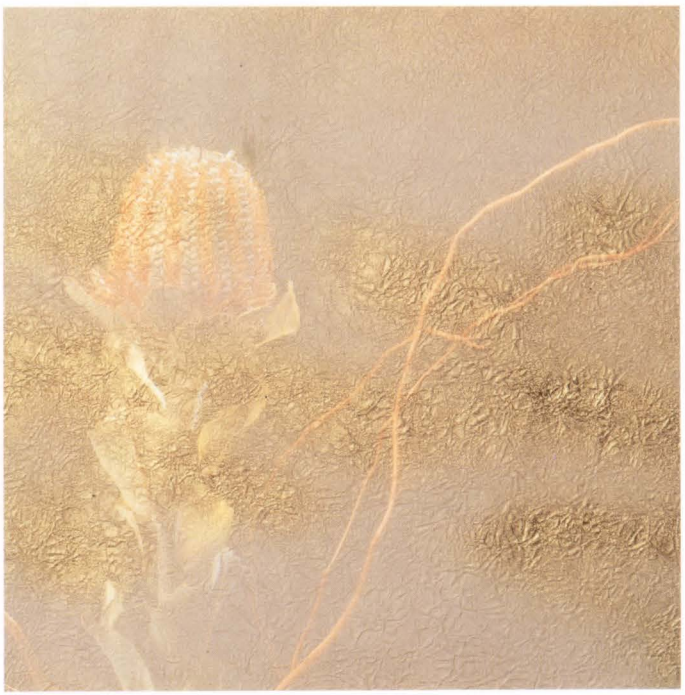
ISSN 0016-2523

FUJITSU

SCIENTIFIC & TECHNICAL JOURNAL

Spring 1993 VOL. 29, NO. 1

Special Issue on Cellular Array Processor AP1000



FUJITSU Sci. Tech. J., **29**, 1, pp. 1-117, Kawasaki, Spring, 1993

The Issue's Cover :

萌芽 (Germination)

FUJITSU Scientific & Technical Journal is published quarterly by FUJITSU LIMITED of Japan to report the results of research conducted by FUJITSU LIMITED, FUJITSU LABORATORIES LTD., and their associated companies in communications, electronics, and related fields. It is the publisher's intent that FSTJ will promote the international exchange of such information, and we encourage the distribution of FSTJ on an exchange basis. All correspondence concerning the exchange of periodicals should be addressed to the editor.

FSTJ can be purchased through KINOKUNIYA COMPANY LTD., 38-1, Sakuragaoka 5-Chome, Setagaya-ku, Tokyo 156, Japan, (Telephone : +81-3-3439-0162, Facsimile : +81-3-3706-7479).

The price is US$7.00 per copy, excluding postage.

FUJITSU LIMITED 1015 Kamikodanaka, Nakahara-ku,
Kawasaki 211, Japan
Cable Address : FUJITSULIMITED KAWASAKI
Telephone : +81-44-777-1111
Facsimile : +81-44-754-3562

Printed by MIZUNO PRITECH Co., Ltd. in Japan

FUJITSU

SCIENTIFIC & TECHNICAL JOURNAL

Spring 1993 VOL. 29, NO. 1

Special Issue on Cellular Array Processor AP1000

CONTENTS

Featuring Papers

Preface

Special Issue on Cellular Array Processor AP1000

Shigeru Sato

Shigeru Sato
Managing Director
FUJITSU LABORATORIES LTD.

The dynamic nature of current research into parallel processing—a technique where hundreds to thousands processors do computation in parallel—comes in part from the realization that single-processor computing is nearing its limit and in part from the need for high-peformance computers in a growing number of new scientific areas such as quantum chromodynamics (QCD), climatological simulation, and human genome analysis.

The research proceeds actively throughout the world: Japan's Ministry of Education, for example, is sponsoring a massively parallel computing research project participated in by leading universities in Japan. A number of programs on parallel processing and its applications in computational science, neural network, physics, molecular science, and earth science are now under way at the Edinburgh Parallel Computing Center in Scotland. In the US, research proceeds for high-performance computers and high-speed networks to access them from organizations around the country.

The idea of parallel processing is not new, of course, and has been approached in different ways since the computer's invention. Recent advances in semiconductor technology have made possible parallel computers that uses hundreds to thousands processors. Such parallel computers have been marketed already, and the latter half of the 1990s is expected to see the advent of several TFLOPS machines.

This special issues of the Fujitsu Scientific & Technical Journal focuses on the AP1000, a highly parallel MIMD computer developed at Fujitsu Laboratories Ltd. Fujitsu Laboratories' first project in parallel computers in 1983 dealt with cellular array processors, including CAP-256 developed in 1987. The AP1000 was developed in 1990, and has an architecture different from that of the CAP-256. The AP1000 system went into commercialization in 1992 as a platform for parallel processing research.

The first article reviews parallel processing. The next four articles discuss the AP1000's architecture, performance evaluation, and programming environment. Because increasing the number of processor elements used in a parallel processing tends to lower efficiency due to communication overhead and load inbalances, fast communication is extremely important. The AP1000's software environment implements a debugging utility, a run-time monitor, and a

performance analyzer for developing and tuning parallel programs and to maximize computation potential. The next two articles explain visualization on the AP1000—an important means for understanding the simulation results for huge amounts of data, and describe a video/disk option board connected to each processor to enhance computer graphics and file I/O performance, and volume visualization techniques for 3-dimensional data.

The next five paper articles deal with parallel processing applications in linear algebra, electronics CAD and computational physics — BLAS-3 and LINPACK, parallel circuit simulation, LSI mask pattern generation, QCD simulation, and the application to data analysis in high-energy physics experiments.

The last article introduces Fujitsu Parallel Computing Research Facilities which makes parallel computing environments available to researchers to promote parallel computing R & D.

In this special issue, three articles from the Australian National University (ANU) are included, which are concerned with Fujitsu-ANU collaborative reaearch started in 1989.

The parallel processing is a key technology in 90's and it will open a new programming paradigm. But in order for unfamiliar people to use the technology widely, R & D into software development tools and parallel algorithms are necessary. Fujitsu is continually improving the technology and creating an advanced computer systems.

UDC 681.322

Advantages of Massively Parallel Processors

● Mitsuo Ishii

(Manuscript received October2, 1992)

Massively parallel processors are cost-effective and have many other attractive features; for example, high linear-speed, suitability in a wide range of applications, ability to solve large-scale problems, high-reliability, and a heterogeneous structure. These features are expected to be more fully exploited in the next decade. This paper looks at some of the features of massively parallel processors.

1. Introduction

Progress in large-scale integration and the development of the microprocessor have opened the way to inexpensive and powerful computing systems. Originally, massively parallel processor used single-bit computing elements, now nearly all such processors use 32-bit general-purpose microprocessors. These microprocessors include high-speed RISC microprocessors, floating-point processors, and dedicated communication chips. Inexpensive microprocessors with built-in vector processors, which have been available since 1990, upgrade microprocessor performance to the level of pipeline supercomputers[1]. These systems have been made possible by the development of highly-integrated chips. However, the development of applications for these advances has been slow, with the result that massively parallel processors are not yet as popular as they should be.

This paper outlines some of the advantages of these massively parallel processors in an effort to help remedy the above situation.

2. Potential advantages

2.1 Increased linear speed

Theoretically, the processing speed of the massively parallel processor is proportional to the number of processors being used. For example, one thousand processing elements, each having the ability to perform 10 MFLOPS, can theoretically achieve 10 GFLOPS. The total processing speed can be increased without being restricted by semiconductor device speeds. This means that a parallel processor using conventional technology has potentiality to process data faster than a single processor that uses the most advanced semiconductor technology.

In practice, however, the speed of a parallel processor does not increase linearly with the number of processors. Overcoming this difficulty will require, at the least, a parallel algorithm that distributes loads uniformly among processors and a high-throughput, low-latency interprocessor communication architecture.

2.2 Diverse applications

By looking at the characteristics of application programs (see Fig. 1), we can see that the multi microprocessor excels in particle simulation and image-processing programs consisting of many independent tasks. For example, by using ray tracing techniques that generate real images, a parallel processor can simultaneously process all the picture elements of a screen. Task programs are suitable for execution on a scalar computer because they consist mainly of IF statements and serial programming logic. Because of the above, we can expect an increase in the number of application programs written for massively parallel processing environments.

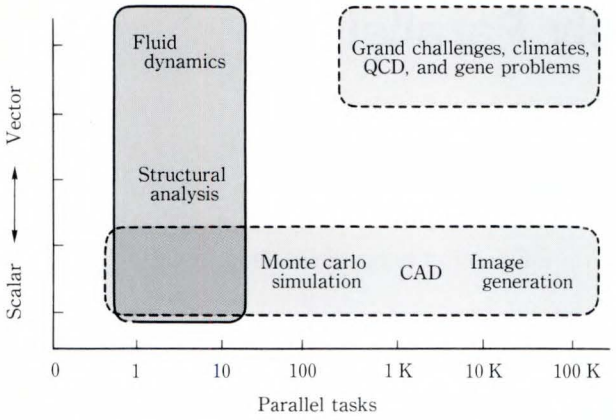


Fig. 1 – Characteristics of application programs.

