3, binary search can be used to discard sites it takes O(log m'), where m' is the total number of the remaining non-accessed and accessed sites. It takes at most O(m' log m') for the BINSEARCH algorithm for Step 3.4. At step 3.5, it takes at most O(m'). Thus, the time complexity for Steps 3.1-3.5 is O(m log m). Suppose there are $k_i$ copies of relation $R_i$. At Step 3.6, Steps 3.1 to 3.5 is repeated at most $k_i$ times with time $O(k_i m \log m)$. We will repeat this process for every relation. Thus, it needs $O(N_d m \log m + r \log r)$ for the algorithm PARTITION.

## 5. Special situation

Suppose both the transmission cost function and the join cost function are linear i.e. $n(X) = a_1 X$ and $t(X) = a_2 X$ for some constants $a_1$ and $a_2$. Then

$$assign(R_f,d,j,s) = \begin{cases} a_1F(R_f,d,j)/Sp(j) + a_2F(R_f,d,j)/Tspeed & \text{if } j \neq s \\ a_1F(R_f,d,j)/Sp(j) & \text{if } j = s. \end{cases}$$

where s is the site where the copy of $R_f$ is to be partitioned. We can rewrite it as

$assign(R_f,d,j,s) = b_{js} F(R_f,d,j)$ where

$$b_{js} = \begin{cases} a_1/Sp(j) + a_2/Tspeed & \text{if } j \neq s \\ a_1/Sp(j) & \text{if } j = s. \end{cases}$$

If we make use of this special property for the algorithm BINSEARCH, the complexity of this algorithm can be reduced to O(m) instead of O(m log m).

In algorithm BINSEARCH, in order to determine whether site (d+1) is a processing site using Proposition 5, we will set $cost_1(R_f,d,j,s) = W(R_f,d+1) + par(R_f)/Sp(s)$ and compute $F_1(R_f,d,j)$ for every site j, $1 \le j \le d$. If we substitute the assign function into $cost_1(R_f,d,j,s)$, we have

$$
\begin{aligned}
cost_1(R_f,d,j,s) \\
&= W(R_f,j) + par(R_f)/Sp(s) + b_{js} F_1(R_f,d,j) \\
&= W(R_f,d+1) + par(r_f)/Sp(s)
\end{aligned}
$$

If we solve for the size of the fragment, we have $F_1(R_f,d,j) = (W(R_f,d+1) - W(R_f,j))/b_{js}$ for $1 \le j \le d$.

The summation of the sizes of the fragments is

$$
\begin{aligned}
R(d) &= \sum_{j=1}^{d} F_1(R_f,d,j) \\
&= \sum_{j=1}^{d} (W(R_f,d+1) - W(R_f,j))/b_{js}.
\end{aligned}
$$

After obtaining R(d), we will continue to search the right half or the left half.

Case 1: Search the right half

In this case, we will determine whether (d'+1)-th site should be an optimal processing site, where d'>d. We need to compute

$$
\begin{aligned}
R(d') &= \sum_{j=1}^{d'} F_1(R_f,d',j) \\
&= \sum_{j=1}^{d'} (W(R_f,d'+1) - W(R_f,j))/b_{js} \\
&= \sum_{j=1}^{d} (W(R_f,d'+1) - W(R_f,j))/b_{js} \\
&\quad + \sum_{j=d+1}^{d'} (W(R_f,d'+1) - W(R_f,j))/b_{js} \\
&= \sum_{j=1}^{d} (W(R_f,d'+1) - W(R_f,d+1))/b_{js} \\
&\quad + \sum_{j=1}^{d} (W(R_f,d+1) - W(R_f,j))/b_{js} \\
&\quad + \sum_{j=d+1}^{d'} (W(R_f,d'+1) - W(R_f,j))/b_{js} \\
&= (W(R_f,d'+1) - W(R_f,d+1)) \sum_{j=1}^{d} 1/b_{js} + R(d) \\
&\quad + \sum_{j=d+1}^{d'} (W(R_f,d'+1) - W(R_f,j))/b_{js}
\end{aligned}
$$

If $1/b_{js}$, $1 \le j \le d$, has been accumulated, the first term can be done at constant time. What remains to be computed is the size of the fragment that will be assigned to site j, $d+1 \le j \le d'$, and their summation. (note that $F_1(R_f,d',j) = W(R_f,d'+1) - W(R_f,j)$ ) It needs only (d'-d) steps instead of d' steps.

Case 2: Search the left half

In this case, we will determine whether (d"+1)-th site should be an optimal processing site, where d" < d. We need to compute

$$
\begin{aligned}
R(d") &= \sum_{j=1}^{d"} F_1(R_f,d",j) \\
&= \sum_{j=1}^{d"} (W(R_f,d"+1) - W(R_f,j))/b_{js} \\
&= \sum_{j=1}^{d} (W(R_f,d"+1) - W(R_f,j) \\
&\qquad + W(R_f,d+1) - W(R_f,d+1))/b_{js} \\
&\quad - \sum_{j=d"+1}^{d} (W(R_f,d"+1) - W(R_f,j))/b_{js} \\
&= \sum_{j=1}^{d} (W(R_f,d+1) - W(R_f,j))/b_{js} \\
&\quad = ( W(R_f,d+1) - W(R_f,d"+1)) \sum_{j=1}^{d} 1/b_{js} \\
&\quad + \sum_{j=d"+1}^{d} (W(R_f,j) - W(R_f,d"+1))/b_{js} \\
&= R(d) - ( W(R_f,d+1) - W(R_f,d"+1)) \sum_{j=1}^{d} 1/b_{js} \\
&\quad + \sum_{j=d"+1}^{d} (W(R_f,j) - W(R_f,d"+1))/b_{js}
\end{aligned}
$$

Again, if R(d) is obtained and $1/b_{js}$, $1 \le j \le d$, is accumulated, then R(d") can be computed in (d-d") steps.

Suppose there are m sites. In binary search, we start at the m/2-th site; this takes m/2 steps to compute R(m/2); then the 3m/4-th site is checked if the right half is searched or the m/4-th site if the left half is searched. But no matter which half is searched, according to the above argument, it takes only m/4 steps to compute R(3m/4) or R(m/4). Thus, it will take m/2, then m/4, m/8 steps and so forth for the sequence of searches. Since the binary search process terminates in (log m) steps,and

$$\sum_{i=1}^{\log m} m/2^i = m-1$$

the algorithm BINSEARCH takes O(m) instead of O(m log m) time.

Suppose not only the cost functions n() and t() are linear, but all sites have the same processing speed. Then, the time complexity is further reduced.

When all sites have the same processing speed, those non-accessed sites will have the same weight associated with the relation to be partitioned. Therefore, by Propositions 3 and 4, if a non-accessed site should (not) be an optimal processing site, then all non-accessed sites should (not) be optimal processing sites. Thus, we only need to use those accessed sites and one non-accessed site to determine the optimal processing sites in algorithm BINSEARCH. The complexity of the algorithm BINSEARCH becomes O(|AS(Q)|+1).

Furthermore, we do not need to repeat the search for the optimal processing sites for every copy of the relation to be partitioned. The copy at the site which has the smallest weight among those sites having a copy of the relation to be partitioned, will yield the best response time. This will be shown in Proposition 12.

Lemma 10: If $a_1 > a_2 > 0$ and $x_1 \geq x_2 > 0$ then $a_2 x_1 + a_1 x_2 \leq a_1 x_1 + a_2 x_2$

Proof: $a_2 x_1 + a_1 x_2 - a_1 x_1 - a_2 x_2$
$= (a_2 - a_1) (x_1 - x_2)$
$\leq 0$ []

Lemma 11: If $a_1 > a_2 > 0$ and $x_1 \geq x_2 > 0$ then $x_1/a_1 + x_2/a_2 \leq x_1/a_2 + x_2/a_1$.

Proof: $x_1/a_1 + x_2/a_2$
$= (a_2 x_1 + a_1 x_2)/(a_1 a_2)$
$\leq (a_1 x_1 + a_2 x_2)/(a_1 a_2)$ (Lemma 10)
$= x_1/a_2 + x_2/a_1$ []

Proposition 12: Let t be the site having the smallest weight among all sites containing copies of the relation, $R_f$, to be partitioned. Let $Res(R_f,p_t,t)$ and $Res(R_f,p_v,v)$ be the response time for using the copy at site t and for using a different copy at site v, respectively.
If all sites have the same processing speed, then $Res(R_f,p_t,t) \leq Res(R_f,p_v,v)$.

Proof:
Case 1: $p_v > p_t$
By Lemma 6, $Res(R_f,p_v,v) > Res(R_f,p_t,t)$.
Case 2: $p_v \leq p_t$
Suppose $Res(R_f,p_t,t) > Res(R_f,p_v,v)$. Let $F_{jt}$ be the size of the fragment assigned to a processing site j, $1 \leq j \leq p_t$, by partitioning the copy at site t. We know that $F_{jt} = ( Res(R_f,p_t,t) - W(R_f,j) - par(R_f)/Sp(t) )/b_{jt}$. Since all sites have the same processing speed, $b_{jt} = b_{jv}$, $j \neq t$ and $j \neq v$. Since $b_{jt} = b_{jv}$ for $j \neq t$ and $j \neq v$, we have that $F_{jv} < F_{jt}$ for $j \neq t$ and $j \neq v$ because $Res(R_f,p_v,v) < Res(R_f,p_t,t)$ and $Sp(t) = Sp(v)$. Since all sites have the same processing speed, $b_{tv} = b_{vt}$, $b_{vv} = b_{tt}$ and $b_{tt} < b_{vt}$. We can have $F_{tv} + F_{vv} < F_{tt} + F_{vt}$

Thus,
$$\sum_{j=1}^{p_v} F_{jv} < \sum_{j=1}^{p_v} F_{jt} \leq \sum_{j=1}^{p_t} F_{jt} = |R_f|,$$
a contradiction that the copy of $R_f$ at site v is partitioned. Thus, $Res(R_f,p_v,v)$ can not be smaller than $Res(R_f,p_t,t)$.
[]

By Proposition 12, if we use the copy of the partitioned relation at the site with the smallest weight to determine the optimal processing sites, the optimal response time of partitioning the relation is obtained. This takes only $O(|AS(Q)|)$. It takes $O(|AS(Q)| \log |AS(Q)|)$ to arrange the accessed sites in ascending order of weight. Thus, it takes $O(r |AS(Q)| \log |AS(Q)|)$ at Step 3 of the algorithm PARTITION if there are r relations. Therefore, the complexity of the algorithm is reduced to $O(r |AS(Q)| \log |AS(Q)| + r \log r)$.

Since the number of relations referenced by a query and the number of accessed sites are usually very small, the algorithm will run very fast. In situation where the join cost function is difficult to estimate accurately, a linear function may serve as a first order approximation. The processing speed of a site may depend on many factors, such as the number of users at a given time, the query type, the job mix etc. In the absence of accurate strategies, the assumption that all sites have the same processing speed is applicable.

## 6. Conclusion

In a fast local network, it is preferable to use a fragment and replicate strategy to process queries. However, if no relation referenced by a query is fragmented, it is necessary to decide which relation is to be partitioned into fragments, which copy of the relation should be used, how the relation is to be partitioned and where the fragments are to be sent for processing. An optimal algorithm in time $O(N_d m \log m + r \log r)$ has been given to provide answers to the above questions. Under special situations that all sites have the same processing speed and the cost functions for data transmission and local processing cost are linear, the time complexity reduces to $O(r |AS(Q)| \log |AS(Q)| + r \log r)$.

## REFERENCES

[ApHY] Apers P., Hevner A. and Yao S. B. OPTIMIZATION ALGORITHM FOR DISTRIBUTED QUERIES. IEEE Transactions on Software Engineering, 1983.

[BeCh] Bernstein P. A. and Chiu D-M. W. USING SEMI-JOINS TO SOLVE RELATIONAL QUERIES. JACM, 1981, pp. 25-40.

[BeGo] Bernstein P. A. and Goodman N. THE THEORY OF SEMI-JOIN. Technical Report, CCA, Nov. 1979.

[BiDT] Bitton, D., DeWitt, D. and Turbyfil, C., "Benchmarking Database Systems: A Systematic Approach", VLDB, 1983.

[BlLu] Black P. A. and Luk W. S. A NEW HEURISTEC FOR GENERATING SEMI-JOIN PROGRAMS FOR DISTRIBUTED QUERY PROCESSING. IEEE COMPSAC, 1982.

[BrTY] Brill, D., Templeton, M. and Yu, C. "Distributed Query Processing Strategies in Mermaid : A Frontend to Data Management Systems", IEEE Data Engineering, 1985, pp. 211-218.

[Brit] Britton Lee Inc. "IDM 500 Software Reference Manual", version 1.3, September 1981.

[CDFG] Chan A., Dayal U., Fox S., Goodman N., Ries D. and Skeen D. OVERVIEW OF AN ADA COMPATIBLE DISTRIBUTED DATABASE MANAGER. ACM

SIGMOD 83, pp. 228-242.

[Chan1] Chang J. M. A HEURISTIC APPROACH TO DISTRIBUTED QUERY PROCESSING. VLDB, 1982.

[Chan2] Chang J. M. QUERY PROCESSING IN A FRAGMENTED DATA BASE ENVIRONMENT. Bell Lab., Technical report, 1982.

[ChBH] Chiu, D. M., Bernstein, p., and Ho, Y. C., "Optimizing Chain Queries is a Distributed Database System", SIAM J. Comput., February 1984.

[ChHo] Chiu D-M. W. and Ho Y. C. A METHOD FOR INTERPRETING TREE QUERIES INTO OPTIMAL SEMI-JOIN EXPRESSIONS. ACM SIGMOD, 1980, pp. 169-178.

[ChHu] Chu, Wesley W. and Hurley, P., "Optimal Query Processing for Distributed Database Systems", IEEE Transactions on Computers, Vol c-31, No. 9, Sept. 1982, pp. 835-850.

[ChLi1] Chen, A. L. P. and Li, V. O. K. "Deriving optimal semi-join programs for distributed query processing", Proc. IEEE INFOCOM, San Francisco, California, April 1984.

[ChLi2] Chen, A. L. P. and Li, V. O. K. "Optimizing star queries in a distributed database system", VLDB, Singapore, August 1984.

[ChLi3] Chen, A. L. P. and Li, V. O. K. "Improvement algorithms for semi-join query processing programs in distributed database systems", IEEE Transactions on Computers, Nov. 1984.

[EpSW] Epstein R., Stonebreaker M. and Wong E. DISTRIBUTED QUERY PROCESSING IN RELATIONAL DATABASES SYSTEM. ACM SIGMOD 1978, pp. 169-180.

[GoSh] Goodman N. and Shmueli O. TRANSFORMING CYCLIC SCHEMES INTO TREES. ACM SIGACT-SIGMOD Conference on Principles of Databases, 1982.

[HeYa] Hevner A. and Yao S. B. QUERY PROCESSING IN DISTRIBUTED DATABASE SYSTEMS. IEEE Transaction on Software Engineering, Vol. 5, No. 3, 1979, pp. 177-187.

[JaKo] Jarke, M. and Koch, J., "Query Optimization in Database Systems", in ACM Computing Surveys, June 1984.

[KeYa] Kerschberg L. and Yao S. B. OPTIMAL DISTRIBUTED QUERY PROCESSING. Bell Lab., Holmdel, 1980.

[LMHL] Lohman, G., Mohan, C., Hass, L., Lindsay, B., Selinger, P. and Wilms, P., "QUERY PROCESSING IN R*" IBM Research Report RJ4272, April, 1984.

[Luk2] Luk W. C. and Luk L. OPTIMIZING QUERY PROCESSING STRATEGIES IN A DISTRIBUTED DATABASE SYSTEM. Simon Fraser University, Burnaby, B. C. Canada.

[Rein] Reiner D. (guest editor) IEEE Database Engineering, Special Issue on Query Processing, Sep., 1982.