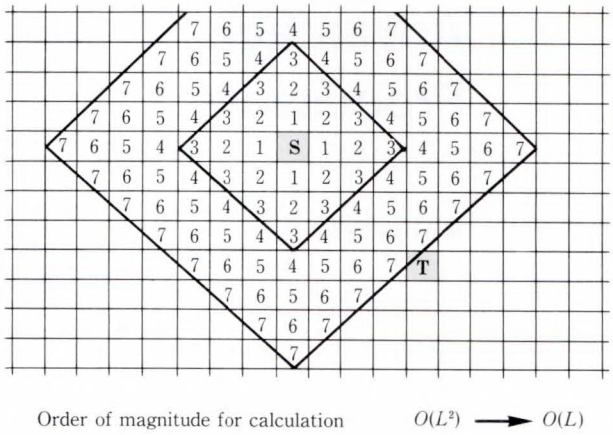


Fig. 2 – Parallel processing for path search.

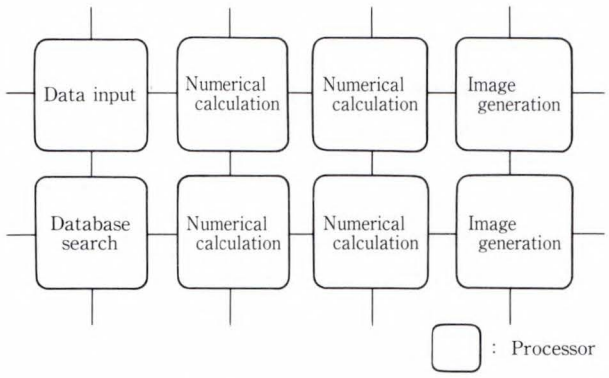


Fig. 3 – Heterogeneous system.

2.3 Large-scale problems

The amount of calculation that must be carried out to solve a problem tends to increase with the scale of the problem. With combinational problems, the amount of calculation increases exponentially with the number of parameters in the problem. For example, the calculation time for circuit analysis problems is proportional to the number of transistors to the power of 1.5. This means that large-scale problems often take too much time to solve.

A good parallel algorithm, however, can greatly reduce the calculation time. For example, the simple task of determining the shortest path from points **S** to **T** on the two-dimensional mesh shown in Fig. 2 requires 64 (8×8) serial operations on a serial processor but only 8 on a parallel processor[2]. That is, the calculation time is reduced from the second power to the first power of the distance between the two points. Therefore, the parallel processor is suited to large-scale problems that take too much time to solve on a serial processor.

2.4 Heterogeneous systems

Massively parallel processors use a number of different processing schemes. In the single-program multiple-data scheme, each processor executes the same program. This scheme can be applied to the large scale numerical calculations, as the simulation of fluid dynamics. On the other hand, in the multiple-program multiple-data scheme, each processor executes a different program. In one example of a multiple-program, one processor receives data from an external device, retrieves data from internal files, another performs numerical calculations, and the remainder generate images (see Fig. 3). Each processor transfers its output to the next processor via the network.

Programs are easy to write for this multiple-program scheme even in a complicated heterogeneous system because different processes are assigned to different processors. Besides that, each process can be carried out very efficiently by determining the number of processors required for the processing load.

2.5 Highly reliable systems

The reliability of the massively parallel processor is generally lower than that of a single processing element by a factor equal to the number of processors.

However, reliability can be improved by

using built-in redundancy. In this method, the processors are backed up by secondary processors that execute the same programs off-line. If an on-line processor fails, it is automatically replaced by its secondary processor without the need to stop the system.

3. Conclusion

This paper has discussed some of the features of the massively parallel processor and its potential advantages over serial processors. The massively parallel processor is still in its infancy, and we can be sure that it will be used in many more applications in the near future.

Mitsuo Ishii

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Engineering
Nagoya Institute of Technology 1966
Dr. of Engineering
Tokyo Institute of Technology 1988
Specializing in Image Processing,
CAD and Parallel Processing

References

1) Hot Chips IV Symposium Record. Stanford University, Stanford, CA, 1992.
2) Nair, R., Hong, S. J., Liles, S., and Villani, R.: Global Wiring on a Wire Routing Machine. Proc. 19th Design Automation Conf., 1982, pp 224-231.

UDC 621.391:681.322

Architecture for the AP1000 Highly Parallel Computer

● Hiroaki Ishihata ● Takeshi Horie ● Toshiyuki Shimizu

(Manuscript received August 11, 1992)

This paper describes an architecture and its implementation for the AP1000 highly parallel computer which consists of 64 to 1 024 processing elements and three independent communication networks. A high throughput, low latency communication network and hardware support for message handling, data distribution/collection, and barrier synchronization are the key technologies in high-performance parallel computing. A new routing scheme developed for the AP1000's torus topology network reduces latency, avoids deadlock, and achieves high throughput. The message controller (MSC) on each processing element reduces message handling overheads for data transfer/reception setup. Dedicated communication networks between processor elements and host reduce overheads for data distribution and collection, and for barrier synchronization.

1. Introduction

Many distributed memory parallel processors (DMPP) have been proposed. The Cosmic Cube[1], Mark IIIfp[2], and J-Machine[3] have been developed for research purposes, and the iPSC[4], nCube, and Transputer have been developed as commercial products. In a larger parallel computer, the DMPP is easily expandable and is cost effective. However, it is not easy to obtain good results for many applications. Some algorithms require a lot of parallel communication when there are many processing elements. To run a program efficiently, communication time should be minimized. Minimizing communication latency and widening communication throughput are critical issues in DMPP development.

The first generation DMPPs used store and forward message passing[1,5]. In every node on the routing path, the message was copied to memory by a Direct Memory Access (DMA) controller, then re-transmitted toward the destination. The communication latency time was $O(N \times D)$, where N is the message length and D is the distance. It usually took several milliseconds, making software overheads a minor concern. To reduce the routing overhead at routing nodes, various small diameter networks were studied.

The second generation DMPPs employed message-routing hardware to reduce network latency[6,7]. Some new routing schemes were proposed, such as wormhole and cut-through routing. In wormhole routing, for example, the messages moving through the network are viewed as worms which travel from the source to destination processors. These developments in communication networks have reduced network latency time to $O(N + D)$. Since N is usually greater than D, path length is no longer a significant problem.

Unfortunately, large message handling overheads prevented optimal use of low latency networks. Conventional DMPPs had poor performance for communicating small messages, because it usually took more than several tens of microseconds for message handling such as message assembly/disassembly and interrupt handling. Most communication time is used for message handling. As the grain size of parallel computers decreases, message size becomes small and communication setup time becomes

more important.

Message handling overheads also exist for communication between processing elements and the host computer. In data parallel programs, data must be distributed from the host computer to processing elements before calculation. After calculation, results dispersed in processing elements must be collected by the host computer. In both cases, the host computer must interact with many processing elements. The overhead for data transfer setup in the host computer increases in proportion to the number of processing elements. The problem becomes more serious for systems with several hundred processing elements.

The same goes for the large overheads involved in barrier synchronization and broadcasting. In conventional DMPPs, these communication functions were emulated by passing several point-to-point messages. Although they are used frequently in parallel programs, many machines lack the hardware to perform them efficiently. This makes synchronization and broadcasting very expensive.

This paper is organized as follows: Chapter 2 presents the AP1000 design concepts. Chapter 3 describes the system and processing element configuration in detail. Chapter 4 describes the message controller which supports fast message handling. Chapter 5 explains the configuration of the three networks in the AP1000.

2. Design concepts

This system incorporates the following design concepts:

1) High throughput, low latency communication network

According to recent studies, wormhole routing on low-dimensional networks with wide channels provides lower latency and less contention than on high-dimensional networks with narrow channels[8]. However, simple wormhole routing has the drawback that it may cause deadlocks and reduce throughput because the channel used for message transfer is blocked during the transfer. A new routing algorithm which routes multiple messages concurrently without causing any deadlocks is needed.

2) Fast message handling

Network latency can be reduced to a few microseconds using wormhole routing. It takes much more time for message handling such as message assembly/disassembly or data transfer/reception setup for communication. To reduce the message receiving overhead, it is important to receive messages without interrupt processing in the processing elements. To reduce the message sending overhead, it is also important to utilize the cache memory which all current high-performance CPUs have.

3) Efficient data distribution and collection

For data distribution and collection, the host computer must interact with many processing elements. The time for data transfer setup and message assembly/disassembly at the host computer should not increase as the number of processing elements increases. To reduce the load on the host computer, data distribution and collection should be done in a single message transfer. Extracting data from the message for each processing element and composing the message from data in each processing element must be done automatically.

4) Fast barrier synchronization

Message passing takes a time of at least $O\{\log(p)\}$ to execute barrier synchronization in a conventional message passing computer, where p is the number of processing elements. Adding a dedicated network to the system for barrier synchronization is worthwhile, since this reduces point-to-point communication network contention in addition to reducing the barrier synchronization overhead.

To realize these concepts, we developed a schemas called *structured channel routing* on a 2D torus topology network. This scheme provides low latency message routing between arbitrary processing elements without decreasing throughput and causing deadlock. To reduce message handling overheads, we developed a message controller (MSC) and a B-net interface (BIF). The message handler in the MSC sends messages directly from cache memory and automatically receives messages in its circular buffer. The BIF supports data distribution/collection and fast barrier synchronization.

3. AP1000 architecture

3.1 System configuration

Figure 1 shows the AP1000 system configuration[9),10)] Processor elements, called cells, are connected by three independent communication networks. These are the torus network (T-net) for point-to-point communication between cells, the broadcast network (B-net) for 1-to-*N* communication in data distribution and collection, and the synchronization network (S-net) for barrier synchronization.

The B-net connects all cells and the host computer and is used for broadcasting, data distribution, and data collection. The B-net uses compound networks including hierarchical common buses and a ring network. The B-net consists of a 32-bit data path and some control signals such as reset, attention, and arbitration.

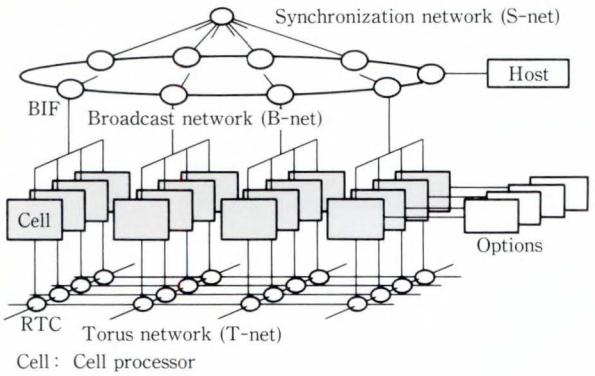


Fig. 1 – System configuration.

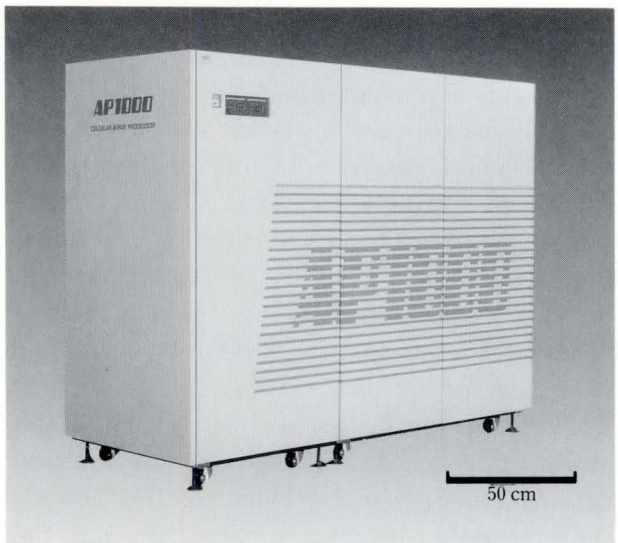


Fig. 2 – System photograph (256 cells).

The B-net uses pipelined handshaking control and has a data transfer rate of 50 Mbyte/s.

The T-net uses a two dimensional torus topology network. Each port of the T-net has a 16-bit data path, a few control signals, and a data transfer rate of 25 Mbyte/s. It also uses pipelined handshaking control.

All cells and the host computer are also connected by the S-net. The S-net has a tree topology and is used for barrier synchronization.

A Sun-4/330 workstation acts as the host computer. The host interface consists of a VME-bus interface, B-net interface, and 32 Mbytes of local memory. Figure 2 shows a photograph of a system with 256 cells. The maximum (1 024-cell) configuration uses four frames in the shape of a cross.

3.2 Cell configuration

Figure 3 shows the cell hardware configuration. Each cell has an integer unit (IU), floating point unit (FPU), a message controller (MSC), a routing controller (RTC), B-net interface (BIF), and 16 Mbytes of dynamic memory. The IU, FPU, and 128 Kbytes of cache memory are connected to the MSC. During normal operation, the MSCs work as a direct-mapped cache memory controller with a copy-back policy. The linesize is four words.

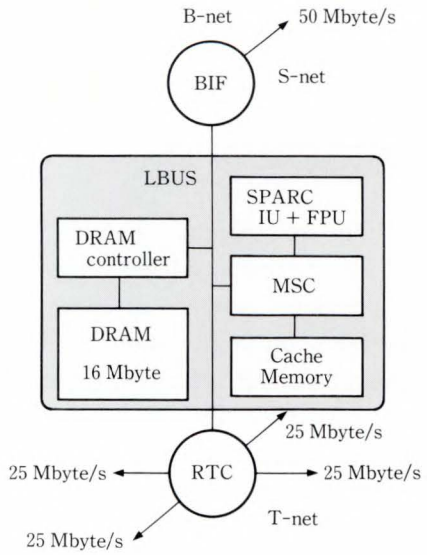


Fig. 3 – Cell configuration.

The MSC, RTC, BIF, and DRAMC in each cell are connected via the LBUS, a 32-bit synchronous bus. Each cell has an external LBUS connector. This enables the installation of various hardware options such as a high-speed I/O interface, disk interface, vector processor, and additional memory.

The DRAMC controls 16 Mbytes of memory which consists of forty 4-Mbit DRAM chips. The DRAMC detects double bit errors and corrects single bit errors. Quadruple interleaving control speeds up access to consecutive addresses. It takes 160 ns for word write access and 400 ns for 4-word block write access. Since wait cycles are inserted at read access, it takes 400 ns for word read access, and 640 ns for 4-word block read access. Thus the maximum data transfer rate from memory to device is 25 Mbyte/s and from device to memory is 40 Mbyte/s.

Figure 4 shows a wiring board; each board accommodates two cells. To reduce the number of components, complex logic is implemented on a set of four CMOS gate arrays.

4. MSC message controller

The MSC[1] consists of a cache controller, a pair of message handlers, and four channel DMA controllers. These MSC features reduce the overheads for data transfer setup and for releasing the IU from data transfer processing. Figure 5 shows the MSC configuration.

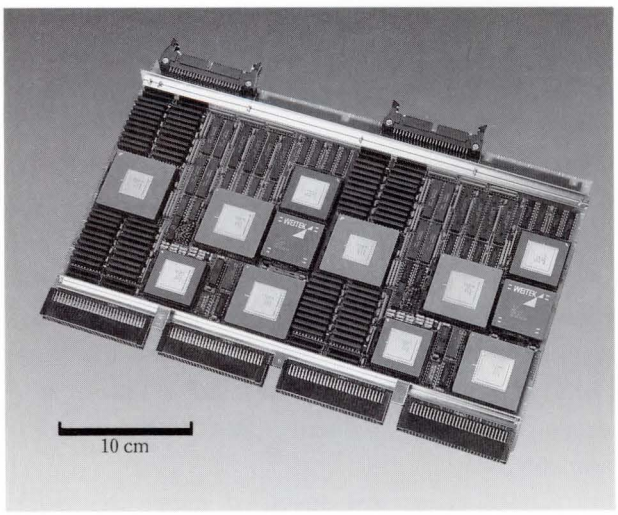


Fig. 4 – Cell wiring board.

4.1 Line sending

A line sending function sends a cache line message in a manner similar to a cache flush. The DMA is activated automatically, even when there is no specified data in the cache. The arrows in Fig. 6 indicate the data flow: HIT when there is data in the cache, and MISS when there is not. For comparison, the figure also shows a normal cache flush (FLUSH).

Line sending is initiated by a simple store instruction, similar to cache memory control. The six most significant address bits are decoded as commands. This operation is different from sending by program mode, which reads data from memory to a register and then writes data to the device. Rather than loading and sending, line sending specifies the address of data. Cache line data is sent by a store instruction.

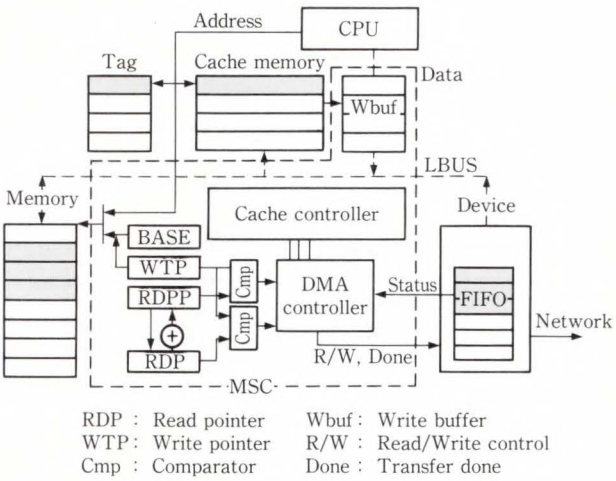


Fig. 5 – MSC configuration.