[SDD1V1] Goodman N., Bernstein P. A., Wong E., Reeve C. and Rothnie J. B. QUERY PROCESSING IN A SYSTEM FOR DISTRIBUTED DATABASES (SDD-1). Technical Report, CCA 1979.

[SDD1V2] Bernstein P. A., Goodman N., Wong E., Reeve C. and Rothnie J. B. QUERY PROCESSING IN SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES. TODS, Vol. 6, No. 4, Dec. 1981, pp. 602-625.

[SeAd] Selinger, P., and Adiba, M. "Access Path Selection in Distributed Database Systems", Proceedings of The First International Conference on Distributed Data Bases, Aberdeen, 1980.

[TBHK] Templeton, M., Brill, D., Hwang, A., Kameny, I., and Lund, E. "An overview of the Mermaid system - A frontend to heterogeneous databases", IEEE Eascon83, Washington, Sept. 1983.

[TBCD] Templeton, M., Brill, D., Chen, A., Dao, S. and Lund, E. "Mermaid Experences with Network Operations", IEEE Data Engineering, 1986(to appear).

[Ullm] Ullman J. D. PRINCIPLES OF DATABASE SYSTEM. Computer Science Press, 2nd edition, 1982.

[WOHL] Williams et. al. R*: AN OVERVIEW OF THE ARCHITECTURE. Proc. 2nd International Conference on Databases, 1982.

[WoKa] Wong, E. and Katz, R. H. "Distributing a database for parallelism", ACM SIGMOD, 1983, pp.23-29.

[Wong] Wong E. RETRIEVING DISPERSED DATA FROM SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES. Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, 1977.

[Yaos] Yao, S. B. "Optimization of Query Evaluation Algorithms", ACM TODS, Vol 4, No. 2, June 1979, pp. 133-155.

[YCTB] Yu C. T., Chang C. C., Templeton M., Brill D. and Lund E. ON THE DESIGN OF A DISTRIBUTED QUERY PROCESSING STRATEGY. ACM SIGMOD, 1983, pp. 30-39.

[YCTBL] Yu, C. T., Chang, C. C., Templeton, M., Brill, D., and Lund, E. "Mermaid: An algorithm to process queries in a fragmented database environment", IEEE Transactions on Software Engineering, August, 1985, pp. 795-810.

[YGCC] Yu C. T., Guh K. C., Chang C. C., Chen C. H., Templeton M. and Brill D. AN ALGORITHM TO PROCESS QUERIES IN A FAST DISTRIBUTED NETWORK. IEEE Real-Time Systems Symposium 1984, pp. 115-122.

[YGZT] Yu, C. T., Guh, K. C., Zhang, W., Templeton, M., Brill, D., and Chen, A., "Algorithms to process Distributed Queries in Fast Local Networks", University of Illinois at Chicago, August 1985.

[YLCC] Yu C. T., Lam K., Chang C. C. and Chang S. K. A PROMISING APPROACH TO DISTRIBUTED QUERY PROCESSING. Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Feb. 1982, pp. 363-390.

[YLGT] Yu, C. T., Lilien, L., Guh, K. C., Templeton, M., Brill, D. and Chen, A., "Adaptive Techniques for Distributed Query Optimization", International Conference on Data Engineering, Los Angeles, Feb. 1986. (to appear)

[YuOz] Yu, C. T. and Ozsoyoglu, M. Z. "An Algorithm for Tree-Query Membership of A Distributed Query", IEEE COMPSAC, 1979, pp.306-312.

# A COMPILER-ASSISTED CACHE COHERENCE SOLUTION FOR MULTIPROCESSORS

*Alexander V. Veidenbaum*

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61801

**Abstract** -- The existing solutions to multiprocessor cache coherence problem are not suitable, in our opinion, for systems with a large number of processors. We propose a new solution in which a compiler generates cache management instructions. Conditions necessary for cache coherence violation are defined. The structure of a program and its dependence graph are used to detect when these conditions become true, and the instructions to enforce coherence are generated. No communication between processors is required at run-time to enforce coherence. The correctness of the solution is proved.

## Introduction

Several multiprocessor architectures have been built or are being built that can truly be called large-scale ([4], [5], [14], [2]). They can have several hundred processors sharing memory and all working on solving a single problem. Such multiprocessors are characterized by a long memory access time making the use of cache memories very important. However, a cache coherence problem makes the use of caches difficult. In this paper we discuss the proposed solutions to the cache coherence problem and why they may not be suitable for a large-scale multiprocessor. We propose a different solution that relies on a compiler to manage the caches. We define conditions necessary for a data incoherence to occur. A restructuring compiler, such as Parafrase [9], can be used to detect when such conditions arise and to generate cache management instructions to enforce coherence.

## Existing solutions to the cache coherence problem

Let us examine the proposed solutions to the cache coherence problem (see [18]) for use on large-scale multiprocessors. The solutions can be divided into the following groups:

(1) Solutions based on a single shared resource.
These include a central directory scheme of [15] and a shared cache scheme of [17].
A large number of processors can not access the shared resource without severe performance degradation. That's why these solutions are not extendable to large-scale multiprocessors.

(2) Bus-based solutions.
An example of such a solution is that of Goodman [6].
These solutions are also using a shared resource, the bus, and could have been considered in the first group. We put them in a separate group because they seem to us capable of supporting a larger number of processors then the solutions in the first group, but not hundreds of processors on a single bus.

(3) Cacheability attribute processed during virtual address translation.
This scheme was used in C.mmp [3]. Again a central resource, shared page tables, can be identified in this solution. The overhead of changing the cacheabilty attribute seems large to us for a truly extendable solution.

In addition, most modern processors contain a translation buffer (TLB) acting as a cache for page table entries. When a cache attribute needs to be changed all of the TLBs need to be updated. Therefore, we may have solved the data cache coherence problem, but we have created a TLB coherence problem.

A variant of this solution has been proposed which allows the cacheability of read-only data. In this case the attribute is never changed during the execution of a program. This solution is restrictive and may not allow a large number of references to be cached.

## Multiprocessor architecture

The architecture we are interested in is a shared-memory multiprocessor. It consists of a global shared memory, a global interconnection network, and proces-

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84-10110 and DCR84-06916, and the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001.

0190-3918/86/0000/1029 $01.00 © 1986 IEEE

sors with private caches. The only way of exchanging data between processors is through the shared memory. The block diagram of this architecture is given in Figure 1.

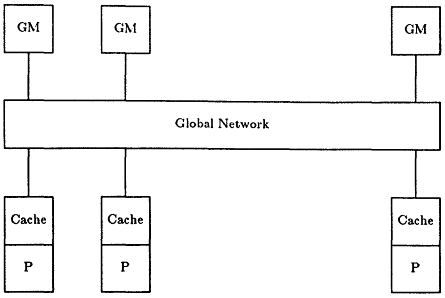


Figure 1. Multiprocessor architecture.

### Interconnection network

We assume the interconnection network to be of the shuffle-exchange type. The network has a unique path from any input to any output port. The order of accesses by a given processor to a given memory location is preserved by the network.

### Cache organization

We assume that a cache is a physical address cache, using the "Write-through" policy, and having a block size of one. We also assume that on context switch the system ensures the correctness of cache contents. Other details of cache organization will be presented in the following sections.

## Definitions

The following notation is used in this paper:

(1) $S_i$ - statement i of a program,

(2) $P_j$ - processor j of a multiprocessor,

(3) $C_j$ - private cache of processor $P_j$,

(4) M[X] - the contents of a memory location of a variable X,

(5) $C_j[X]$ - the value of X contained in the cache of processor $P_j$,

(6) $X^s$ - a store of X,

(7) $X^f$ - a fetch of X,

(8) $X^a$ - an access to X, i.e either a fetch or a store, In some cases we may qualify the access type by a processor that performed the access as follows: $X^{a(P_j)}$.

(9) $(Y)^*$ - zero or more repetitions of Y.

### Data dependencies

The definitions of this section follow those of [8] and [16] which should be consulted for any additional information on dependence analysis.
For each statement $S_i$ we define two sets, IN and OUT.

-- $IN(S_i)$ is a set of variables the statement uses.

-- $OUT(S_i)$ is a set of variables the statement generates.

The following types of dependencies are defined between two statements $S_i$ and $S_j$ if there is a flow of control path from $S_i$ to $S_j$ and $S_i$ precedes $S_j$ on that path.

-- $S_j$ is data flow dependent on $S_i$ if the set $IN(S_j) \cap OUT(S_i)$ is not empty.

-- $S_j$ is data antidependent on $S_i$ if the set $IN(S_i) \cap OUT(S_j)$ is not empty.

-- $S_j$ is output dependent on $S_i$ if the set $OUT(S_j) \cap OUT(S_i)$ is not empty.

-- $S_j$ is control dependent on a conditional statement $S_i$ if its execution depends on the execution of $S_i$ and the path chosen after that.

If the two statements are both enclosed by DO loops a dependence may exist on some but not all iterations of the enclosing loops. Also, a dependence may exist between $S_i$ executed in one iteration and $S_j$ executed in another iteration of a loop. Such a dependence is called a cross-iteration dependence. The data dependence information can be represented by a graph on the statements of a program, called a data dependence graph.

### Loop types

A program can have any of the following four loop types:

(1) A DOALL loop - a loop which does not have any cross-iteration deoendencies [11].

(2) An R(N,1) loop - a loop solving a first-order linear recurrence [8].

(3) A DOACROSS loop - a loop that has a dependence graph cycle but can be executed in a pipelined fashion [13].

(4) A serial loop - a loop with a dependence graph cycle where pipelined execution is not possible.

The type of a loop is determined by the dependence graph of statements nested in it. The first three types of loops can be executed on multiple processors to achieve performance improvement.

## A memory reference sequence

A memory reference sequence **X** for a memory location of a variable X is an ordered sequence of accesses generated by a program as observed at the memory:

$$\mathbf{X} = X^{s} \left( (X^{f})^{*} \, (X^{s})^{*} \right)^{*}$$

Any access $X^{a}$ generated by a program can be identified by a sequence number i giving the position of $X^{a}$ in **X**. We use a subscript i to indicate the sequence number, $X^{a}_{i}$. Let us assume that the sequence number is know a priori for any access.

For a serial program the sequence is unique for a given set of input data. For a deterministic parallel program the sequence is unique except for possible permutations in any of the fetch subsequences.

The model of computation we use assumes that the following two conditions are satisfied at any time T when a processor generates $X^{a}_{i}$ :

A1. $M[X] = X^{s}_{j}$, where $j < i$ and $\forall$ k, $j < k < i$: $X_{k}^{a}$ was a fetch,

A2. If $X^{a}_{i}$ is a store then $\forall$ j, $j < i$: $X^{a}_{j}$ has been performed.

In other words, the value in the shared memory is always current. All the stores preceding the current access have been completed but none of the following stores have been completed. This is necessary to keep things "simple" in the system where the only way of communicating between processors is through shared memory. These conditions a equivalent to enforcing the data dependence constraints between the statements. We assume that the ordering is enforced through the use of synchronization primitives.

## Cache incoherence

We define an incoherence as a condition when:

1. a processor $P_{j}$ performs a memory fetch $X_{i}^{f}$

and

2. $M[X] \neq C_{j}[X]$ at such time.

An incoherence cannot occur if $X_{i}$ is a store. Note that we require a processor to try to fetch X, otherwise the fact that the memory and the cache have different values is not an error.

Using the notion of the reference sequence let us define the necessary conditions for the cache incoherence to occur.

### Lemma 1

The two conditions necessary for a cache incoherence to occur at processor $P_{j}$ issuing $X^{f}_{i}$ are:

C1. $C_{j}[X] = X_{l}^{a(P_j)}$, where $l < i$ and $X_{l}^{a}$ was the last access to X by $P_{j}$.

C2. $\exists$ k, $l < k < i$: $M[X] = X_{k}^{s(P_m)}$, where $j \neq m$. (If more then one such stores took place let $X_{k}^{s}$ be the one with largest sequence number.)

That is, a value of X is present in the cache of $P_{j}$ (C1), and a new value has been stored in the shared memory by another processor since the last access by $P_{j}$ (C2).

### Proof

(1) The first condition must be true, for if it is not true the value is fetched out of GM and is therefore correct.

(2) Let us assume that an incoherence occurred on $X_{i}^{f(P_j)}$, i.e. $M[X] \neq C_{j}[X]$ at that time, but C2 is not true. Let us also assume that this is the first time any of the conditions {A1, A2, C1, C2} we specified are not satisfied.
C2 being false means:

$$M[X] = X_{k}^{s(P_m)},$$
$$\text{where } k \geq i \text{ or } k \leq l \text{ or } j = m.$$

If $j = m$ then the value in the cache is that stored by processor j. Since it was the last store before $X_{i}^{f}$ we have $M[X] = C_{j}[X]$, which is not true. Therefore $j \neq m$.
If $k > i$ then condition A2 has not been satisfied when the store of $X^{s}_{k}$ occurred since not all the preceding accesses have been made. We assumed that this was the first time any of the conditions were false, but found that A2 must have been false earlier. Therefore, $k \leq i$.
k cannot be equal to i since $X_{k}$ is a store and $X_{i}$ is a fetch.
k cannot be equal to l since this will imply $j = m$ which we have shown is false.
For the case of $k < l$ we have two possibilities: $X_{l}$ is a store or $X_{l}$ is a fetch. If $X_{l}$ is a store then $M[X] = C_{j}[X]$ when $X_{i}$ is issued since no other processors store into X after $X_{l}$ and neither does $P_{j}$. If $X_{l}$ is a fetch we again have $M[X] = C_{j}[X]$. Since the contents of memory and the cache are not equal, k cannot be less then l.
It follows then that C2 is a necessary condition for the incoherence to occur.□

### Corollary

Cache incoherence cannot occur on a variable that is assigned only once during program execution.

The proof follows from the fact that the conditions C1 and C2 of Lemma 1 cannot be true at the same time.

### Detecting the necessary conditions

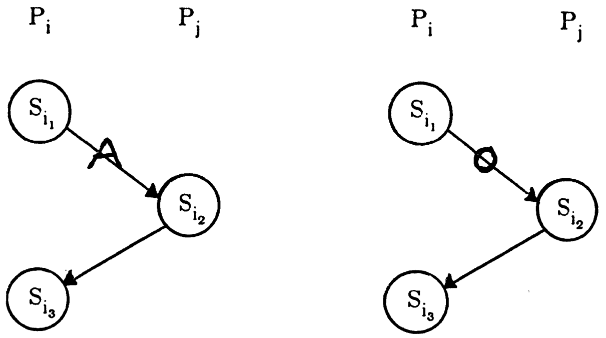
Our goal is to have the restructuring compiler detect the conditions of Lemma 1 and generate the cache management instructions. The power of restructuring compilers is in the ability to perform the data dependence analysis. Therefore, let us express the conditions necessary for cache incoherence to occur in terms of data dependencies. It follows from Lemma 1 that:

(1) The value of X has to be in the cache of processor $P_j$. This implies that a statement $S_{i_1}$ was executed by $P_j$ such that X belongs to $IN(S_{i_1})$ or X belongs to $OUT(S_{i_1})$.

(2) A different processor has computed a new value of X. This implies that a statement $S_{i_2}$ was executed by $P_k$ such that X belongs to $OUT(S_{i_2})$, $k \neq j$.

(3) Finally, an incoherence can only occur if $P_j$ is fetching X. This implies a statement $S_{i_3}$ is executed by $P_j$ such that that X belongs to $IN(S_{i_3})$.