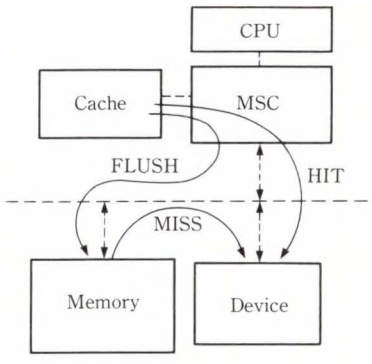


Fig. 6 – Data movement during line sending.

When a CPU initiates a line sending command, the MSC checks the tag memory to see if the data is in the cache or in main memory. It simultaneously checks the output network device status to see if the device can accept the data, whether the local bus is ready, and that no other DMA is about to write to the same network device. If all the devices are ready and there is data in the cache, that data is read and written to the Wbuf (HIT). If there is no data in the cache, the DMA to one cache line is invoked, and the data is read from main memory and written to the device (MISS).

The CPU is blocked until the status has been checked and data has been moved from the cache to Wbuf, or a DMA is initiated. The CPU does not wait for all the data to be written to the device. To implement this mechanism, the devices must have a FIFO deeper than the cache line size. Because it is not known when a whole line is sent to the network, the MSC checks the FIFO status to see whether there is room for the message.

If the devices are busy, a data access exception is reported to the CPU, and a trap occurs. This trap enables the CPU to retry line sending and avoid deadlock.

Line sending will not invalidate the cache entry or write back to main memory. Because this data may be accessed again, invalidation might lose the newest data, and flushing is expansive. Programs often modify a part of the messages and send them again.

4.2 Buffer receiving

Buffer receiving (see Fig. 7) corresponds to any asynchronous data transfer request using the ring buffer in main memory. Hardware monitors the ring buffer for overflow, notifies the CPU of such an error by an interrupt, or stops transferring until data reading produces a receive area in the buffer. The accessed area is released and returned to the MSC as a receive buffer by modifying the register.

Data movement is done in 4-word aligned blocks – in the same cache line to facilitate fast LBUS access.

The BASE, write pointer (WTP), and read pointer (RDP) are registers (see Fig. 5). The BASE holds the buffer address. A pair of pointers to the circular buffer, WTP and RDP, point to data that has been received from other processors but has not yet been read by the CPU. The WTP points to the address where the next received data is to be stored. It is updated by hardware after each data storage. The RDP points to the address that the CPU is currently accessing. The CPU updates the RDP to dispose of data which is no longer needed. The buffer size can be between 64 Kbytes and 512 Kbytes.

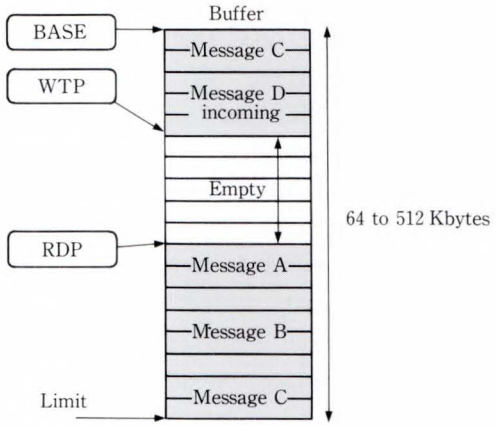


Fig. 7 – Buffer receiving.

When data arrives from a network, the MSC checks that the WTP value does not exceed the RDP contents. If WTP and RDP hold the same value, no more data can be received, and the MSC suspends transfer or notifies the CPU by generating an interrupt.

A received message is not written to the cache memory directly. This is because it may overwrite cached data which is being used.

4.3 Stride DMA

The stride DMA function assembles the regularly dispersed data items from memory into a message, as shown in Fig. 8, and transfers this message to the RTC or BIF. The function also regularly stores in memory the messages received from the RTC or BIF. The shaded areas in Fig. 8 indicate data items actually transferred.

These areas are specified using the *Addr, Size, Hskip, Vskip, Vcnt,* and *Hcnt* parameters.

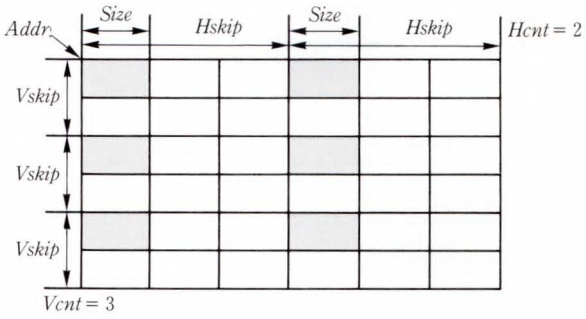


Fig. 8 – Stride DMA.

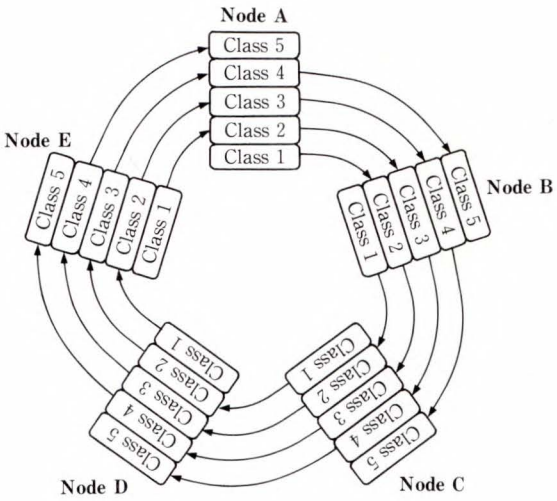


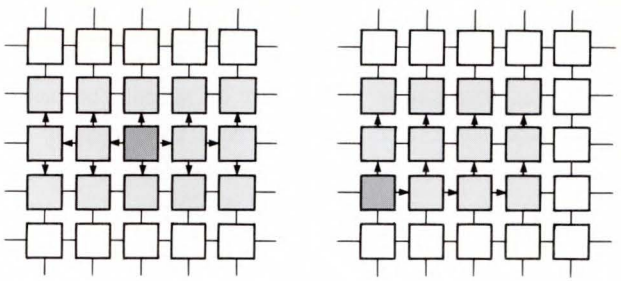
Fig. 9 – T-net routing scheme.

5. Communication networks

5.1 T-net

The T-net has a two-dimensional torus topology. The message routing scheme is *structured channel routing*. In wormhole routing, the intermediate node stores only a few bytes called a flit. When a routing node receives the message header, the node selects the channel of the next route. The node then transfers all subsequent flits for that message header to the route selected. Although wormhole routing has the advantage of low latency, deadlock may occur and throughput may suffer. This is because message transfer blocks the channel.

To minimize throughput deterioration and to avoid deadlock, the routing scheme employed incorporates the structured buffer pool algorithm into wormhole routing[12]. With this algorithm, a message does not block the channel it is using while it is being transferred. Figure 9 shows an example of this routing scheme. Each node has a buffer. These buffers can store flits equal to the maximum internode distance plus one. For example, if five nodes are connected by a unidirectional channel, each node has a buffer that can store five flits.



a) Double side b) Single side

Fig. 10 – Local broadcast in the T-net.

After a flit has been stored in the buffer, it is transferred to the next node. The data class is transferred along with the data itself. A flit of any data class can be transferred, and the data class can vary for each flit transferred. If more than one flit is to be transferred, the routing node determines which one to send. The algorithm avoids deadlock conditions because a transfer from class 1 to class 5 does not form a closed loop. Note that there is only one physical channel between nodes.

The RTC handles routing and flow control of communications between cells. It routes messages in a fixed way, first in the x direction and then in the y direction. The RTC can route at a rate of 160 ns/word. Provided there is no contention in the network, the latency is specified by $160 \times (D + N \times 4 + 1)$ ns, where D is communication distance and N is the message size in bytes. The maximum network size for the RTC is 32 by 32, a restriction imposed by the capacity of the RTC message buffer.

In addition to point-to-point communication, the RTC has a broadcast function on the T-net. Even though broadcast communication can consist of a series of point-to-point message routes, this is not practical since their latency would be unacceptably high. Using this function, a cell can broadcast its data to other cells. Figure 10 shows the path of a broadcast message in the RTC. The extent of the broadcast is restricted by encoding the address header of the

messages as the address of the node farthest from the source node. There are two patterns in broadcast messages – one for broadcast to both sides, and the other for broadcast to one side. A broadcast message and point-to-point communication messages can be transferred concurrently.

5.2 B-net

The B-net[13] performs communication among cells, and between cells and the host computer. The B-net consists of hierarchical common buses and a ring network. Up to eight cells share the lowest level bus. Four buses are connected to the ring node which is connected to the ring network. Thus up to 32 cells form a hierarchical common bus which is connected with a ring node. Since the host computer is also connected by a host interface, which looks like one of the ring nodes, the host computer and cells are equivalent.

The B-net protocol is similar to that of a simple common bus. Before data transfer, a cell or host trying to transfer data must flag a request to the B-net controller, then wait until the request is granted. After the B-net is granted to the requester, data transfer can begin.

A B-net interface (BIF) consists of read and write FIFO buffers and a scatter/gather controller. Data transfer is performed efficiently using the eight-stage FIFO buffers in BIF.

In many parallel programs, data must be distributed from the host computer to cells and results must be collected from the cells to the host computer (see Fig. 11). In a conventional machine, the host computer must interact with many cells. The time for data transfer setup increases as the number of cells increases. For efficient data distribution and collection, the BIF provides scatter and gather functions.

For broadcast, all data on the B-net from the host is written into the FIFO buffer in each cell and then the cells' MSC or IU reads it out.

For scatter, data on the B-net from the host is selected by the scatter/gather controller in each cell. In some cells, the data is selected and written into the FIFO buffer, then the MSC or IU in each cell reads it out from the FIFO

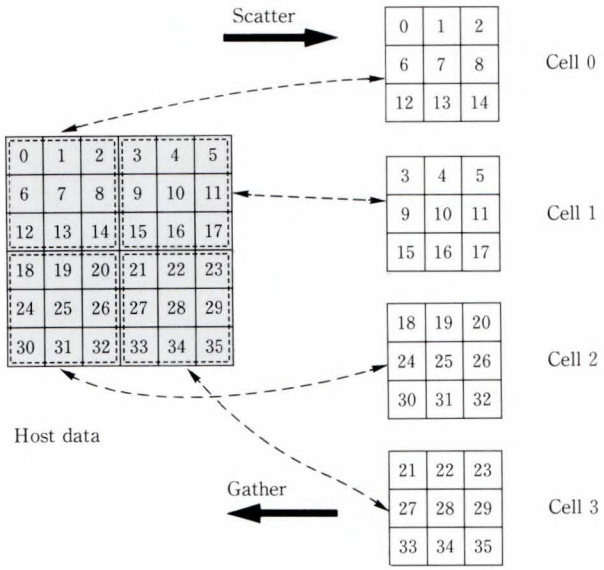


Fig. 11 – Scatter and gather operation.

buffer. In other cells, the data is not selected and is disposed of. Centralized data in the host is thus regularly dispersed to cells.

For gather, all cells try to output the data on the B-net. However, only data selected by the scatter/gather controller goes out on the B-net. The data not selected is held in the cells' FIFO buffer until selected. Then the DMA controller or host reads the data on the B-net. Regularly dispersed data among the cells is thus collected in a proper sequence. The data selection patterns are parameterized in the same format as the stride DMA pattern in the MSC.

5.3 S-net

The S-net has a tree topology and data is transferred serially. Data from each cell goes up to the root of the S-net along the tree topology. At intermediate nodes, data is merged by the logical AND operation. The logical AND of all the data is thus calculated at the root node. The result is then distributed to all cells along the tree from the root of the S-net.

The BIF also has a synchronization and status control block which executes barrier synchronization and status checking. It consists of 40 sets (8 system sets and 32 user sets) of a synchronization request register (SYRR), synchronization detection register (SYDR), status register (STR), and status detection register

(STDR). The output from the request register of each set is time multiplexed and sent out to the S-net. Data received from the S-net is then demultiplexed and used to update the detection registers.

Barrier synchronization is executed using the SYRR and SYDR. If all cells have set their SYRR, the SYDR is set and the SYRR is cleared according to the data from the S-net. The IU waits for synchronization detection by interrupt report or from SYDR polling.

Status checking is executed using the STR and STDR. The IU can set or reset its STR any time. The contents of the STDR are updated according to data from the S-net, and at any time the IU can sense the ANDed result of the STR value for all cells.

The synchronization and status control block also provides a combined operation for barrier synchronization and status checking. This operation provides status data only when barrier synchronization is detected. The overhead for barrier synchronization is 1.6 μs (system) to 5.2 μs (user).

6. Conclusion

A low latency, high throughput, and deadlock-free communication network is essential for highly parallel computers. The *structured channel routing* scheme used for the T-net meets these requirements.

For fast message communication, it is important to employ not only a fast communication network, but also fast message handling functions. Line sending and buffer receiving functions implemented in the MSC reduce message handling overheads. Line sending can initiate message transfer directly and immediately, and can be started in the same way whether data is cached or not.

The scatter and gather functions and barrier synchronization are implemented in the BIF for efficient data distribution and collection. The overheads for these functions do not increase even if the number of cells in the system increases. Without the scatter and gather controller in the BIF, the overhead for data transfer setup in host-cell communication dominates actual data transfer time in a highly parallel computer with a large number of cells. The BIF also reduces the overhead of barrier synchronization.

Using these fast communication features, the AP1000 achieves high efficiency in fine grain parallel processing such as LU decomposition[14)-16)]. The development of parallel programs is strongly influenced by available hardware. We believe that providing a powerful machine will motivate many programmers to develop applications on the AP1000.

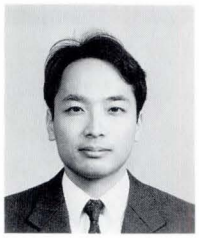
References

1) Seitz, C. L.: The Cosmic Cube. *Commun. ACM*, **28**, 1, pp. 22-33 (1985).
2) Tuazon, J., Peterson, J., and Pniel, M.: Mark IIIfp Hypercube Concurrent Processor Architecture. Proc. 3rd Conf. Hypercube Concurrent Comput. Appl., Pasadena, CA, 1988, pp. 71-80.
3) Dally, W. J., Chien, A., Fiske, S., Hoewat, W., Keen, J., Larivee, M., Lethin, R., Nuth, P., and Wills, S.: "The J-Machine: A Fine-Grain Concurrent Computer." *Information Processing 89*, Elsevier North Holland, 1989, pp. 1147-1153.
4) Arlauskas, R.: iPSC/2 System: A Second Generation Hypercube. Proc. 3rd Conf. Hypercube Concurrent Comput. Appl., ACM, 1988, pp. 33-36.
5) Merlin, P. M., and Schweitzer, P. J.: Deadlock avoidance in store-and-forward networks-I: store-and-forward deadlock. *IEEE Trans. Commun.*, **COM-28**, 3, pp. 345-354 (1980).
6) Scott, D. S., and Rattner, J.: The Scalability of Intel Concurrent supercomputers. Proc. 3rd Int. Conf. Supercomput., 1988, pp. 121-124.
7) Ametek Corporation: Ametek 2010 product announcement. 1987.
8) Dally, W. J., and Seitz, C. L.: Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, **C-36**, 5, pp. 547-553 (1987).
9) Ishihata, H., Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly

Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal process., 1991, pp. 13-16.
10) Ishihata, H., Inano, S., Horie, T., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Processor CAP-II. (in Japanese), 90-ARC-83, 1990, pp. 217-222.
11) Shimizu, T., Ishihata, H., and Horie, T.: A Message Controller for a Highly Parallel Processor, CAP-II. (in Japanese), 90-ARC-83, 1990, pp. 235-240.
12) Horie, T., Ikesaka, M., and Ishihata, H.: Routing Controller of Cellular Array Processor CAP-II. (in Japanese), 90-ARC-83, 1990, pp. 223-228.
13) Kato, S., Shimizu, T., Horie, T., and Ishihata, H.: Broadcast-network of Highly Parallel Processor CAP-II. (in Japanese), 90-ARC-83, 1990, pp. 229-234.
14) Horie, T., Ishihata, H., Shimizu, T., and Ikesaka, M.: AP1000 Architecture and Performance of LU Decomposition. Proc. 1991 Int. Conf. Parallel Processing., 1991, pp. 634-635.
15) Dongarra, J. J.: Performance of Various Computers Using Standard Linear Equations Software. *Tech. Rep.*, Univ. Tennessee, 1991.
16) Brent, R. P.: The LINPACK Benchmark on the AP1000: Preliminary Report. Proc. 2nd Fujitsu-ANU CAP Workshop, Australian Natl. Univ., 1991, pp. 28-29.

Hiroaki Ishihata

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
Waseda University 1980
Specializing in Computer Architecture and Parallel Computing



Takeshi Horie

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
The University of Tokyo 1984
Master of Electrical Eng.
The University of Tokyo 1986
Specializing in Computer Architecture and Parallel Computing

Toshiyuki Shimizu

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Physics.
Tokyo Institute of Technology 1986
Master of Computer of Sience
Tokyo Institute of Technology 1988
Specializing in Parallel Computing and Architecture

UDC 681.322.004.14

Performance Evaluation of the AP1000

● Toshiyuki Shimizu ● Takeshi Horie ● Hiroaki Ishihata

(Manuscript received August 11, 1992)

This paper evaluates the performance of the functions provided by the AP1000 for fast, parallel program execution. These functions include message handling, broadcasting, barrier synchronization, and gather/scatter functions. Message handling supports low-latency communication between processing elements, barrier synchronization allows all processing elements to be synchronized, and the gather/scatter functions support efficient communication between multiple processing elements. These functions were benchmarked at the user-library level. Also, this paper discusses the impact of library function speed on the performance of large standard benchmarks such as LINPACK, SLALOM, and SCG.