From the above, $S_{i_2}$ depends on statement $S_{i_1}$, and statement $S_{i_3}$ depends on statement $S_{i_2}$. One of the following two dependence graphs are possible:



In addition to the above dependence structure, it is necessary that $S_{i_2}$ be executed on a different processor. How can we detect that? We propose to use the loop type information for this purpose. (If other types of parallelism are being exploited they can be taken care of in a similar fashion.) Recall that only three of the four types of loops we are dealing with can be executed on multiple processors. Let us concentrate on DOALL and DOACROSS loops. We assume that one or more consecutive iterations of such a loop are assigned to a processor. By definition, any dependence between two statements inside a DOALL loop is not across iterations, but there are cross-iteration dependences in DOACROSS. It follows that a statement $S_i$ in a DOALL dependent on a statement $S_j$ in the same loop is executed on the same processor as $S_j$. In a DOACROSS loop, two statements with a cross-iteration dependence are executed on different processors, while statements with a dependence on the same iteration are executed on the same processor.

Using the above we first construct a simplified algorithm using only the loop type information. We then extend it to consider the dependence structure and the flow of control.

### A cache management algorithm

Let us assume that the following instructions are available for cache management:

Flush. This instructions invalidates the entire contents of a cache.

Cache_on. This instruction causes all global memory references to be routed through the cache.

Cache_off. This instruction causes all global memory references to by-pass the cache and go directly to memory.

In addition, the cache state, on or off, must be part of the processor state and has to be saved/restored on context switch. Processes are created in the cache-off state.

The algorithms uses loop types for its analysis as follows:

(1) A DOALL loop does not have any dependencies between statements executed on different processors. Therefore condition C2 of Lemma 1 is false, and any shared memory access in such a loop can be cached.

(2) A serial loop is executed by a single processor, and hence condition C2 of Lemma cannot be true.

(3) A DOACROSS or an R(N,1) loops do have cross-iteration dependencies. Therefore condition C2 can be true. Initially, let us just turn caches off in such loops.

The algorithm is shown in Figure 2. The algorithm turns cacheing on and off, depending on the type of loop a program enters. (Note that cache management instructions inserted in parallel loops are executed once by every participating processor.) Conditions for incoherence are checked at loop boundaries. In addition, procedure and function calls which may have parallel loops in called routines are considered. The algorithm does not really consider individual dependencies or look for the dependence structure satisfying conditions of Lemma 1. It states that within certain loop types (or nests of this type) the conditions cannot be satisfied or cacheing is not allowed.

The algorithm is simple enough to be executed either at compile time or at run time. At compile time we know which loops can be executed on multiple processors, but whether they will be executed as such depends on run-time processor allocation. At run time we know which loops get multiple processors. For example, if a DOACROSS is serialized we can turn the cacheing on during the whole time the loop is executed.

```
cache_state := on
insert Cache_on, Flush
For every statement in a program do
        case(statement type)
          of(DO)
               case(DO type)
                 of(DOALL)
                      insert Cache_on, Flush after
                              the DO statement
                      push(cache_state)
                      cache_state := on
                 of(serial)
                      if cache_state = off
                      then   insert Cache_on, Flush before
                                      the DO statement
                      fi
                      push(cache_state)
                      cache_state := on
                 of((DOACROSS, R(N,1))
                      insert Cache_off after the DO statement
                      push(cache_state)
                      cache_state := off
               endcase
          of(DOEND)
               do_type := type of DO for which this is DOEND
               old_cstate := cache_state
               cache_state := pop()
               if cache_state=on AND old_cstate=off
               then   insert Cache_on after DOEND
               fi
               if       cache_state=on
                 AND
                      do_type={DOALL,DOACROSS,R(N,1)}
               then   insert Flush after DOEND
               fi
          of(CALL)
               insert Cache_off before the CALL statement
               if cache_state = on
               then   insert Cache_on, Flush after
                              the CALL statement
               fi
        endcase
enddo
```

Figure 2. The cache management algorithm

## Correctness proof

We will prove the correctness of the algorithm by showing that the conditions necessary for an incoherence to occur are not satisfied in programs processed by the algorithm. The conditions necessary for an incoherence on a variable X to occur when executing a statement $S_i$ are:

CC1: Statement $S_i$ depends on statement $S_j$ through a variable X.

CC2: $S_i$ is executed by processor $P_l$ and $S_j$ is executed by processor $P_m$, $l \neq m$.

CC3: X is in the cache $C_l$ prior to the execution of $S_i$ and $C_l[X] \neq X^{s(P_m)}$.

A general loop structure enclosing statements i and j is shown in Figure 3. Note that a statement not in any loop can be represented by a statement enclosed in a serial loop with one iteration. Therefore some of the loops shown in Figure 3 can be removed to obtain simpler cases.

```
DO a
      .
      .
      DO b
            .
            .
            DO c
                  .
                  .
                  Sj
                  .
                  .
            DOEND c
            .
            .
      DOEND b
      .
      .
      DO d
            .
            .
            DO e
                  .
                  .
                  Si
                  .
                  .
            DOEND e
            .
            .
      DOEND d
      .
      .
DOEND a
```

Figure 3. Program structure

We have to show that for every path in the control flow graph of our program between $S_j$ and $S_i$ one of the conditions above is false for any $S_i$. A note about loops with GOTOs exiting the loop: we assume such loops have a single exit path regardless of how the loop was exited. This is usually required for correct synchronization, while it also simplifies the analysis of control flow between loops in our proof.

Let us consider the innermost loop $DO_e$ enclosing $S_i$. Three cases are possible:

(1) $DO_e$ is a DOACROSS or an R(N,1) loop.
In this case a Cache_off instruction is issued by every processor executing the iterations of the $DO_e$.

Therefore, X will be fetched from memory when $S_i$ is executed. This is equivalent to CC3 being false.

(2) $DO_e$ is a DOALL loop.
In this case a Flush instruction is issued by every processor executing iterations of $DO_e$. Therefore CC3 is false.

(3) $DO_e$ is a serial loop.
Consider all the loops enclosing $S_i$ between $DO_e$ and $DO_d$. Let us skip all loops e, ..., k+1 that are serial to $DO_k$, $k \leq d$, such that:

A. $DO_k$ is a DOACROSS.
In this case $S_i$ is enclosed by a nested set of serial loops $DO_e$ through $DO_{k+1}$. The nested set is executed on one processor and a Flush instruction is executed before the $DO_{k+1}$ according to the Algorithm 1. Therefore CC3 is false.

B. $DO_k$ is a DOALL.
In this case a Flush instruction is executed by every processor executing iterations of $DO_k$. Since all of the loops $DO_{k+1}$ through $DO_e$ are serial, every processor executing $S_i$ has executed a Flush instruction. Therefore CC3 is false.

C. Otherwise $DO_k$ is a serial, i.e. all of the loops $DO_d$ through $DO_e$ are serial.
Now let us consider the loop type of $DO_a$:

$\alpha$ $DO_a$ is a DOACROSS.
In this case a Flush instruction is executed before $DO_d$ according to Algorithm 1. Since $S_i$ is executed by the same processor, CC3 is false.

$\beta$. $DO_a$ is a DOALL or a serial loop.
In this case we have to consider the loops enclosing $S_j$.

I) $DO_b$ through $DO_c$ are all serial.
In this case both loop nests $DO_b$ through $DO_c$ and $DO_d$ through $DO_e$ are serial. Since they are nested in a DOALL or a serial loop they will be executed on the same processor. Hence condition CC2 cannot be true for statements $S_i$ and $S_j$.

II) Otherwise consider the outermost loop $DO_k$, $b \leq k \leq c$, that is not serial, i.e. it is a DOALL, a DOACROSS, or an R(N,1).
In this case a Flush instruction is executed after $DOEND_k$ by the same processor that is to execute $S_i$. Therefore CC3 is false.

Finally, let us consider CALL statements. Suppose $S_j$ is a subroutine or function CALL. In this case a called routine may have loops in it enclosing the statement actually generating X. In addition, the called routine may not have any cache management instruction in it (that is why we turn cacheing off before a call). However, in all but one of the cases considered in the proof a Flush or Cache_off instruction is performed by a processor that executes $S_i$. Therefore, CC3 is false when $S_i$ is executed regardless of the type of statement of $S_j$.

The one case that does not have the cache flushed is C.$\beta$II. In that case $S_i$ and $S_j$ are executed by the same processor, and there is no Flush on any path between the two statements. The algorithm inserts a Flush instruction after a CALL statement taking care of this case.□

### Improving the cache management algorithm

In this section we describe the extensions of the cache management algorithm. The first one allows cacheing to be used in DOACROSS loops. The second and third attempt to reduce the number of cache flushes by doing a more detailed dependence and flow analysis.

(1) Cacheing of data inside DOACROSS loops.
A DOACROSS loop is executed by assigning successive iterations to different processors (mod the number of processors available). The cross-iteration dependencies that exist in a DOACROSS are thus between statements executed by different processors. Synchronization primitives have to be used between these processors to ensure that dependencies are satisfied (say, the classical P and V primitives).
A straight-forward solution is to issue a Flush instruction after the V by each processor executing a statement depending on a statement executed by another processor. Since the shared memory has the current value after the V instruction and the cache does not have anything, the value will be fetched out of global memory. Otherwise the DOACROSS loops can be treated the same way as the DOALL loops by the cache management algorithm.
A more interesting solution is possible for architectures that support memory access combined with synchronization, such as [7] or [5]. In this case we can identify exactly which word requires synchronization and invalidate just one word, not the entire cache. Cache controller design has to be modified to allow invalidation of individual words, preferably as part of a synchronized fetch. The correct value is fetched from memory and can be put in the cache. Any subsequent unsynchronized fetch will use the value out of the cache.

The processor performing a synchronized store does not have to do anything special, just write the data through to shared memory.

The last solution requires a synchronized data access for every variable on which a cross-iteration dependence exists. Any attempt to minimize the number of synchronization primitives, as proposed by [12], or use of implied synchronization may result in an incorrect execution of a program.

(2) The simplified algorithm we presented does not really look for the dependence structure implied by the Lemma 1 conditions. Specifically, it does not check the existence of a statement bringing a variable into a cache prior to the execution of the two statements with a dependence on two different processors. An incoherence cannot occur if such a statement does not exist. In such a case it is not necessary to invalidate the contents of a cache.

Consider a DOACROSS loop with cacheing enabled. Assume each processor executing this loop performed a Flush instruction just after it entered the loop. Let us know consider a statement $S_i$ that uses a variable X generated in another iteration of the DOACROSS. If we examine all the flow of control paths from the first statement in a loop to $S_i$ and determine there are no generations or uses of X on any of them then we do not have to invalidate X in the cache before $S_i$ (a single assignment condition). If the above is true for all the cross-iteration dependencies in the loop we do not need a Flush instruction in this DOACROOS.

This technique can be extended to analyze the whole program to avoid Flushing after every parallel loop.

(3) The algorithm uses data dependence information indirectly, through loop types. A beginning and end of a parallel loop are synchronization points it detects and uses to issue cache management instructions. This synchronizes all dependencies from statements in such loops to statements outside of such loops. However, this synchronization point may be located much earlier than the statement using the data. Another synchronization point may exist later in the program that takes care of an earlier one. For example, consider the program segment in Figure 4. If no statement in DOALL$_b$ has dependence arcs to statements in DOALL$_c$ or any code between the two loops, and the flow of control always goes through DOALL$_c$ after passing through DOALL$_b$, a Flush is not necessary after DOALL$_b$.

The correctness proof can be easily extended to include the algorithm improvements shown in this section.

```
DO a
      .
      .
      DOALL b
      .
      DOEND b
      .
      .
      DOALL c
      .
      DOEND c
DOEND a
```

Figure 4. Program example.

## Conclusions

We have presented a solution to the cache coherence problem in which a restructuring compiler generates cache management instructions. An algorithm to do that is presented and its correctness proved. In this solution each processor manages its own cache without any additional communication with other processors. The cache management instructions are very simple, affect only the processor issuing them and have a small fixed cost. Finally, the total number of such instructions issued by any processor is a function of loop bounds and loop structure of a program, not of the number of processors used or the number of stores in the program. That is why we believe the solution is scalable to multiprocessors of any size.

## Acknowledgments

We thank all the members of the Cedar project who discussed this proposal with us, especially Madhu Sharma, Dave Padua, Rich Barton, Dan Sorensen, and Hoichi Cheong.

## References

[1] Utpal Banerjee, "Data Dependence in Ordinary Programs," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-76-837, November, 1976.

[2] W. Crowther et al., "Performance measurements on 128-node Butterfly$^{(TM)}$ parallel processor" Proceedings of the 1985 International Conference on Parallel Processing, pp. 531-540, 1985.

[3] S.H. Fuller and S.P. Harbison, "The C.mmp multiprocessor," Department of Computer Science, Carnegie-Mellon University, Technical Report, 1978.

[4] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared-Memory Parallel Machine," IEEE Trans. on Computers, Vol. C-32, No. 2, pp. 175-189, February 1983.

[5] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh, "Cedar -- a Large Scale Multiprocessor," Proceedings of the 1983 International Conference on Parallel Processing, pp. 524-529, August, 1983.

[6] J.R. Goodman, "Using cache memory to reduce processor-memory traffic," Proceedings 10th International Symposium on Computer Architecture, pp.124-131, June, 1983.

[7] "Heterogeneous Element Processor Principles of Operation," HEP technical documentation series, Denelco, part. no.9000001, February, 1981.

[8] David J. Kuck, "The Structure of Computers and Computations," Volume 1, John Wiley and Sons, New York, 1978.

[9] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," Fourth International Computer Software and Applications Conference, October, 1980.

[10] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, vol. C-24, no. 12, pp. 1145-1155, December, 1975.

[11] S. F. Lundstrom and G. H. Barnes, "Controllable MIMD Architecture," Proceedings of the 1980 International Conference on Parallel Processing, pp. 19-27, 1980.

[12] S.P. Midkiff, "Compiler Generated Synchronization for High Speed Multiprocessors", M.S. Thesis, University of Illinois at Urbana-Champaign, May 1986

[13] D.A. Padua Haiek, "Multiprocessors: Discussions of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-990, November 1979.

[14] G.F. Pfister et al, "The IBM Research Parallel Processor Prototype," Proceedings of the 1985 International Conference on Parallel Processing, pp. 764-772, 1985.

[15] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," Proceedings AFIP National Computer Conference, vol.45, pp.749-753, 1976.

[16] Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1982.

[17] W.C. Yen, J.H. Patel, E.S. Davidson. "Shared cache for multiple-stream computer systems," IEEE Trans. on Computers, Vol. C-34, No. 1, pp. 56-65, January, 1983.

[18] W. C. Yen, D. W. L Yen, and K. S. Fu, "Data Coherence Problem in a Multicache System," IEEE Trans. on Computers, Vol. C-34, No. 1, pp. 56-65, January 1985.

# LOAD-BALANCED TASK ALLOCATION IN LOCALLY DISTRIBUTED COMPUTER SYSTEMS

*Hongjun Lu*
Computer Science Center
Honeywell, Inc.
Golden Valley, MN 55427

*Michael J. Carey*
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT — This paper presents a heuristic algorithm for allocating the tasks in a task force to a set of execution sites in a locally distributed computer system. The algorithm dynamically allocates tasks on arrival with the objectives of balancing the load of the system and of minimizing the communications costs for the task force to the extent possible within the constraints imposed by load balancing. Results from experimenting with the algorithm on pipelined task forces arising from distributed database queries indicate that it finds the optimal allocation in most cases, and also that load-balanced task allocation effectively improves system performance.

## 1. INTRODUCTION

Task allocation and load balancing are major issues associated with the design of distributed computing systems. When a task or set of tasks are to be executed in a distributed system, they must be properly allocated to sites in the system. The problem of selecting appropriate sites is known as the *task allocation* problem, and a number of solution methods have been developed for this problem [2,4,10,12,13]. A common trait of these task allocation schemes is that they operate using static information. That is, they ignore an important dynamic characteristic of distributed computer systems: the probability of one or more sites being idle while tasks are waiting at other sites can be remarkably high [6]. *Load balancing* aims to reduce this probability via *task migration*, which is the process of transferring tasks from one site to another (idle or less heavily loaded) site to either begin or continue execution there. In the class of task migration schemes known as sender-initiated schemes [3,6], tasks are migrated upon arrival in the system by the site at which they arrive. This is similar in some ways to more traditional task allocation schemes, but the allocation decision is made dynamically and the load status of the system is considered. To distinguish this dynamic, load-based approach to task allocation from static approaches, we refer to it as *load-balanced task allocation*.