1. Introduction

Parallel processing on a distributed memory parallel processor (DMPP) includes message passing. DMPP performance depends on communication performance. To enhance system performance by increasing the number of processing elements (cells), more communication is required. The setup time for message passing becomes very important for DMPP system performance because increasing the number of processors increases the amount of message passing and decreases the size of messages.

Ordinary DMPPs, for example, the Intel iPSC[1], do not take the management of large numbers of processing elements into consideration and have too much overhead for message passing. Except when solving very large problems, their performance is limited, even when many processing elements are used.

The AP1000 has several unique structures that enable efficient parallel processing in a variety of applications. These structures include two message handling techniques (line-sending and buffer-receiving), broadcasting facilities for cell communication, fast barrier synchronization among cells, and scatter and gather mechanisms for host-cell communication[2)-5)].

Message handling sends messages directly from cache memory and receives them automatically at the circular buffer in main memory. The broadcasting facilities send a message to cells in a rectangular region specified by *xy* coordinates. The specified region has a 2-D torus topology. Barrier synchronization is performed using a tree network. The scatter mechanism enables efficient distribution of data in the host memory. The gather mechanism collects data distributed in the memories of the cells in a single step.

We evaluated the effectiveness of this architecture in a variety of applications. In Chap. 2 we describe the AP1000 architecture. In Chap. 3 we summarize the basic performance using user-level libraries. In Chap. 4 we evaluate the performance of some standard benchmarks and analyze the effects of the AP1000 architectural support.

2. AP1000 architecture

Figure 1 shows the AP1000 system configuration. The AP1000 is a distributed memory parallel processor (DMPP). Each processing element, or cell, is connected by three independent networks: the S-net, B-net, and T-net.

The T-net is used for inter-cell communication. This net has a two-dimensional torus topology, uses wormhole routing, and is constructed using the routing controller (RTC)[4].

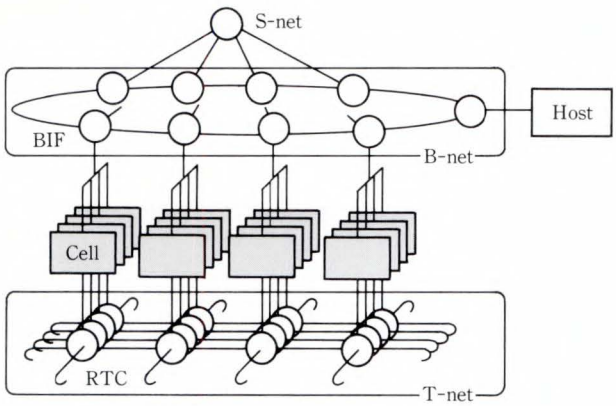


Fig. 1 – AP1000 system configuration.

The broadcasting facility is implemented on the T-net.

To send and receive messages on the AP1000, line-sending and buffer-receiving are used[6]. In this paper, these techniques are referred to as XY-transfer. Line-sending sends a message directly from the cache memory. If there is no message in the cache, line-sending moves a message from main memory automatically. Line-sending is activated in a way that is similar to cache flushing, and its setup overhead is quite small. When a message arrives, buffer-receiving moves the message from the network device to the circular buffer allocated in main memory, thus eliminating setup times such as processing interrupts.

The S-net is used for inter-cell synchronization. This net has a tree topology and is time-multiplexed. Cell requests for synchronization are merged or logically ANDed at the root of the S-net. The result is distributed to all cells connected to the S-net. Because of these mechanisms, synchronization is achieved quickly and the synchronization time does not depend on the number of cells. In conventional parallel processors, cell synchronization is done using message passing and the synchronization time is a function of the number of cells.

Scatter and gather mechanisms are implemented on the B-net. B-net connection is done via the bus interface (BIF). To distribute data from a host to cells, the host initiates a transfer to write all data at the same time. The data which appears on the B-net is selected and

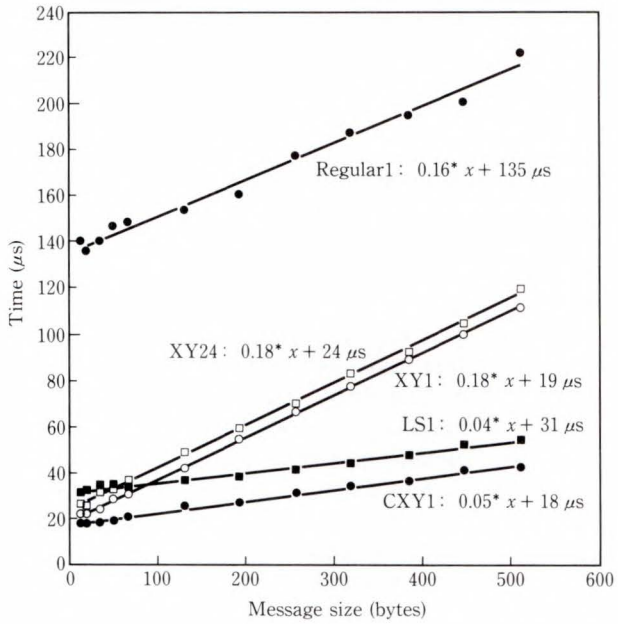


Fig. 2 – Pingpong performance.

received by the BIF in each cell. To gather data from cells, each cell's BIF monitors the B-net to determine when the cell puts data on the B-net. Data is then merged on the B-net and collected.

3. Performance of basic functions

This chapter evaluates the performance of basic functions using user-level libraries. We use the AP1000 in the 2-D torus configuration, which is the AP1000's physical configuration.

3.1 Communication latency

The AP1000 uses the T-net for inter-cell communication and incorporates wormhole routing. This routing and XY-transfer provide low-latency communication. To determine the communication latency between cells, the times taken to send messages of various lengths from a master cell to a slave cell and then back were measured. This test is known as the pinpong benchmark[7]. During this test the other cells are idle and have no effect on the communicating cells. Half of the time for this test is used to evaluate the overall latency, which includes latency in the network and the message handling overhead.

Figure 2 shows the message handling time versus message size in bytes. In this figure,

Regular1 shows the timing of ordinary Direct Memory Access (DMA) transfer functions l_asend( ) and crecv( ). XY1 and XY24 show the results for an XY-transfer distance of 1 and 24, respectively. The times taken to send x bytes over a transfer distance of 1 and 24 are $T_{XY1} = 0.18 \times x + 19\ \mu s$ and $T_{XY24} = 0.18 \times x + 24\ \mu s$, respectively. The differences in the times for the various inter-cell distances concur with the hardware specification of 160 ns/unit-distance. The XY-transfer setup time is very small because a great deal of overhead is eliminated. Total throughput is about 5.5 Mbytes/s. Throughput is slightly lower than with Regular1 because xy_recv( ) copies the receipt message to a user-specified buffer[Note]. When xy_crecv( ) (referred to as CXY1) is used, the time is $T_{CXY1} = 0.05 \times x + 18\ \mu s$. A throughput of about 20 Mbytes/s is obtained because xy_crecv( ) eliminates copying at reception. LS1 shows the speed-up that results when message handling is switched from ordinary DMA to XY-transfer by specifying the −LSEND option at runtime. Regular1 and LS1 use the same object, but the runtime options are different. The time for LS1 is $T_{LS1} = 0.04 \times x + 31\ \mu s$. The setup time is required for runtime parameter conversion.

3.2 Broadcasting

X, Y, and XY directional broadcasting are done on the T-net. We measured the Y-direction broadcasting performance of 8 and 16 cells aligned in the Y direction. In this experiment, cells whose Y cell-id is 0 broadcast the same data in the Y direction to 7 or 15 cells. The results of X-direction and XY-direction broadcasting are the same. In Fig. 3, y_brd16 shows the timing of a 256 (16 × 16) cell configuration and y_brd8 shows the timing of a 64 (8 × 8) cell configuration. ybrd16 and ybrd8 are the results of a simulation using point-to-point communication. The timing results are $T_{y\text{-}brd8} = 0.11 \times x + 2.8\ \mu s$, $T_{y\text{-}brd16} = 0.13 \times x + 2.7\ \mu s$, $T_{ybrd8} = 0.20 \times x + 27\ \mu s$, and $T_{ybrd16} = 0.27 \times x + 36\ \mu s$, where x is the number of broadcast bytes.