In this paper we present a heuristic algorithm for solving an instance of the load-balanced task allocation problem. The particular problem that we consider here is unique as compared to most previous load balancing work, as we address the problem of selecting execution sites for an *entire distributed program* consisting of a number of tasks which are to be executed concurrently and which communicate with one another. Such distributed programs are known as *task forces* [5]. The particular class of task force that motivated this study is distributed database queries. In distributed relational database systems, queries are usually decomposed into sequences of *data moves* and *subqueries*. The subqueries may be executed concurrently at different sites in the system in a pipelined fashion to achieve good performance [7,8]. Note that the problem of allocating subqueries to sites in a locally distributed database system in a load-balanced fashion is a variant of the load-balanced task allocation problem.

The load-balanced task allocation problem addressed in this paper differs from previous work on task allocation in several ways. First, and most importantly, load-balanced task allocation is a dynamic problem. It takes a task force and the current load status of the system as inputs, and its main objective is to achieve a load-balanced system dynamically through proper task allocation. Second, given load balancing as the primary objective, the communications cost for executing the task force is minimized as a secondary objective. Third, each task in the task force is assumed to have a *feasible assignment set* that specifies the sites to which the task may be allocated. This constraint arises in our application (distributed query processing) because each relation is stored at only a subset of the sites in the system. Finally, since load-balanced task allocation is dynamic in nature and must be accomplished quickly at runtime, we require a solution method that is capable of solving the problem quickly.

The remainder of this paper presents our heuristic algorithm and summarizes the results of its evaluation. Section 2 defines the load-balanced task allocation problem more precisely, and Section 3 describes the details of our algorithm. In Section 4, we present some performance evaluation results plus two enhancements to the algorithm. Finally, Section 5 outlines our plans for future work.

## 2. THE LOAD-BALANCED TASK ALLOCATION PROBLEM

In many research efforts, the load of a processing site $s_i$ is represented by the number of tasks currently being served or awaiting service at that site, $N(s_i)$ [6,11]. That is, $LD(s_i) = N(s_i)$. In order to quantitatively measure the degree of "balancedness" of a system, we extend Livny's definition of the *load unbalance factor* [6], $UBF$, to be the variance of the system's load distribution:

$$UBF = \frac{\sum_{j=1}^{n}(LD(s_j) - \overline{LD})^2}{n} = \frac{\sum_{j=1}^{n}(N(s_j) - \bar{N})^2}{n}$$

$N(s_j)$ is the number of tasks at site $s_j$ and $\bar{N}$ is the average number of tasks per site after all query units are allocated. Using this definition, the load-balanced task allocation problem for a locally distributed system with $n$ sites, $\{s_1, ..., s_n\}$, can now be expressed as follows:

**Given:**

(1) A number of tasks $\{t_1, ..., t_m\}$, to be executed concurrently as a task force.

(2) A feasible assignment set $S_i = \{s_{i_1}, ..., s_{i_k}\}$ for each task $t_i$ $(1 \le i \le m)$.

(3) An *initial load vector* specifying the intial load at each site $s_j$, $1 \le j \le n$, as given by $LD(s_j)$.

**Find:** An allocation of tasks to processing sites such that:

(1) The unbalance factor under this allocation is minimized.

(2) The total communications cost, measured as the sum of the total data transfers between communicating tasks that are allocated to different sites, is also minimized to the extent possible without increasing the unbalanced factor of the allocation.

In the following discussion, we assume that the $m$ tasks are executed in a pipelined fashion because of our application [1,9]. In this case, communication only occurs between pairs of adjacent tasks in the pipeline, so the total communications cost can be roughly approximated by the number of nonlocal task pairs (i.e., pairs of adjacent subtasks $t_i$ and $t_{i+1}$ that are allocated to different sites). However, this assumption does not affect the generality of our algorithm, and only relatively minor modifications are required in order to apply the algorithm to concurrent task forces with more general communication patterns.

## 3. A HEURISTIC APPROACH

The fact that task allocation is to be performed at runtime implies that the allocation algorithm should introduce as little overhead as possible. Heuristic methods requiring less computational effort while providing near-optimality are therefore preferable to exhaustive (optimal) solution methods. In this section, we present a heuristic algorithm for solving the load-balanced task allocation problem as defined in the previous section.

This research was partially supported by National Science Foundation Grant Number DCR-8402818, an IBM Faculty Development Award, and the Wisconsin Alumni Research Foundation.

0190-3918/86/0000/1037 $01.00 © 1986 IEEE

### 3.1. Task Allocation Heuristics

Three main heuristics are employed by the algorithm — a heuristic to control the order in which tasks are considered for allocation, a heuristic to avoid considering sites with particularly heavy loads, and a heuristic to help ensure that a good allocation site is selected for each task.

**Heuristic 1: Order of Allocation.** The notion of the *degree of freedom* of a task is used in our algorithm to determine the order in which tasks are allocated to sites. This is an important consideration, as our algorithm avoids backtracking or reconsidering previous allocation decisions. The algorithm uses two freedom-related metrics associated with each task: (1) *static freedom* — the number of sites where the task can be allocated (i.e., the cardinality of its feasible assignment set). A task with a higher static freedom value will be relatively more flexible than one with a lower value because there are more site choices available. (2) *dynamic freedom* — the sum of the current load of the sites in the task's feasible assignment set.

Using these freedom measures, tasks are allocated by our heuristic algorithm one by one in order of their degree of freedom. That is, tasks with fewer site choices are allocated earlier, and the degree of static freedom is the primary consideration. In the event of a tie, the task with the largest dynamic freedom measure is chosen, as its candidate sites are more heavily loaded.

**Heuristic 2: Elimination of Full Sites.** A site with a current load which is larger than the expected post-allocation average load is considered to be full. A site that becomes full will not be assigned any further tasks (except if it is the only site in the feasible assignment set for a given task). Thus, whenever a site is full, whether due to its initial load or to the assignment of a new task to the site, it is deleted from the feasible assignment set of each task that also has some other site in its feasible assignment set.

**Heuristic 3: Site Selection Criteria.** After determining the task to be considered next, four metrics are used to select a site for the task from its feasible assignment set. These are:

(1) The *current load* of a candidate site: This is the total number of tasks currently assigned to the site, including those tasks in the initial load of the site.

(2) The *potential load* of a candidate site: This is the number of unassigned tasks that have the site in their feasible assignment set.

(3) The *benefit* of a possible assignment: This is the communications cost that can be avoided by this assignment. In our case, since communications occur only between adjacent tasks in the pipeline, the benefit of assigning task $t_i$ to site $s_j$ is the number of adjacent tasks (e.g., $t_{i-1}$ or $t_{i+1}$) that have already been allocated to the site (either 0, 1, or 2).

(4) The *potential benefit* of a possible assignment: This measures the communications cost that might be eliminated by this assignment. In our case, this is the number of unassigned tasks which have the site in their feasible assignment set and which would form a consecutive sequence of tasks including the task being allocated if they too were assigned to the site.

Since load balancing is the main objective, an attempt is first made to allocate task $t_i$ to the site with the minimum current load among its feasible assignment sites. In the event of a tie, the benefit of assigning $t_i$ to each site with the minimum load is calculated, and the site with the maximum benefit is selected. In case of another tie, the site with the minimum potential load is chosen, with maximum potential benefit being used if there are several sites with the same minimum potential load.

### 3.2. The Basic Load-Balanced Task Allocation Algorithm

The algorithm's input includes a *feasible assignment set* for each task plus an *initial load vector* that specifies the number of existing tasks at each site when the new tasks are to be allocated. The originating site and result site for the task force are also specified as input so that the communications costs for initiating tasks at remote sites and for returning results to the result site (assuming that a user is awaiting results at that site) will be considered by the allocation algorithm. These are handled using two dummy tasks, $t_0$ and $t_{m+1}$, which are assumed not to introduce any real load at their sites, but whose communications costs play a role in influencing allocation decisions. The basic load-balanced task allocation algorithm can now be described as follows:

(1) Allocate dummy task $t_0$ to the task force's site of origin, and allocate dummy task $t_{m+1}$ to the result site for the task force.

(2) Compute the static and dynamic freedom metrics for each task $t_i$, $1 \le i \le m$.

(3) Select the next task to be allocated as the one with the least assignment flexibility using the degree of freedom metrics.

(4) Select an allocation site $s_j$ for this task ($t_i$) by choosing the site from its feasible assignment set with the least current load, considering the benefit, potential load, and potential benefit metrics in turn as necessary (to break any ties).

(5) Increment the load of site $s_j$, and recompute the freedom metrics of any unallocated tasks that have $s_j$ in their feasible assignment set. (If $s_j$'s load now exceeds the expected average, simply delete $s_j$ from their feasible assignment sets.)

(6) If any unassigned tasks remain, go to step (3).

**EXAMPLE:** Consider a task force consisting of 5 tasks, and suppose that there are 8 sites in the system. Assume that the originating and result sites for the task force are both $s_3$, that the initial system load vector is {4, 1, 1, 2, 2, 0, 1, 1}, and that the feasible assignment sets for the tasks are:

| | | | |
|---|---|---|---|
| $t_1$: | $\{s_1, s_2, s_5, s_7\}$ | $t_2$: | $\{s_2, s_4, s_5\}$ |
| $t_3$: | $\{s_3, s_5, s_7, s_8\}$ | $t_4$: | $\{s_1, s_6\}$ |
| $t_5$: | $\{s_3, s_4, s_5, s_6, s_7\}$ | | |

Our algorithm would then operate as follows:

i. The total initial load is 12 and the expected average load after allocation (rounded up) is 3. Site $s_1$ is therefore full initially, so it is deleted from the feasible assignment sets of $t_1$ and $t_4$.

ii. The static degrees of freedom for the tasks are {1, 3, 3, 4, 1, 5, 1}. The dummy tasks $t_0$ and $t_6$ are allocated to $s_3$ first.

iii. $t_4$ is allocated to $s_6$, the only site in its feasible assignment set.

iv. Both $t_1$ and $t_2$ have the same static degree of freedom value (of 3). Since $t_2$ has less dynamic freedom, it is allocated next.

v. $t_2$ is allocated to site $s_2$ because $s_2$ is the site with the minimum load (of 1) in its feasible assignment set.

vi. $t_1$ is now allocated to site $s_7$ for the same reason as in v.

vii. Since $t_3$ is next in increasing order of static freedom, it is considered next. It is allocated to site $s_8$, which has the smallest potential load.

viii. Finally, $t_5$ is allocated to site $s_6$, which has the largest benefit among the sites in $t_5$'s feasible assignment set.

ix. The final task assignment is:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| task | ($t_0$) | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | ($t_6$) |
| execution site | ($s_3$) | $s_7$ | $s_2$ | $s_8$ | $s_6$ | $s_6$ | ($s_3$) |

After assignment, the load vector is {4, 2, 1, 2, 2, 2, 2, 2}. The *UBF* for this assignment is 0.61, and the total communications cost is 5. It turns out that this is actually an optimal allocation under the problem definition.

## 4. PERFORMANCE AND ENHANCEMENTS

The heuristic algorithm described in the last section is a greedy algorithm in the sense that tasks are assigned one by one, without lookahead or backtracking. In order to evaluate the quality of the allocations generated by our algorithm, we conducted a study in which we compared its allocation decisions to those of an exhaustive search method which always finds the optimal allocation [9]. This study led to the design of two enhancements for the basic algorithm. The algorithm was also used in a simulation study to investigate the usefulness of load-balanced task allocation for distributed database systems [1,9]. We briefly summarize all of these results here.

### 4.1. Optimality of the Algorithm

Three groups of tests were conducted to investigate the optimality of the heuristic algorithm [9]. The first group of tests studied the general behavior of the algorithm, varying the number of tasks and the number of sites in the system. In the second group of tests, the initial "unbalancedness" of the system's load was varied. In the last group of tests, the tasks' feasible assignment set sizes were varied.

The results of these tests showed that, in most cases, the heuristic algorithm finds the optimal allocation. The percentage of optimal allocations generated was always at least 75%, and typically higher, for runs in the first group of tests. About 95% of the allocations obtained in this group were optimal if the unbalance factor alone was considered, which is encouraging since load balancing is the main objective. Also, more than 80% of the allocations had the same (or lower) communications costs as the optimal allocation. As the numbers of sites and tasks were increased, however, the number of optimal allocations found by the heuristic algorithm decreased. The explanation for this is that larger numbers of sites and tasks increase the number of possible allocations, so the probability of selecting an optimal allocation using a greedy algorithm decreases. It is then more likely that a near-optimal allocation will be chosen instead of the true optimum. This was especially clear in a case where each task was capable of being executed at any site, where our heuristically obtained allocations frequently resulted in higher communications costs.

### 4.2. Enhancing the Basic Algorithm

The results of these tests motivated the design of two enhancements to further reduce communications costs, improving the overall optimality of the resulting allocation of tasks to sites.

*Enhancement 1.* Enhancement 1 is applied to each task individually. If a task $t_i$ has been assigned to site $s_j$, its feasible assignment set is searched to find another site $s_k$ so that, if $t_i$ is instead assigned to $s_k$, the unbalance factor of the system is not affected but the communications cost decreases.

*Enhancement 2.* Enhancement 2 tries to group as many tasks together at the same site as possible without affecting the *UBF*. It looks at each pair of adjacent tasks $(t_i, t_{i+1})$, where $t_i$ and $t_{i+1}$ have been allocated to two different sites $s_i$ and $s_k$, and it tries to find a third task $t_j (j > i+1)$ which was also allocated to site $s_i$ (i.e., to the same site as $t_i$); it considers the possibility of reversing the choice of sites for $t_{i+1}$ and $t_j$ without affecting the *UBF* but eliminating the communications cost between $t_i$ and $t_{i+1}$.

The tests described earlier were repeated with Enhancements 1 and 2 employed, and the results indicated that they indeed improve the algorithm [9]. Enhancement 2 was especially helpful for the previously troublesome case where every task is able to run at any site; the optimal allocation was always found in this case using the algorithm with Enhancement 2.

### 4.3. Execution Time and Algorithm Complexity

The execution time and complexity of the heuristic algorithm are also important concerns. Tests were run using an unoptimized version of the algorithm, and its execution time was indeed found to be fairly small [9]. For instance, with 3-5 tasks in a task force and 8 sites in the system, its execution time on a VAX 11/780 was in the 15-40 millisecond range. In our database application [1,9], query task forces are typically small like this, and the time required to initiate a compiled query will easily dominate such task allocation times. As the numbers of tasks and sites were increased, the execution time for our heuristic algorithm was observed to be basically linear in $mn$, whereas the elapsed time for an exhaustive search was seen to rise dramatically. This agrees with a complexity analysis of the basic algorithm which determined its cost to be $O(max(mn, m^2 \log_2 m))$ for $m$ tasks with a total of $n$ candidate execution sites.

### 4.4. An Example Application

As mentioned in the Section 1, the load-balanced task allocation algorithm presented here was motivated by the load balancing problem for locally distributed database systems. We have designed and evaluated a load-balanced approach to query processing for locally distributed database systems, and this work is reported in [1,9]. In particular, a simulation study was conducted to address the impact of using our load-balanced query allocation algorithm. The results of this study indicate that load-balanced task allocation provides better performance than either static or random task allocation strategies, improving the response time and waiting time for queries, and even improving overall system throughput in many cases. Waiting time reductions of 50% or more were typical under moderate CPU loads, and throughput improvements in some cases reached 10-30%.