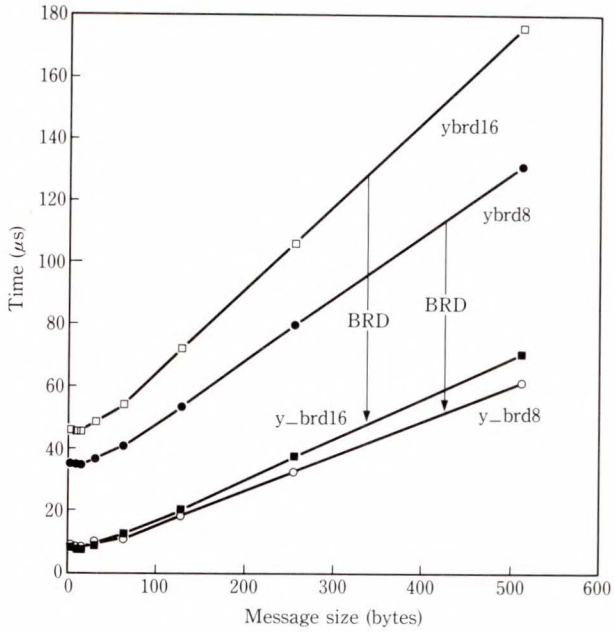


Fig. 3 − Broadcasting.

The setup time is smaller than that required for the test described in Sec. 3.1. Throughput is higher because sending cells always send messages and receiving cells always receive messages; therefore, sending and receiving are pipelined by multiple (1 000) trials.

By comparing the y_brd and ybrd results, we can see that the broadcasting mechanism reduces the setup time by about 90 % and doubles the throughput.

3.3 Global functions

Global functions, for example, xy_damax( ) and xy_dsum( ), are used to compute the maximum values and summations of all data for all cells. These functions are implemented using binary tree communication. The return values for all cells are the same.

xy_dsum( ) computes an 8-byte double-precision real summation for all data in all cells. xy_damax( ) finds the absolute maximum value in double-precision real value of data in all cells. These functions use short messages in the 16-byte format shown in Table 1. DATA index in Table 1 indicates which cell held the selected data.

Note: Regular copies a message when sending. This copying is faster than reading from main memory, which causes a cache miss in the xy_recv( ) function.

Table 1. Message format for global functions

Offset	Field	Size (bytes)
0	Routing header	4
4	DATA index	4
8	DATA(float,int,double)	8

Table 2. Completion times for global functions (in μs)

Cells	4 × 4	8 × 8	16 × 16
x_damax/y_damax	30	40	52
xy_damax	58	78	104
x_dsum/y_dsum	23	30	39
xy_dsum	44	59	79

Since messages in this format only occupy one cache line, communication is efficient. The nine least-significant bits of the routing header specify the type of global function. Table 2 shows the completion times for global functions.

Since xy_damax( ) performs more calculations, for example, indexing, xy_dsum( ) is faster than xy_damax( ).

Although xy_dsum( ) and xy_damax( ) calculate only one element, some applications perform element-wise calculation of a vector, for example, summation, and find the maximum value. The performance results for vector calculation are shown in Fig. 4.

Figures 5 and 6 show programs used to measure performance. These programs calculate element-wise vector summation in the Y-direction. The program shown in Fig. 5 gives the y_dsum results, and the program in Fig. 6 gives the btree results. The numbers 16 and 32 in Fig.4 indicate the number of cells in the Y-direction (*ncely*). Function xy_send(*rx, ry, type, msg, size*) sends a message (*msg*) from the cell-id in 2-D representation (*cidx, cidy*) to the cell of (*cidx + rx, cidy + ry*).

In Fig. 4, we can see that the y_dsum and btree performances intersect at vector lengths of 3 and 4. y_dsum( ) is faster because the message length is limited to 16 bytes (including headers), and only one library call is needed. Since the xy_send( ) function appends a 32-byte header to the message, the amount of data to be exchanged

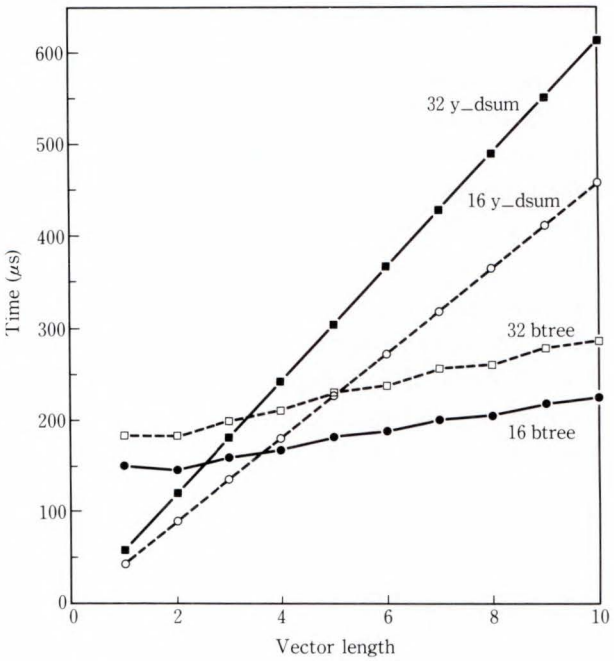


Fig. 4 – Binary tree and y_dsum performance.

```
/*
** Global sum of each vector element.
*/
for (i=0; i < vlen; ++i) /* vlen: vector length */
    /* sum of y-direction */
    y_dsum(results[i],&results[i]);
```

Fig. 5 – Global function (y_dsum).

```
/*
** Global sum of each vector element.
**   Binary tree sum and broadcast
*/
length = vlen * sizeof(double);
for (i = 1; i < ncely; i += i) {
    if (cidy & i) {
        k = xy_send(0, -i, MSGTYP, results, length);
        break;
    }
    else if (i+cidy < ncely) {
        k = xy_recvs(0, i, MSGTYP, tmp, length);
        qadd_(tmp, results, vlen);
        /* vector add: results += tmp */
    }
}
y_brd(cidx,results,length); /* broadcast */
```

Fig. 6 – Binary tree sum and broadcasting.

when the vector length is 4 equals that for the y_dsum. The results show that the main contribution to efficient implementation from y_dsum is the incorporation of specially formatted messages.

3.4 Scatter and gather functions

Scatter and gather functions are implemented on the B-net. Host data is distributed to cells and distributed data is collected by single operations.

Figure 7 shows the performance of distribution (Scat) and collection (Gathc, Gathf) from cells on 64 and 512 cell systems. The x axis indicates the total data size to be collected or distributed. The y axis gives the performance time.

In Fig. 7, Poll64 shows the timing for polling (with no support from gather mechanisms) realized by the host-cell communication function in a 64-cell system. The results for a 512-cell system are not shown because they required too much time to obtain. The number of cells does not affect the results of Gath or Scat. Performance is limited by the speed of the memory. Gathf shows the result of systematic fine-scale data collection in which each cell puts out 4-byte data. Gathc shows the result of coarse data collection in which each cell puts out all of its data at the same time. For coarse data collection (Gathc), large data blocks are read from cell memory and performance is limited by the cell memory speed. For fine data collection, the data in each cell's BIF FIFO is merged into the B-net and sent to the host. Since this approach is similar to p-way interleaved access, where p is the number of cells, good performance is obtained. The data transfer rate is 38 Mbytes/s, which is very close to the host interface's design specification of 40 Mbytes/s for memory writing.

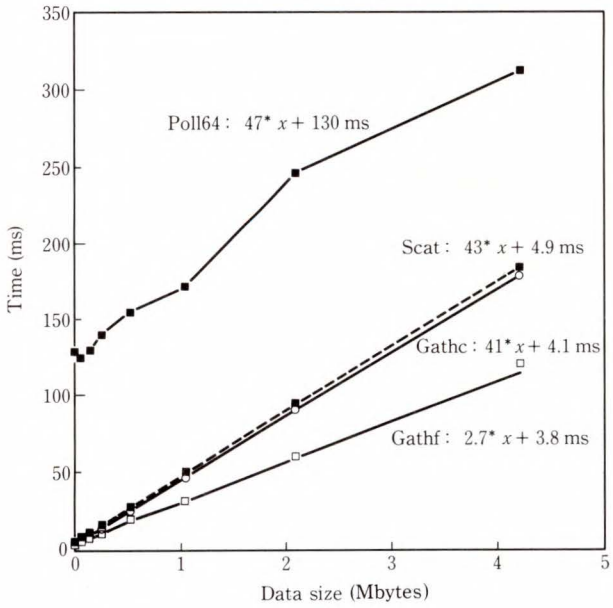


Fig. 7 – Scatter and gather performance.

The times required to gather x bytes of data for the host are $T_{Gathf} = 0.027 \times x + 3\,800\,\mu s$, $T_{Gathc} = 0.041 \times x + 4\,200\,\mu s$, $T_{Scat} = 0.043 \times x + 4\,900\,\mu s$, $T_{Poll\,64} = 0.047 \times x + 130\,000\,\mu s$, and $T_{Poll\,512} = 0.16 \times x + 1\,900\,000\,\mu s$. ($T_{Poll\,512}$ is the time required for a 512-cell system.) It is time-consuming to collect data without gather functions, especially for large numbers of cells. To simulate the Gathf function using the Poll function, the received data should be rearranged after communication. The scatter function also reduces interaction between cells. To simulate the scatter function by ordinary host-cell communication, the number of interactions is the same as the number of cells.