## 5. CONCLUSIONS AND FUTURE WORK

This paper presented a new algorithm for allocating task forces to sites in a locally distributed computer system. The algorithm is novel in that it takes the current system load into account, using load balancing as the primary objective driving the task allocation procedure, although communications costs are also considered (as a secondary objective).

Because the algorithms and ideas presented in this paper resulted from work on load balancing for distributed database systems, our discussions of task forces and our algorithm descriptions have assumed a linear, pipelined task force structure. The ideas discussed here are applicable to more general task force topologies, however. Future work is needed to evaluate the optimality of the plans generated via our heuristic algorithm (and also the complexity of the algorithm) for other task force topologies. It would also be interesting to study the performance improvements provided for various task force topologies using our approach to dynamic task allocation. Finally, we have not addressed the question of how the necessary load information is to be exchanged among the sites, making the integration of our ideas with an appropriate load information exchange policy a problem for future work as well.

## REFERENCES

[1] M. J. Carey and H. Lu, Load balancing in a locally distributed database system, *Proc. of ACM-SIGMOD Int'l Conf. on Management of Data*, May 1986.

[2] W. W. Chu, L. J. Holloway, M. T. Lan and K. Efe, Task allocation in distributed data processing, *IEEE Trans. on Computers*, C-29, 11 (Nov. 1980), 57-69.

[3] D. Eager, E. D. Lazowska, and J. Zahorjan, A comparison of receiver-initiated and sender-initiated dynamic load sharing, *Proc. of the ACM-SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Aug. 1985.

[4] V. B. Gylys and J. A. Edqards, Optimal partitioning of workload for distributed systems, *Digest of Papers, COMPCON Fall 1976*, Sept. 1976, 353-357.

[5] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl, StarOS: a multiprocessor operating system for the support of task forces, *Proc. of the 7th ACM Symp. on Operating Systems Principles*, Dec. 1979, 117-127.

[6] M. Livny, *The study of load balancing algorithms for decentralized distributed processing systems*, Ph.D. Dissertation, Weizmann Institute of Science, Aug. 1983.

[7] G. Lohman et al, Query processing in R*, in *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, eds., Springer-Verlag, 1985.

[8] H. Lu and M. J. Carey, Some experimental results on distributed join algorithms in a local area network, *Proc. of the 11th VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[9] H. Lu, *Distributed query processing with load balancing in local area networks*, Ph.D Dissertation, Computer Sciences Department, University of Wisconsin-Madison, Dec. 1985.

[10] P. R. Ma, E. Y. S. Lee and M. Tsuchiya, A task allocation model for distributed computing systems, *IEEE Trans. on Computers*, C-31, 1, Jan. 1982.

[11] L. M. Ni and K. Abani, Nonpreemptive load balancing in a class of local area networks, *Proc. of the 1981 Computer Networking Symposium*, Dec. 1981, 113-118.

[12] C. C. Price and S. Krishnaprasad, Software allocation models for distributed computing systems, *Proc. of the 4th Int'l Conf. on Distributed Computing Systems*, San Francisco, California, May 1984, 40-48.

[13] H. S. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. on Software Eng.*, SE-3, 1 (Jan. 1977), 85-93.

# A Comparative Analysis of Static and Dynamic Load Balancing Strategies

*M. Ashraf Iqbal*
Institute for Computer Applications in Science and Engineering
and
University of Engineering and Technology, Lahore, Pakistan

*Joel H. Saltz*
Institute for Computer Applications in Science and Engineering

*Shahid H. Bokhari*
Institute for Computer Applications in Science and Engineering
and
University of Engineering and Technology, Lahore, Pakistan

### *ABSTRACT*

We consider the problem of uniformly distributing the load of a parallel program over a multiprocessor system. We describe and analyze four strategies for load balancing. The performance of each of these strategies is compared on a set of problems whose structure permits the use of all four strategies.

The four strategies are (1) the optimal static assignment algorithm which is guaranteed to yield the best static solution, (2) the static binary dissection method which is very fast but sub-optimal (3) the greedy algorithm, a static fully polynomial time approximation scheme, which estimates the optimal solution to arbitrary accuracy and (4) the predictive dynamic load balancing heuristic which uses information on the precedence relationships within the program and outperforms any of the static methods.

It is also shown that the overhead incurred by the dynamic heuristic (4) is reduced considerably if it is started off with a static assignment provided by either (1), (2) or (3).

## 1. Introduction

Efficient utilization of parallel computer systems requires that the task or job being executed be partitioned over the system in an optimal or near-optimal fashion. In the general partitioning problem, one is given a multicomputer system with a specific interconnection pattern as well as a parallel task or job composed of modules that communicate with each other in a specified pattern. One is required to assign the modules to the processors in such a way that the total execution time of the job is minimized.

An assignment is said to be *static* if modules stay on the processors to which they are assigned for the lifetime of the program. A *dynamic* assignment, on the other hand, moves modules between processors from time to time whenever this leads to improved efficiency.

Given an arbitrarily interconnected multicomputer system and an arbitrarily interconnected parallel task, the problem of finding the optimal static partition is very difficult and can be shown to be computationally equivalent to the notoriously intractable NP-Complete problems [1]. However, many practical problems have special structure that permits the optimal solution to be found very efficiently.

In this paper we will consider four methods of balancing load. The first three produce static mappings of modules to processors. These static methods are: (1) the calculation of the optimal static load balance, (2) a suboptimal but very inexpensive static load balancing method, and (3) a fully polynomial time approximation scheme; the solution of which can be made to approach the optimal load balance. A dynamic load balancing method is also considered - this method allows the mapping of modules to processors to vary. These methods for balancing load are suitable for distinct but overlapping varieties of problems. These problems can arise, among other places, in signal processing and image analysis, in the solution of systems of linear equations using point or block iterative methods, in problems of adaptive mesh refinements, as well as in time driven discrete event simulation. We describe our experience with four different algorithms that we have used to solve a problem for which all these methods are applicable.

The first method finds the optimal static assignment using the bottleneck path algorithm described in [2]. This algorithm captures the execution costs of the modules or processes of the task as edge weights in an *assignment graph.* A minimum bottleneck path in this graph then yields the optimal assignment. This algorithm has moderate complexity and is guaranteed to yield the optimal static assignment.

The second method that we evaluate is the binary dissection algorithm which is derived from the work of Berger and Bokhari [3],[4]. This algorithm is very fast but does not always yield the optimal static solution.

The third scheme that we consider is based on a widely used *greedy method* described in [5], which when combined with a binary search yields an approximate solution to the static partitioning problem.

Finally we evaluate the predictive dynamic load balancing method developed by Saltz[6]. This is a dynamic algorithm in that modules are reassigned form time to time during the course of execution of the parallel program. This heuristic takes the precedence relationships of the subtasks into account when deciding whether and when to relocate modules. This additional information and the capability to relocate dynamically permits this algorithm to usually allow for higher processor utilizations than the optimal static algorithm. Each of the load balancing strategies incurs an overhead. The overheads of the static algorithms are incurred prior to the execution of the program and consists of the calculations required to decide on the assignment of modules to processors. In contrast, the overhead of the dynamic load balancing problem is incurred throughout the execution of the program.

The following section discusses in detail the problem addressed in this research. Sections 3, 4, and 5 describe the optimal static, binary dissection and greedy static load balancing schemes, along with an

Supported by NASA Contracts NAS1-17070 and NAS1-18107 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center.

0190-3918/86/0000/1040 $01.00 © 1986 IEEE

analysis of the costs of performing the calculations necessary. A proof is presented demonstrating that the greedy algorithm produces an approximate solution to the static partitioning problem which can be made to converge to the optimal solution. Section 6 contains a description of the dynamic algorithm along with descriptions of data structures required for the efficient execution of this method. Section 7 compares the performance of these four algorithms. The factors influencing the run time overhead of the dynamic load balancing method are explored. An investigation is made of the performance and overheads that result from initially balancing load using the various static load balancing methods, and then dynamically balancing load.

## 2. Formulation of Problem

The methods of load balancing discussed here are applicable to distinct but overlapping sets of problems. The following is a description of a class of problems for which all of the load balancing methods presented here are applicable. We consider the partitioning on a multiprocessor system of problems which are composed of $m$ computational modules, numbered 1 to $m$ with a chain like pattern of inter-module data dependencies, such that module $i$ is connected only to modules $i + 1$ and $i - 1$. The computation is divided into steps, and each module requires data from neighboring modules at step $s$–1, to begin the computations required for step $s$.

When the relevant domain is partitioned into strips, explicit schemes for solving time dependent partial differential equations [7], problems in discrete event simulation [8], time driven discrete event simulation [a] , as well as Jacobi and block Jacobi iterative methods [6], [9] used to solve a discretized partial differential equation can exhibit this pattern of data dependence. Pipelined algorithms in signal processing and image analysis [2] can also take this form. A similar but slightly more complex pattern of data dependence occurs in red black and multicolor SOR [10].

The importance of good load balancing strategies is accentuated when the work involved in solving a problem separates naturally into a number of subunits that is relatively small compared to the number of processors utilized, and when partitioning any one of these subunits across several processors is inconvenient or expensive.

Consider the simulation of physical processes, either by means of solving a partial differential equation or by means of a discrete event simulation. The computations relating to a particular spatial strip may be assigned to a specific process which handles all computations describing events occurring in that region. Interactions between strips are local, each strip is coupled only to its immediate neighbors. It should be noted that in adaptive methods for the explicit solution of time dependent partial differential equations [11], block iterative methods, and discrete event simulation, the subunits of computation into which the problems naturally separate may each involve substantial amounts of computation.

In signal processing, a fixed sequence of operations or transforms is applied to a long series of inputs. For example, each arriving packet of data may have to be Fourier transformed, multiplied by a fixed frequency, filtered, clipped and inverse transformed. This type of application has a chain like structure and lends itself naturally to pipelining[12]. We will index successive arriving data packets or inputs by a positive integer i. Module 1 processes i and sends the results to module 2. Each module $j$, $2 \leq j \leq m$–1, receives the modified packet $i$ from module $j - 1$, further processes it and sends it off to module $j + 1$. We will say that module j is computing step $s = i + j$, when it is working on calculations pertaining to packet $i = s - j$. When module $j$ advances from step $s$ to $s + 1$, the module processes the modified packet $i - j$.

From the above, we see that in order to advance to step $s$, module $j$ must obtain information from module $j - 1$ at step $s - 1$. While module $j - 1$ does not require information from module $j$, it is useful to impose the condition that module $j - 1$ must be no more than one step ahead of module $j$. By doing this, we ensure that no more than one modified data packet need be stored per module.

The static load balancing methods yield a mapping of modules to processors. The time required to complete a problem is determined by the processor with the heaviest load. With the dynamic load balancing method, each module may proceed at a rate constrained only by the local availability of computational resources and its data dependence on other modules. Load balancing is performed in a way that is explicitly designed to prevent processor inactivity due to a lack of data availability.

The performance of the static and dynamic load balancing methods are compared through a variety of simulations. The performance of the dynamic load balancing method may be expected to depend to some extent on the initial balance of load at the time dynamic load balancing is initiated. One would expect the performance of the dynamic load balancing method to be favorably influenced by the use of static load balancing to improve the initial load balance.

The performance of each load balancing strategy discussed here is compared on a set of problems whose structure permits the use of all four strategies. Dynamic load balancing becomes particularly desirable in problems in which the time needed for a process to complete one step is difficult to determine before the problem is mapped onto a machine, or when the time required to complete a step changes during the problem's execution.

In the case of discrete event simulations, the time required by a process to complete a step is difficult to determine a-priori, and both the cases of discrete event simulations and methods that solve time dependent partial differential equations using an adaptive grid as part of an explicit timestepping scheme, the activity in a given region may vary during the course of the solution of the problem.

## 3. The Optimal Static Algorithm

In this section we discuss briefly Bokhari's algorithm for optimally partitioning a chain structured parallel or pipelined program over a chain of processors [2]. We assume that a chain structured program is made up of $m$ modules numbered $1..m$ and has an intercommunication pattern such that module $i$ can communicate only with modules $i$+1 and $i$–1 as shown in Fig. 1. Similarly, we assume that the multiprocessor of size $n<m$ also has a chain like architecture. We work under the constraint that each processor has a contiguous subchain of program modules assigned to it. Thus the partitions of the chains have to be such that modules $i$ and $i$+1 are assigned to the same or adjacent processors. This is known as the *contiguity* constraint. The optimal partitioning would then be the assignment of subchains of program modules to processors that minimizes the load on the most heavily loaded processor.

The above problem is solved by first drawing a layered graph (Fig. 2) in which every layer corresponds to a processor and the label on each node corresponds to a subchain of modules. Every layer in this graph contains all subchains of modules i.e. all pairs $<i,j>$ such that $1 \leq i \leq j \leq m$. A node labeled $<i,j>$ is connected to all nodes $<j+1,q>$ in the layer below it for all $j$ except 1 and $m$. All nodes $<1,i>$ in the first layer are connected to the starting node $s$ while all nodes $<i,m>$ in every layer are connected to the terminating node $t$. Any path connecting nodes $s$ and $t$ corresponds to an assignment of modules to processors. For example the thick edges in Fig. 2 corresponds to the assignment of Fig. 1.

Weights can now be added to the edges of this layered graph as follows. In layer $k$, each edge emanating downwards from node $<i,j>$ is weighted with the time required for processor $k$ to process nodes $i$ through $j$ and this accounts for the total computation time on processor $k$. It is clear now that there is a path in this graph corresponding to every possible contiguous subchain assignment and the weight of the heaviest edge in a path corresponds to the time required by the most heavily loaded processor to finish. Thus to find the optimal assignment, we have to find the path in the layered graph in which the heaviest edge has minimum weight – the bottleneck path.

The bottleneck path can be found by using the following labeling procedure. Initially all nodes are given labels $L(i) = \infty$ except in the first layer, in which all nodes are labeled zero. Then starting at the top and working downwards we examine each edge $e$ emanating downwards from a layer. If this edge connects node $a$ (above) to node $b$ (below) then replace $L(b)$ by $min(L(b),max(W(e),L(a))$ where $W(e)$ is the weight

[a] D. Nicol and J. Saltz, "A Statistical Methodology for the Control of Dynamic Load Balancing," to be published as an ICASE Report.

associated with edge $e$. Once the graph has been labeled, we then find the edge incident on node $t$ which has maximum weight. Suppose the edge joining node $<i,m>$ of layer $k$ with node $t$ has maximum weight, then it means that the bottleneck path would contain the node $<i,m>$ of layer $k$ and thus modules $i$ through $m$ would be assigned to processor $k$. The rest of the bottleneck path can be found in the same manner by working upwards from layer $k$ to the top.

The number of nodes per layer in the layered graph is $O(m^2)$ and thus the total number of nodes in the graph is $O(m^2n)$. The number of edges emanating from a node is at the most $m$, thus the total number of edges would be $O(m^3n)$. As the labeling algorithm looks at each edge once, therefore the space as well as time required by this algorithm is $O(m^3n)$.

## 4. The Binary Dissection Method

The binary dissection approach to the solution of the basic partitioning problem addressed in this paper is very efficient in terms of run time and gives solutions that are very close to optimal. This algorithm is a one dimensional version of the two dimensional partitioning strategy developed by Berger and Bokhari [3],[4]. The two dimensional binary dissection method gives a heuristic for partitioning a two dimensional grid of modules, and [3] , [4] give methods for mapping this partitioning onto a variety of architectures.

The algorithm proceeds as follows. The given chain of $m$ modules is split up into two halves such that the difference of the sums of execution costs in each half is minimum. The two halves are then recursively subdivided as many times as desired. Clearly, the number of pieces into which the chain can be partitioned must be exactly $2^k$ where the integer $k$ represents the depth of partitioning.

Thus this algorithm is useful for problems in which the number of processors is a power of 2. The time required by this algorithm is $O(m\log n)$ for a problem with $m$ modules and $n$ processors since there can be no more than $\log n$ levels of partitioning with each level requiring at most one access to each module weight.

At first sight this algorithm may seem capable of yielding the optimal solution. This is not always so, as the example in Fig. 3 demonstrates. In the next paragraph we will find an upper bound on the difference between the optimal solution and the solution yielded by the binary dissection method.

Let $W_T$ represent the sum of the weights of all $m$ modules. A lower bound on the weight of the heaviest subchain $W_{OPT}$ in the optimal partition will be $W_T/n$ under the special case when all the $n$ processors are uniformly loaded. Let us designate the weight of the heaviest module by $w_{\max}$ and the weight of the heaviest subchain assigned to a processor using the techniques of binary dissection by $W_{MAX}$. Then whenever a chain is divided into two parts, the maximum difference between the two halves will be bounded by $w_{\max}$. Thus if $n$=2 then $W_{MAX} \leq W_T/2 + w_{\max}/2$. Similarly if there are $n$ processors then an upper bound on $W_{MAX}$ will be:

$$W_{MAX} \leq W_T/n + w_{\max}(\frac{1}{2}+\frac{1}{4}+...+\frac{1}{n})$$

$$\leq W_T/n + w_{\max}(n-1)/n$$

Thus the maximum difference between $W_{MAX}$ and $W_{OPT}$ will be given by the following equation under the assumption that $m>n$.

$$W_{MAX} - W_{OPT} \leq w_{\max}(n-1)/n \qquad (1)$$

## 5. The Greedy Algorithm

The Greedy algorithm yields an approximate solution to the optimal partition of a chain structured parallel or pipelined program over a chain of processors. This algorithm is based on a greedy method, which is a widely used technique and is applied to a variety of problems [5]. Sahni [1] has devised a polynomial time approximation scheme to solve the knapsack problem using a greedy method while Kernighan uses a similar approach [13] for finding optimal sequential partitions of graphs. Utilizing this method one can devise an algorithm which works in stages and at each stage a decision is made regarding whether or not the next input be included in the partially constructed solution. If the inclusion of the next input will result in an infeasible solution then this is not added to the partial solution. Greedy methods may not necessarily provide optimal answers. For example consider the binpacking problem: Given a finite set $W=\{w_1,w_2,...,w_m\}$ of $m$ different weights, find a partition of $W$ into $n$ disjoint subsets $W_1,W_2,...,W_n$, such that $n$ is minimum and the sum of the weights in each subset $W_i$ is no more than a fixed constant. The First Fit algorithm for the above problem is essentially a greedy method in the sense that it tries to place each weight in the lowest indexed subset as far as possible, but this does not result in the optimal solution [1]. If however we put an extra condition on the problem that weights $w_i$ and $w_{i+1}$ are to be placed in either the same subset or subsets $W_j$ and $W_{j+1}$ respectively then the same greedy approach will be able to find the optimal solution.

The greedy algorithm is based on the function *PROBE* (described below) and takes advantage of the fact that the weight assigned to the most heavily loaded processor in the optimal partition lies somewhere between $W_T/n$ and $W_T/n + w_{\max}$ as discussed in the previous section. The algorithm selects a trial weight $w$ in the above range and then uses the function *PROBE*. The function *PROBE(w)* returns *true* if it is possible to partition the chain of modules into subchains such that the weight of each subchain is less than or equal to $w$, and *false* otherwise. The partition that results from a successful attempt to partition the chain so that the weight of each subchain is no more than w is called the *greedy partition(w)*

```
function PROBE(Processors[1..n], Modules[1..m], w):boolean;

begin
  $i$ = 1; $j$ = 1; $p$ = 1;
  while $p \leq n$ do
    begin
      repeat
        $j$=$j$+1;
      until weight of subchain Modules[$i..j$] > $w$ or $j$=$m$;
      If $j$ = $m$ (all modules have been assigned) then return(true);
      Assign the subchain Modules[$i..j-1$] to processor $p$;
      $i$ = $j$; $p$ = $p$+1;
    end;
  return(false);
end.
```

The greedy algorithm then makes a binary search in the range $W_T/n$, $W_T/n+w_{\max}$ using the above function to find the partition for which the weight of the heaviest subchain is minimum. For each trial weight $w$ the function *PROBE* has to look at each module only once. If the above range is resolved to an accuracy of $\varepsilon$ then the greedy algorithm will find a *greedy partition*($\hat{w}$) in time proportional to $O(m\log_2(w_{\max}/\varepsilon))$ with the assurance that $W_{OPT}\leq\hat{w}\leq W_{OPT}+\varepsilon$ where as before, $W_{OPT}$ is the weight of the heaviest subchain in the optimal assignment. It is important to note that the order of the greedy algorithm is proportional to $\log(w_{\max}/\varepsilon)$ unlike other *fully polynomial time approximation schemes* in which the time complexity is polynomial in $1/\varepsilon$ as described in [1].

In the following paragraphs we will prove that if there exists an assignment with the weight of its heaviest subchain equal to $w$ then the procedure PROBE will always find that or an equivalent assignment assuming that subchains with no modules in them (empty subchains) are allowed.

*Definition:* The *weight of a partition* is the weight of its heaviest subchain.

*Notation:*

$\pi_{w,n}$ a partition with weight $w$ and $n$ subchains.

$\gamma_{w,n}$ a greedy partition with weight $w$ and $n$ subchains.

$\mu_{w,n,k}$ a mixed partition with weight $w$ and $n$ subchains in which the partition up to the first $k$ subchains is greedy and the remaining partition may or may not be greedy.

Observe that $\mu_{w,n,0}$=$\pi_{w,n}$ and $\mu_{w,n,n}$=$\gamma_{w,n}$.

Claim 1: $\mu_{w,n,k}$ can always be transformed into $\mu_{w,n,k+1}$

Proof: Move the right hand partition of subchain $k+1$ to the right until any further movement would cause the weight of subchain $k+1$ to exceed $w$ or exhaust the modules.

1. If this is possible without disturbing the right hand partition of subchain $k+2$ then $\mu_{w,n,k}$ been transformed into $\mu_{w,n,k+1}$ and the claim is correct.
2. If during the course of this movement the r.h. partition of subchain $k+1$ coincides with the r.h. partition of subchain $k+2$, this means that subchain $k+2$ is now empty (which is permitted). Continue movement of both partitions together, combining with any further partitions that may be encountered. When the threshold point is reached, $\mu_{w,n,k}$ been transformed into $\mu_{w,n,k+1}$, one or more subchains to the right of $k+1$ are empty but the claim is still correct. □

Claim 2: If there exists a $\pi_{w,n}$ then there must also exist a $\gamma_{w,n}$.

Proof: Recall that $\pi_{w,n}=\mu_{w,n,0}$.

By repeatedly applying transformation (1) above we can transform: $\pi_{w,n}=\mu_{w,n,0}\rightarrow\mu_{w,n,1}\rightarrow\cdots\rightarrow\mu_{w,n,k}\rightarrow\cdots\rightarrow\mu_{w,n,n}=\gamma_{w,n}$ □

Result: If there exists an assignment of weight $w$ then the procedure PROBE will find that or an assignment of equal weight.

## 6. The Predictive Dynamic Load Balancing Method

We assume that a computation is composed of a fixed number of computational processes or modules. The computation is divided into steps, and each module requires data from a set of other modules at step $s-1$ to begin the computations required for step $s$. Each module may proceed at a rate constrained only by the time required for the processor to perform the computations required by the module, the local availability of computational resources and data dependence on other modules. Hence at a given point in time, the system may contain modules advanced to a variety of different step numbers. In the predictive dynamic load balancing method, the data dependencies between step $s-1$ and $s$ are arbitrary, although in the section following this only chain structured data dependencies will be considered.

At any given point in the computations, the data dependency requirements place limitations on the number of steps a module may advance. As stated before, in order to advance to step $s$, a module requires data from step $s-1$ from a specific set of associated modules. The *largest step reachable* by each module is the largest step number to which a module may be advanced while maintaining the required data dependency relationships. Load balancing is performed in a way that is explicitly designed to prevent processor inactivity due to a lack of data availability.

The *potential work* of a processor is defined as the amount of time that will be required to advance all modules in a processor as many steps as possible given the data currently available from other processors. The *parallel efficiency* of a processor may be defined as the percentage of time a processor spends performing the computations required by the modules assigned to it. Transfers of modules between processors impact parallel efficiencies in a machine dependent way. The communication time required to transfer a module from one processor to another along with the degree to which that communication can be masked with computation are essential factors in this dependency.

In the predictive dynamic load balancing method to be discussed here, load is shifted between processors in a way that attempts to equalize the potential work in each processor. When the potential work of a processor falls below a predetermined threshold, load balancing is considered. A module is shifted from a neighboring processor when the neighboring processor has stored an amount of potential work greater than or equal to the threshold plus a pre-determined safety factor. If more than one neighboring processor fits this criterion, the processor with the largest potential work contributes a module. In the chain structured one dimensional problem considered here, there is only one module that a given processor can contribute at a given point in the computations. In general, the choice of the module to be contributed may not be straightforward [6].

The ability to efficiently calculate the potential work in a processor is central to the usefulness of this method. Simple and inexpensive methods for calculating potential work will now be described. The potential work stored in a processor may have to be calculated from scratch in some situations. When the computations involved in solving a problem are initiated or when modules are shifted in or out of a processor after load balancing, one must take into account both the pattern of data dependencies within a processor and the availability of data from other processors in order to calculate potential work. Given a processor which has assigned to it a value for potential work, a simpler set of computations can be performed to update the value of potential work in response to the receipt of a new datum from another processor.

It is useful at this point to describe in more detail the interaction between step numbers achievable by the modules assigned to a processor and the external data available to the processor. A linked data structure representing an undirected graph DEPEND, with weighted vertices is defined for each processor P. The I *internal vertices* represent the modules in P while the B *boundary vertices* represent modules in other processors directly coupled to modules in P. Let $z_i$ $1 \le i \le B$ represent boundary vertices and let $v_i$ $1 \le i \le I$ represent vertices within the processor. The *weight* $w_i$ of each vertex $v_i$ represents the largest step reachable by each module, given the currently available boundary information. The weight $q_i$ of each of the vertices $z_i$, represents the step of the largest available boundary variable data for the module.

The largest step reachable by a vertex $v_i$ in the processor given currently available boundary data is determined by adding one to the minimum of: (1) the largest steps reachable by all internal vertices $v_j$ linked to $v$ and (2) the step number of the latest available boundary data for the boundary vertices $z_l$ linked to $v$. The weight assigned to $v_i$ may be written as

$$w_i=\min_{j,l}(w_j,q_l)+1 \tag{2}$$

where $v_j$ and $z_l$ are linked to $v_i$.

Denote the current step number of $v_i$ as $s_i$ and the time required to advance $v_i$ one step $t_i$. The potential work associated with P at a given point in the computations may be written as

$$\sum_i (w_i-s_i)t_i \tag{3}$$

where the sum is over all $i$ corresponding to $v_i$ in P. For each boundary vertex $z_i$ the graph DEPEND may be divided into equivalence classes based on the minimum number of edges that have to be traversed to get to $z_i$. We define $r_{k,j}$ as the equivalence class of $z_k$ to which $v_j$ belongs. Note that each internal vertex belongs to B different equivalence classes, one corresponding to each boundary vertex $z_k$, $1 \le k \le B$. The proposition below states a sort of superposition principle that holds for the determination of the maximum achievable step for internal vertices in response to constraints arising from boundary vertices.

Proposition: The weight of $v_i$ is given by

$$w_i = \min_{1 \le k \le B}(q_k+r_{k,i}) \tag{4}$$

Proof: The proof is carried out by substituting the postulated solution into (2). Fix attention on an internal vertex $v_i$. Corresponding to each $r_{k,i}$ where $r_{k,i} \ge 2$ there must be an internal vertex $v_j$ linked to $v_i$ with $r_{k,j}=r_{k,i}-1$. If there were not, it would not be possible to find a shortest path from $v_i$ to $z_k$ consisting of $r_{k,i}$ edges. Moreover, there cannot be an internal vertex $v_j$ connected to $v_i$ with $r_{k,j}<r_{k,i}-1$, if there were, then $v_i$ would have a shortest path to $z_k$ consisting of fewer than $r_{k,i}$ edges. Corresponding to each $r_{k,i}$ where $r_{k,i}=1$ there is a direct edge from $v_i$ to $z_k$.

Now substituting (4) for each $w_j$ into (2) yields

$$w_i=\min_{j,l}[\min_{1 \le k \le B}(q_k+r_{k,j}),q_l]+1 \tag{5}$$

for all j,l such that $v_j$ and $z_l$ are linked to $v_i$. Equation 5 may be rewritten as

$$w_i = \min_{1 \le k \le B} \ [\min_{j,l}((q_k + r_{k,j}), q_l)] + 1$$

For each k, there exists an internal vertex $v_j$ with $r_{k,j} = r_{k,i} - 1$ connected to $v_i$ and there cannot be a vertex $v_j$ where $r_{k,j} < r_{k,i} - 1$. Hence from (5) we obtain (6)

$$w_i = \min_{1 \le k \le B, l} [q_k + (r_{k,i} - 1), q_l] + 1 \qquad (6)$$

For boundary vertices $z_l$ to which $v_i$ is directly connected, $r_{l,i} = 1$. Since all quantities involved are positive in sign, we obtain from (6) the equation (4) for $v_i$ as desired. □

We are now in a position to calculate the potential work from scratch, given values of $s_i$ and $t_i$ corresponding to all vertices $v_i$ in P. For each $v_i$ in P one may calculate $w_i$ from (4) in O(B) operations per vertex. Since there are I vertices the calculation of potential work from scratch requires O(IB) operations.

If a processor has a value of potential work assigned to it the potential work may be updated in response to the receipt of a boundary datum. One finds the weights for each vertex $v_i$ in P in the following way. By equation (4) incrementing the weight of a single boundary vertex can either leave the weight of interior vertices unchanged or increase the weight by one unit. Moreover, only interior vertices currently constrained by the incremented boundary vertex will have their weights incremented.

In response to an increment in a boundary vertex $z_k$, the weights in equivalence classes may be adjusted in order of increasing equivalence class number with only one pass necessary. Assume that $z_k$ has had its weight incremented from $q_k - 1$ to $q_k$. Before $z_k$ was incremented, the constraint on the weight of vertices in equivalence class $r_k = n$ was $q_k - 1 + n$. The constraint on the weight of vertices in equivalence class $r_k = n - 1$ after $z_k$ is incremented is $q_k + (n-1)$. The adjusting of equivalence class $r_k = n$ will have no effect on the adjustment of equivalence class $r_k = n - 1$.

If a vertex in equivalence class $r_k = n$ has a weight of less than $q_k + n - 1$ before being considered for readjustment, it is not being constrained by $z_k$. Incrementing $z_k$'s weight will consequently not affect the vertex. Since the only vertices which can possibly have their weights incremented have weights $q_k + n - 1$, the order in which vertices in an equivalence class are considered is unimportant.

Updating DEPEND may proceed as follows. The weight of the vertex in DEPEND representing $z_k$ is first incremented. In a breadth first manner beginning with the vertex representing $z_k$, DEPEND is searched for vertices whose weights must be incremented. When a vertex $v$ is found that does not require a weight increment, the search does not continue to examine other vertices linked to $v$.

In the model problem, the time and space requirements of this updating algorithm algorithm are $O(nm)$ and $O(m)$ where $n$ is the number of modules in the problem and m is the number of steps over which advancement is to proceed.

## 7. Comparison of Results

We have compared the performance of both the static load balancing methods and the predictive dynamic method through a variety of simulations. Note that with minimal computational effort, on a set of weights consisting of single precision floating point numbers, the greedy approximation scheme produces a balance identical to the optimal load balance. Thus, the performance obtained through the use of the optimal method and the greedy approximation scheme were identical, and in this section we shall simply refer to the performance of the optimal load balancing method.

Static and dynamic methods can be combined; a static load balancing may be performed before beginning work on a problem, and a dynamic load balancing policy may be utilized once work on the problem has begun. It is found that the initial use of static load balancing policies can enhance the performance of the dynamic policy when compared to the performance obtained through the initial assignment of an equal number of contiguous modules to each processor. Furthermore, both the optimal and the binary dissection static load balancing methods yield rather comparable performance when used with the dynamic predictive load balancing method. Used without a dynamic load balancing method, the optimal load balance was found to be notably superior to binary dissection, while there was hardly any difference between the optimal load balance and the greedy load balance on the test problems described here.

It is important to note that the performance measured in the case of the dynamic load balancing method is average processor utilization, the overhead required to move modules from one processor to another is accounted for through counting the average number of module shifts per step per processor. The time required for these shifts is an architecture dependent variable.

We consider a system with 16 processors and a fixed number of modules. In each trial, random deviates representing the weights of modules are drawn from a truncated normal distribution. For each set of random deviates, both the optimal static load balance and the binary dissection balance are calculated and the performance is tabulated. Simulations utilizing the predictive dynamic policy are also run using the same set of random deviates. These simulations utilize both static policies and the assignment of a fixed number of modules to each processor as starting conditions. Performance is measured by calculating the average percentage of time processors are occupied advancing modules over the course of the simulations. Performance results are averaged over 50 trials differing only in the values of the random deviates generated.

In Fig. 4 and Fig. 5 the performance obtained through the use of the static and dynamic policies is depicted. In these figures, the performance of the policies is plotted against the variance of the truncated normal distributions from which the module weights were drawn. In the experiments depicted in the above figures, the weights for the modules were drawn from truncated normal distributions with variances of 0.5, 1.0 and 2.0 and mean 1, and the problem was assumed to run for 200 steps. In Fig. 4 during each trial 64 modules were assigned to the system while in Fig. 5 96 modules were assigned to the system. In both of these cases, for all variances tested, the dynamic load balancing method outperformed both static load balancing methods. Note however, that this measure of performance does not take into account the machine dependent cost of shifting modules between processors, a cost that will be studied in more detail below. The binary dissection static method was in all cases noticeably inferior to the optimal static load balance. The use of a static load balancing method initially had a relatively minor positive impact on performance in the experiments with 96 modules, and no discernible impact at all in experiments with 64 modules. The performance impact of the initial use of a static load balancing method is quite dependent on the number of steps required to solve a problem. It will be seen later that for problems that continue for a relatively small number of steps, the initial use of a static load balancing method can markedly improve performance.

In the dynamic load balancing method, the moving of modules from one processor to another will exact a cost that will depend on the details of the machines' interprocessor communication network. We can calculate an approximate expression that estimates the overhead that results from the module transfers required by the predictive dynamic method. Let $\gamma$ be the ratio between the average cost of transferring a module and the average computation cost $C$ of advancing a module one step. Letting P represent the number of processors in the system, t be the average module transfers per step per processor, and recall that m is defined as the number of modules in the problem. The average time spent computing per processor per step is $\frac{Cm}{P}$, and the average time spent transferring modules per processor per step is $\gamma Ct$. Under the pessimistic assumption that no communication can be masked by computation, the ratio of the time spent transferring modules to the time spent in computation is $\frac{\gamma sp}{m}$.

In Fig. 6 and Fig. 7 the average number of modules that must be moved from one processor to a neighbor per step of the computation is plotted against performance for a range of values of the dynamic method's safety factor. The average processor utilization obtained through the use of optimal static load balancing and through the use of binary dissection static load balancing are depicted in these figures for the sake of comparison, these involve no module shifts. In each of the two figures, the use of static load balancing does play a notable role in increasing performance and decreasing the frequency with which modules have to be shifted. On each curve in Fig. 6 and Fig. 7 both the cost and performance were strictly decreasing functions of the safety factor used. As the safety factor increases, modules are moved increasingly frequently and the performance and the overhead cost both increase. If the time required to transfer a module is equal to the time required to advance a module one step, the ratio of the time spent in computation to the time transferring modules in Fig. 6 and Fig. 7 is simply the average number of module shifts per step per processor divided by six. This ratio ranged from 0.005 to .0045 in the plots depicted in Figures 6 and 7.

The number of steps advanced are varied and the performance and the overhead in modules moved per step are depicted for the dynamic load balancing method in Fig. 8 and Fig. 9 respectively. In both figures, the effects of using the two static load balancing methods as well as using no load balancing at the beginning of the computation are compared. In all cases, the performance increases with the number of steps advanced.

For problems that do not require a large number of steps, the performance obtained by starting out with a static load balancing method is superior to that arising from the dynamic load balancing method without initial static load balancing. Perhaps somewhat counter-intuitively, initially balancing load with binary dissection leads to better performance than initially performing an optimal balance for problems requiring over 10 steps. The optimal static load balance is not necessarily the initial load distribution that best allows the dynamic load balancing method to move modules so that processor idleness is avoided. As the number of steps increases, the performance differences obtained through the use of different initial load distributions becomes less marked.

The initial use of static load balancing also leads to marked reduction in module transfer overhead as depicted in Fig. 9. In this figure the overhead per step generally increases with the number of steps. For problems with very large numbers of steps, the overheads for the initial load distributions all approach a single value. When no initial static load balancing is used in a problem that is advanced a small number of steps, both low performance and relatively high costs in number of modules transferred are incurred. It is noted that in Figure 9, when of initial static load balancing was not used, the number of modules transferred reaches a local maximum for problems of 10 steps, and then declines briefly before resuming its long term increase. This phenomena has been observed in a number of similar experiments, its cause is unclear.

The performance obtained through the use of binary dissection as a static load balancing method was notably poorer than that produced by the optimal balance. We have observed in these and other experiments that initial static load balancing used along with the predictive dynamic load balancing method improves performance and reduces the frequency with which modules must be moved. The choice of method used to initially balance load does not appear to have a marked impact on performance or cost.

## 8. Conclusions

The experimental results presented here revealed that the predictive dynamic load balancing method led to processor utilizations that were consistently above those obtained by the optimal static load balancing method, when the delays caused by the overhead of transferring modules are negligable. As one would expect, the optimal static load balancing method, in turn, consistently out performed the binary dissection method.

The initial partitioning of load at the point dynamic load balancing was initiated proved to have a marked effect on the performance of the dynamic load balancing algorithm. All three static load balancing methods used in conjunction with the dynamic load balancing method lead to a substantial improvement in performance and a decrease in module transfer overhead when compared to a uniform initialization of modules to processors. The magnitude of these effects depended on the number of steps the problem is advanced, being most pronounced when a problem is finished after relatively few steps. It is interesting to note that the binary dissection algorithm appeared under some circumstances to consistently lead to results that were superior to optimal load balancing when used in conjunction with dynamic load balancing.

One of the principal costs of the predictive dynamic load balancing method is expected to be the machine dependent cost of transferring the computational modules between processors. The effect of initial load distribution on this cost was examined and it was found that the frequency with which modules were transferred between processors was markedly reduced when either form of static load balancing was initially employed.

The initial distribution of load in a multiprocessor system is clearly an important determinant of the performance gains achievable by the dynamic load balancing policy; this initial distribution also has a strong influence on the overhead costs of the dynamic policy.

The four load balancing methods discussed in this paper each have their own distinct advantages and disadvantages. Finding an optimal static load balancing is in general an NP-Complete problem unless special structure is present to permit a low order polynomial solution. For the test problems that we have considered, the greedy algorithm was an order of magnitude faster than the optimal load balancing algorithm and it provided results as good as the optimal solutions. The binary dissection method and the predictive dynamic load balancing algorithms are both quite useful in situations in which low order polynomial solutions to the optimal static load balancing problem do not appear to be available.

## 9. References

[1] M. Garey and D. Johnson, *Computers and Intractability,* W. H. Freeman and Company, 1979.

[2] S. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing," ICASE Report No. 85-54, November, 1985.

[3] M. Berger and S. Bokhari, "A Partitioning Strategy for PDES across Multiprocessors," in *Proc. 1985 International Conference on Parallel Processing,* pp. 166-170, August, 1985.

[4] M. Berger and S. Bokhari, "A Partitioning Strategy for Non-Uniform Problems on Multiprocessors," ICASE Report No. 85-55, November, 1985, to appear in IEEE Trans. Computers.

[5] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computers Science Press, 1978.

[6] J. H. Saltz, "Parallel and adaptive algorithms for problems in scientific and medical programming," Ph.D. Thesis, Dept. of Computer Science, Duke University, 1985.

[7] R. Richtmyer and K. Morton, *Difference Methods for Initial-Value Problems,* Interscience Publishers, 1967.

[8] D. Nicol and P. Reynolds, Jr., "The Automated Partitioning of Simulations for Parallel Execution," Tech. Report TR-85-15, Dept. of Computer Science, University of Virginia, 1985.

[9] R. Varga, *Matrix Iterative Analysis,* Prentice-Hall, 1962.

[10] L. Adams and H. Jordon, "Is SOR Color Blind?," ICASE Report No. 84-14, May, 1984.

[11] William D. Gropp, "Local Uniform Mesh Refinement on Loosely-Coupled Parallel Processors," Technical Report YALE/DCS/RR-352, Yale University, Department of Computer Science, December 1984.

[12] G. Bolch, F. Hofman, B. Hoppe, H.J. Kolb, C.U. Linster, R. Polzer, W. Schussler, G. Wackersreuther and F. X. Wurm, "A multiprocessor system for simulating data transmission systems (MUPSI)," *Microprocessing and Microprogramminng,* vol. 12, pp. 267-277,1983.

[13] B. Kernighan, "Optimal Sequential Partitions of Graphs," *J.A.C.M.,* Vol. 18, No. 1, pp. 34-40 , January 1971.

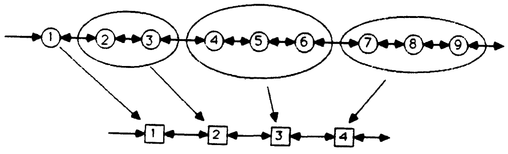


Fig. 1 A 9 module chain mapped onto a 4 processor chain.

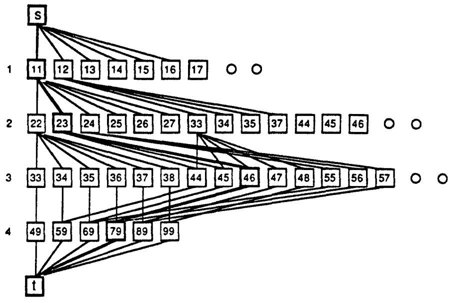


Fig. 2 The layered graph for a problem with 9 modules and 4 processors.

2 | 6 2 | 2 1 1 | 2 2 2

Fig. 3(a) A 9 module chain, with each module represented by its execution cost, mapped onto a 4 processor chain using the Binary Dissection method. The load on the most heavily loaded processor is 8 units.

2 | 6 | 2 2 1 1 | 2 2 2

Fig. 3(b) Under an optimal mapping of the 9 module chain on the processor chain, the load on the most heavily loaded processor would only be 6.

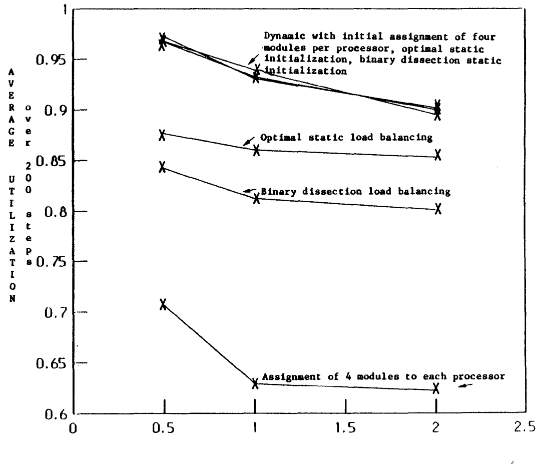


**Figure 4.** 16 processors, 64 modules, each trial run for 200 steps. Module weights drawn from truncated normal distribution with unit mean and standard deviations of either 0.5, 1.0, or 2.0, prior to truncation.

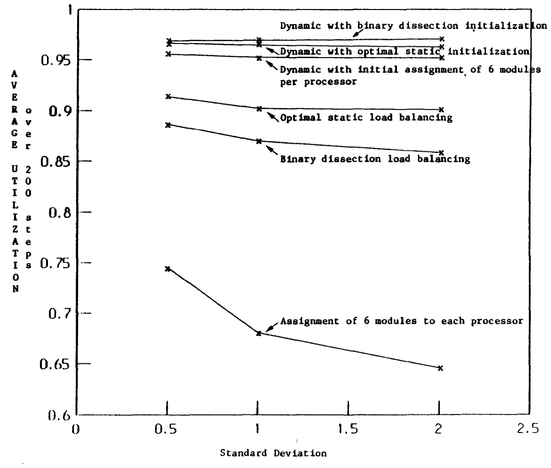


**Figure 5.** 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal distribution with unit mean and standard deviations of either 0.5, 1.0, or 2.0, prior to truncation.

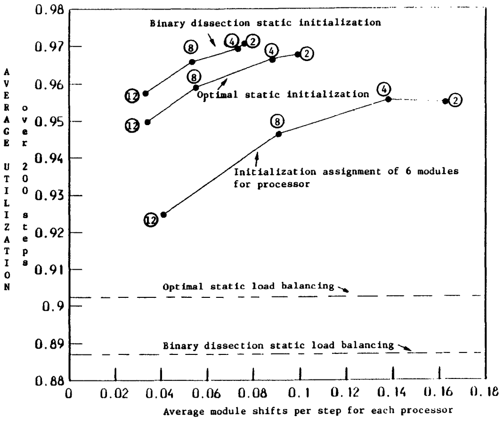


**Figure 6.** 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 0.5. Circled figures represent safety factors. Optimal static load balancing and binary dissection static load balancing performance for the same input data included for purposes of comparison.

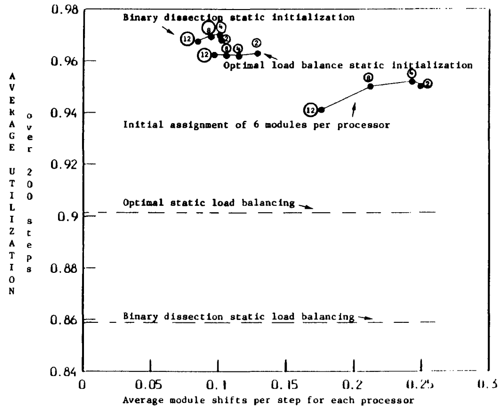


**Figure 7.** 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal with unit mean; standard deviation of normal distribution is 2.0. Circled figures represent safety factors. Optimal static load balancing and binary dissection static load balancing performance for the same input data included for purposes of comparison.

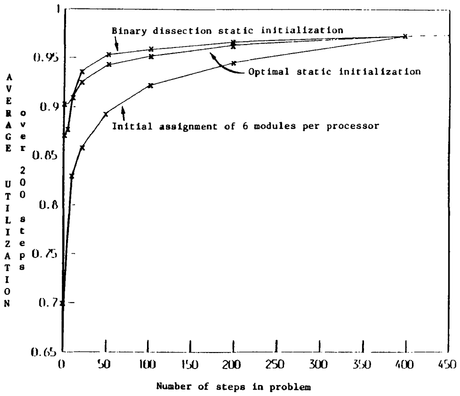


**Figure 8.** 16 processors, 96 modules, each trial run for 5, 10, 20, 50, 100, 200, 400, 800 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 1.0.

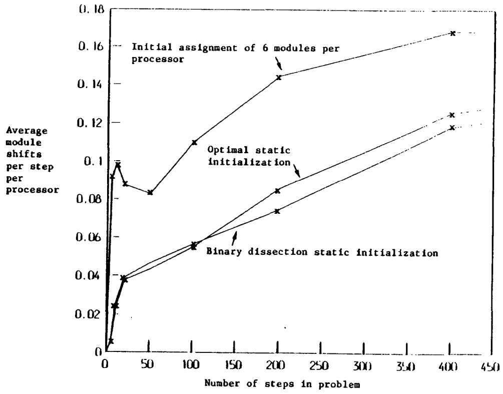


**Figure 9.** 16 processors, 96 modules, each trial run for 5, 10, 20, 50, 100, 200, 400, 800 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 1.0.

# PARALLEL PROCESSING IN LOCAL NETWORKS*

**G. C. Shoja** and **R. G. Gurr**

Department of Computer Science
University of Victoria
Victoria, B.C., Canada V8W 2Y2

**Abstract**

High level tasking constructs and rendezvous mechanism are employed to experiment with parallel processing of program components on nodes of a local network. In contrast to instruction level synchronization requirements of SIMD machines, our study centers around *interaction point* level synchronization in a distributed MIMD machine. Program decomposition issues are discussed and a strategy for allocation of tasks to optimum number of processing stations together with experimental performance results are presented.

## 1. Introduction

Local area networks have generally been used for connecting a number of computing modules within a limited geographical space. The primary objective has been to enable users on different hosts to communicate and share access to expensive peripheral devices and remote disc files. There has also been much work on distributed processing aspects of local networks, with emphasis on intertask communication primitives and remote call implementation [1,2,3]. In this paper we address parallel processing potentials of local area networks.

Parallel processing within a local network is attractive for two main reasons. First, it enables utilization of the spare processing capacity of the increasingly powerful personal workstations by distributing the computational intensive jobs among them. Secondly, a fast local network can be used as an inexpensive way of connecting processors in a stand-alone multiprocessor computer. The basic issues of parallel processing in a local network include: the choice of the programming language, suitable granularity of parallelism, and especially program partitioning and assignment strategies.

In the following sections we will discuss the above issues and present the results of our experiments with task level parallelism. The experiments were carried out to measure effects of the frequency of *interaction points* (rendezvous) between constituent tasks modules, as well as the number of processing stations employed, on the execution time of a given job. Finally we present a heuristic method which can be used to estimate the optimum number of workstations which should be used for parallel execution of components of a given job.

## 2. System Support Components

Since parallelism at task level was our main objective, a high-level language with concurrent tasking facilities was the logical choice. The programming language used in this project is Martlet [4] which is a Pascal derivative with concurrent tasking facilities added on. The details of Martlet and its implementation have been given in references [5,6]. Briefly, a Martlet program generally consists of a number of cooperating tasks which communicate using the Entry_Call Accept rendezvous mechanism similar to Ada [7]. The compiled tasks are assigned to the processing stations by means of a system generation program called *SETUP* which creates the final core images for the processing stations. The criteria for allocation of tasks to processing stations will be discussed in the section dealing with program decomposition. Martlet has proven very suitable for distributed and parallel programming applications and was ported to a network of PC/XTs for this project [8]. An example of a Martlet program is given in the Appendix I.

The target network consists of a number of workstations (PC/XT/AT's) connected by 10 Mbps Ethernet [9] as shown in Figure 1. Martlet's run-time support system consists of a distributed multitasking kernel and an interpreter with copy of each in every station of the local network. The station kernel is responsible for managing the processor and the tasks

*This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

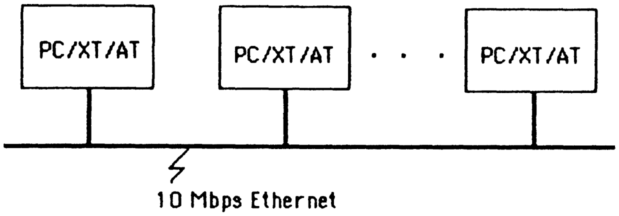


**Figure 1 The Local Network Configuration**

in the station. Interstation communication between tasks is handled transparently by pseudo transport level tasks which are created at system SETUP time.

## 3. Program Decomposition

The problem of decomposing a large program into concurrently executable sub-tasks has been studied for some time [10,11]. Partitioning (i.e. division of a program into procedures, modules, and tasks) together with assignment which deals with allocation of these units to the available processing units, are the two most important issues in distributed and parallel processing systems. In a network or distributed environment, the actual placement of the computational objects has substantial effect on the overall performance of the system. This is mainly because of additional communication delays when communicating tasks are located in different stations of a local network [3].

Minimizing interprocessor communications as well as load balancing are the two main criteria for allocating tasks to the available processors in a system. Exact partitioning of a set of communicating tasks into a distributed multiple processor system is an np-complete problem. However, heuristic algorithms have been developed for optimum allocation of computational objects in a distributed system. These algorithms generally define a graphical model of the communications structure of the tasks, then use a heuristic approach to 'cut' the graph into distinct components such that the sum of the weights (representing the frequency of communications) of the edges in the resulting edgecut is a minimum [12].

However, we have to be cautious when trying to apply the above allocation algorithms in a local network because these algorithms assume: (1) the cost of in-station communications between tasks to be zero and (2) the cost per inter-station communication to be the same for all configurations. Our experimental results show that both of these assumptions are far from being acceptable when using high level intertask communication constructs in a broadcast type local network. To achieve higher execution speed we must distribute the load among additional stations. The question is how many stations should be employed before the benefits of additional processing power are offset by the increased communication delays. We try to answer this question in the next section.

## 4. The Task Allocation Method

Here, we are concerned with the class of problems which can be implemented using task level parallelism. Of particular interest are ***synchronized algorithms*** where parallel tasks must exchange data and synchronize their operations at some ***interaction points.*** As an example, a number of producer tasks may perform many iterations of a simple function and then communicate the results to a consumer task at specific points within the computation. Therefore we are dealing with a MIMD environment where operations are synchronized after some intervals; this is in contrast to a SIMD environment where instruction level synchronization of the processing units is necessary.

We now present the heuristic allocation strategy which was developed for the class of problems which can be translated into a set of producers-consumer tasks as stated above. The strategy is based on the simple guideline that the actual processing speed-up due to an additional workstation should be at least equal to or greater than the corresponding communication overhead which must be handled by the new station. To find the formula for estimating the optimum number of stations we define:

Ts = Total execution time for a job when all component tasks are assigned to a single station.

Tr = Average delay overhead of a network rendezvous (message exchange).

n = Total Number of Rendezvous during the execution of the entire job.

N = Maximum number of stations to be used for a given job.

Assuming balanced load distribution, then for an additional station to have a positive effect we should have:

$$\frac{Ts-nTr}{N-1} - \frac{Ts-nTr}{N} \geq \frac{nTr}{N}$$

Therefore:

$$N \leq \frac{Ts}{nTr}$$

Ts can easily be obtained from the single station execution of a job in the class of problem concerned. Even Tr can also be estimated from communications delay overheads when all tasks are run in a single station (see Figure 3). Once N is estimated from the formula, the component tasks are divided evenly among N stations. Figures 2(a) and 2(b) support the validity of such a strategy as discussed above.

## 5. Experiments and Results

The experiments were conducted for configurations which ranged from a single station to 9 stations in the local network. The entire job consists of one consumer and eight instances of a function (producer) task as given in Appendix I. Each producer performs an identical amount of computation and returns a single result to the consumer. The consumer then computes the sum of eight results. Each function is a simple iterative loop from 1 to 100. At run-time, each function is given a parameter called Force = 1..200 telling the function how many 1..100 iterations it should perform before returning a value. The entire system is given a parameter called WORK = 1..10, telling the system how many times the consumer should calculate the sum. WORK determines the number of rendezvous during the test. Figures 2(a) and 2(b) show the speed-up in execution time as the number of stations is increased. The total workload and the number of rendezvous are used as parameters.

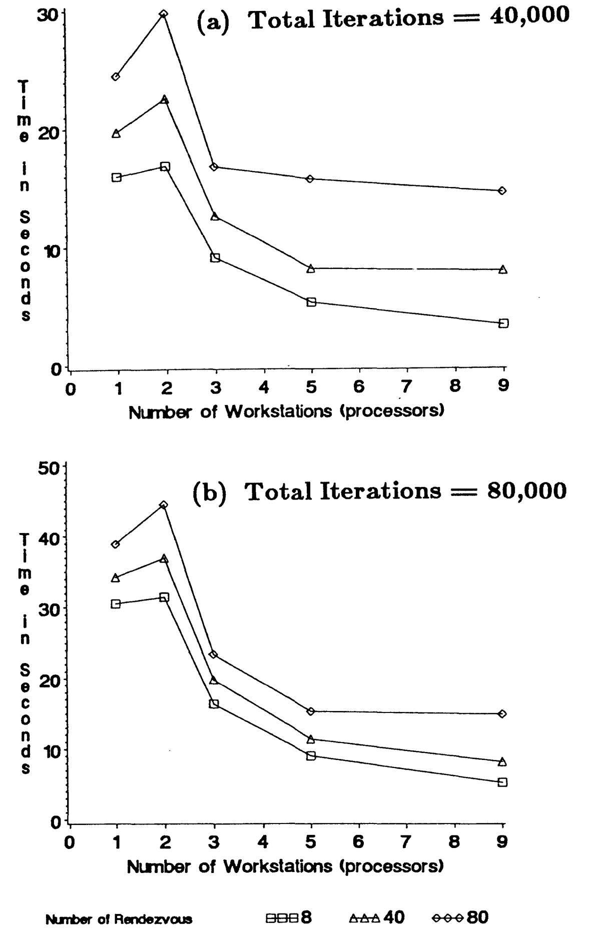


**Figure 2 Execution Time vs No. of Workstations**

Figure 3 gives the variation in rendezvous delay overhead as a function of the number of stations taking part in execution of the job. We should point out here that due to the interpretative nature of the run-time system the execution and delay figures are larger than normally encountered in non-interpretive systems. However, this does not effect the relative nature of our comparisons.

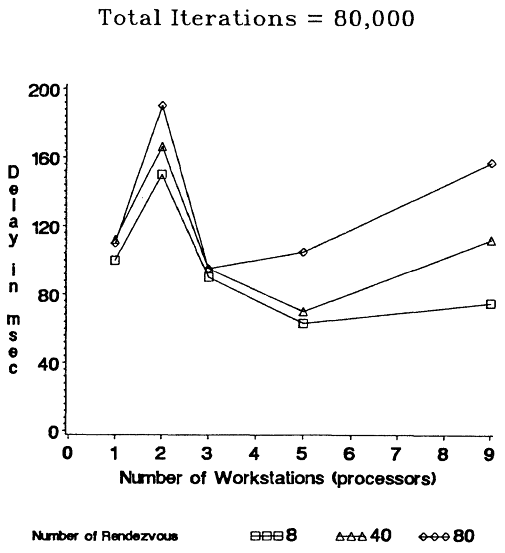


**Figure 3 Variations in Rendezvous Delay**

Table I shows the maximum number of stations which should be employed for different workloads and frequency of rendezvous as calculated using our proposed allocation method. These numbers are in close agreement with the actual number of stations at which execution speed-up levels off, as shown in Figures 2(a) and 2(b).

**Table I Optimum No. of Workstations for a Given Job**

| No. of Iterations | Execution time on a single Station(Ts) | No. of Rendezvous (n) | Optimum No. of workstations using N = Ts/nTr and load balancing |
|---|---|---|---|
| 40,000 | 20.5 (sec) | 40 | 5 |
| 40,000 | 24.5 | 80 | 3 |
| 80,000 | 34.8 | 40 | 9 |
| 80,000 | 39.2 | 80 | 5 |

## 6. Conclusions

We have shown that a particular class of problems which require *interaction point* level synchronization similar to the given test problem, are suitable for parallel processing on stations of a local network. We have also presented a heuristic method which uses the data from a single station execution of a set of cooperating tasks to estimate the optimum number of stations that should be used for their parallel execution.

Further work includes finding methods by which a general computation intensive problem can be decomposed into a set of producers-consumer tasks which can then be handled by the method described in this paper.

## 7. References

[1] Walker, B. *et. al.* , "The LOCUS Distributed Operating System", ACM Operating Systems Review, Vol. 17, No. 5, 1983, pp. 49-70.

[2] Birrell, A.D. and B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol.2, No.1, Feb. 1984, pp. 39-59.

[3] Cheriton, D.R. and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", ACM Operating Systems Review, Vol. 17, No. 5, 1983, pp. 129-140.

[4] Grimsdale, R.L., Halsall, F., Martin-Polo, F., and Shoja, G.C., "Martlet: A programming language for a distributed multiple microprocessor system", ICS Conference Proc.,London March 1981, pp. 403-414.

[5] Halsall, F., Grimsdale, R.L., Shoja, G.C., Lambert, J.E., "Development environment for the design and test of applications software for a distributed multiprocessor computer system", IEE Proc., 1983 130, (1), pp. 25-31.

[6] Faci, M. and Shoja, G.C., "A Distributed Kernel for Support of Transparent Communication Between Tasks", in Proc. of Fifth Phoenix Conference on Computers and Communications, March 26-28, 1986, pp. 625-631.

[7] Ichbiah, J. *et. al.*, "Rationale for the design of the Ada programming language", SIGPLAN Not., 1979, Vol.14, No.6.

[8] Gurr, R.G. and Shoja, G.C., 'Experience with porting a multitasking system to a local network of IBM PCs', University of Victoria, Dept. of Computer Science, Technical Report No. DCS-51-IR, Oct. 1985.

[9] Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed packet switching for local computer networks", Commun. ACM, Vol.19, No. 7, July 1976, pp. 395-404.

[10] Arvind, "Decomposing a Program for Multiple Processor Systems", Proc. of International Conference on Parallel Processing, Aug. 1980, pp. 7-14.

[11] Kruskal, C.P. and Weiss A., "Allocating Independent Subtasks on Parallel Processors", IEEE Trans. on Software Engineering, Vol., SE-11, No. 10, Oct. 1985, pp. 1001-1015.

[12] Jenny, C.J., "On Allocation of Computational Objects in Distributed Systems", IBM Research Report RZ 1123, IBM Zurich Research Laboratory, January 18, 1982.

## APPENDIX I : Test Programs Written in Martlet

```
(* Consumer *)

task consumer;
  type coeff = integer;
  entry sync[0..7](var w: integer; var f: integer);
  entry fn[0..7] (x: coeff);
close;

task body consumer;
var
    WORK, FORCE, w: integer;
    a, b, c, d, e, f, g, h: coeff;
    results: array [1..200] of integer;
body

(* Synchronize tasks & pass them work assignments *)
  accept sync[0](var w: integer; var f: integer)
    then begin w := WORK; f := FORCE;
      accept sync[1](var w: integer; var f: integer)
        then begin w := WORK; f := FORCE;

            . . .
            accept sync[7](var w: integer; var f: integer)
              then begin w := WORK; f := FORCE
              end
                . . .
      end
  end;

(* Collect the coefficients *)
  for w := 1 to WORK do
    begin
      accept fn[0] (x: coeff) then a := x;
      accept fn[1] (x: coeff) then b := x;
         . . .
      accept fn[7] (x: coeff) then h := x;

        (* The coefficients have been computed  *)
        (* The result can be calculated & saved *)

      results[w] := (a + b + c + d + e + f + g + h) * w;
    end;
  close;

(* Producers *)

task producers[0..7];
close;

task body producers;
use consumer;
var
    WORK, FORCE, w,f,i,j: integer;

body

  sync[myinstance](WORK, FORCE);   (* Synchronize *)
  for w := 1 to WORK do
    begin
      for f := 1 to FORCE do
        begin
          i := 0;
          for j := 1 to 100 do       (* 100X the FORCE value  *)
             i := succ(i)  (* keep busy within loop *)
        end;
    (* Return producer instance as value *)
      fn[myinstance](myinstance)
    end
close;
```