It is essential to reduce the number of interactions between the cells and a host. A SUN4/330 running under UNIX[Note] is used as the AP1000 host. The setup time is very large.

3.5 Barrier synchronization

Ordinary DMPP uses message passing for synchronization. This method, however, is very slow and transfer data and synchronization messages can be mixed, causing confusion.

The AP1000 incorporates a private network for synchronization to solve the above problems. The synchronization time is 10.6 μs and does not vary with the number of cells. For synchronization with message passing, the

Note: The UNIX operating system was developed and is licensed by UNIX System Laboratories, Inc.

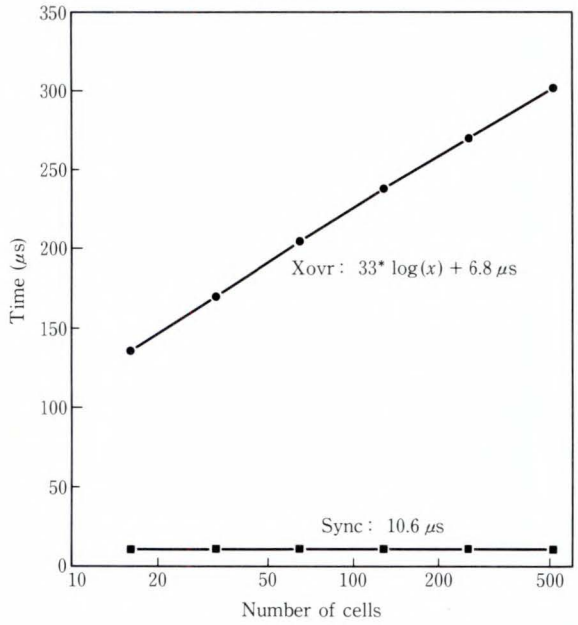


Fig. 8 – Synchronization time.

synchronization time is $T_{XOVR} = 33 \times \log(p) + 6.8$ μs, where p is the number of cells.

4. Benchmark results

We benchmarked some programs using user-level libraries, and investigated the effects of the message handling, broadcasting, and gather/scatter functions. The following sections briefly describe the benchmarks and our results.

4.1 LINPACK

The LINPACK[8] program solves dense systems of equations using LU decomposition. The problem size, n, is for an $n \times n$ matrix. We measured the Gaussian elimination and backward substitution parts as in the LINPACK benchmark[9]. Matrix generation and result output were not done.

Blocked LU decomposition is used here with partial pivoting. Global functions (xy_damax) are used to find the pivot row. To send pivot row data to other cells and to perform backward substitution, X- or Y-directional broadcast functions (x_brd, y_brd) are used.

Table 3 shows the analysis of library calls. Results are shown for square matrix sizes of 1 000, 2 000, and 4 000 using 64 or 256 cells. *BRD*, *GLB*, and *XY* stand for broadcasting, global functions, and XY-transfer data, respectively. The top figures are the average counts of library calls, and the lower figures are the average times, in seconds, consumed in that library. L is the time consumed by communication as a percentage of the total time for calculation. T is the time required to perform the benchmark. (The T values of 4.67 and 25.0 in the table include the function trace delay, and the T values of 3.59 and 22.5 do not include the trace delay.) The trace overhead can be determined

Table 3. Analysis of library calls for LINPACK benchmark

Cells	Size	*BRD*	*GLB*	*XY*	L (%)	T (s)
64	1 000	6 861	1 000	17		3.59
		1.3	0.2	–	32	4.67
	2 000	13 679	2 000	22		22.5
		4.0	0.42	–	18	25.0
	4 000	27 493	4 000	32		159
		14.0	1.2	–	9	164
256	1 000	6 972	1 008	11		1.4
		1.1	0.2	–	49	2.5
	2 000	11 868	2 000	17		7.0
		4.0	0.38	–	39	9.5
	4 000	27 721	4 000	22		45
		9.0	0.91	–	20	50

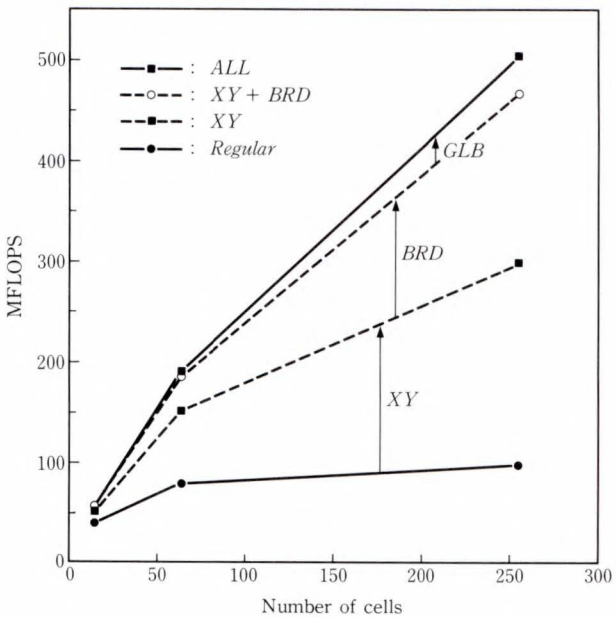


Fig. 9 – LINPACK performance.

from the trace and benchmark times.

Figure 9 shows the FLOPS results for the 1 000 × 1 000 LINPACK benchmark. In this figure, *Regular* shows the results when ordinary message passing functions (l_asend and crecv) are used. Broadcasting and global functions are simulated by ordinary message passing. *XY* shows the results for XY-transfer (xy_send, xy_recv). *XY + BRD* shows the results when XY-transfer and broadcasting are used. *ALL* shows the results when XY-transfer, broadcasting, and global functions are used.

XY-transfer and broadcasting greatly affect benchmark speed. The Regular results show that unless XY-transfer and broadcasting are done, increasing the number of cells does not increase speed. The effects of global functions are not outstanding because the number of global function calls is small. The number of XY-transfers is also small, but these functions are used to simulate broadcast and global functions. Therefore, the performance of XY-transfer is very important.

4.2 SCG

SCG solves 2-D Poisson equations using the scaled conjugate gradient method. n is the size of the mesh to be partitioned ($n \times n$). This problem is equivalent to solving a sparse $n^2 \times n^2$ matrix.

In this program, global functions are used to calculate the summation of inner-products. Table 4 shows the analysis of library calls. The mesh size is varied from 100 × 100 to 400 × 400. The statistics are arranged as in Table 3.

Figure 10 shows the FLOPS performance for the solution of a 200 × 200 mesh. This figure shows a trend similar to that observed for LINPACK. The difference between *XY* and *XY+BRD* is that the latter uses the broadcasting function to simulate global functions. In simulation, we use an algorithm similar to the one shown in Fig. 6. Unlike the LINPACK case, using global functions here does make a difference because global functions consume large amounts of time.

Table 4. Analysis of SCG benchmark

Cells	Size	*GLB*	*XY*	*L* (%)	*T* (s)
64	100	445	888		0.64
		0.10	0.21	34	0.93
	200	893	1 784		4.1
		0.27	0.66	19	4.8
	400	1 789	3 576		43
		1.2	2.5	8	45
256	100	445	1 332		0.25
		0.10	0.24	54	0.64
	200	893	2 662		1.3
		0.32	0.69	45	2.3
	400	1 789	3 576		8.3
		0.86	1.8	27	9.6

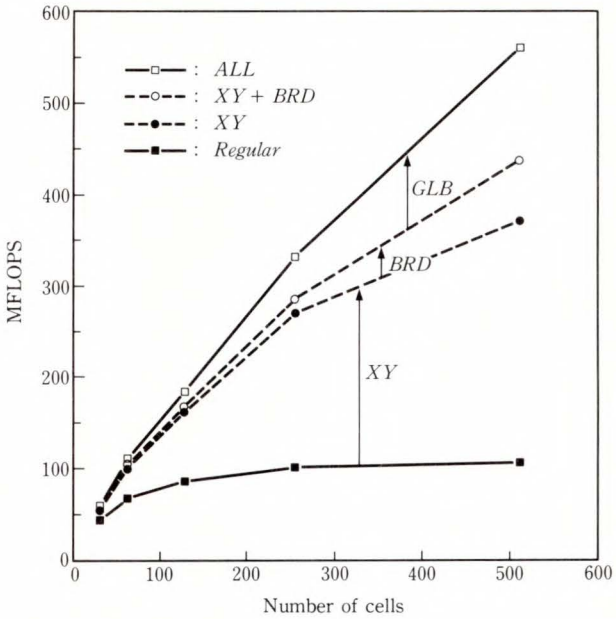


Fig. 10 – SCG performance.

4.3 SLALOM

SLALOM is a standard benchmark program that finds the optical radiosity of the interior of a box[10]. The walls of the box are divided into patches, and the color is calculated for each patch. This benchmark evaluates the overall calculation, from the reading of initialization data to the output of results. The results are given as the number of patches calculated in 60 seconds.

The program can be divided into four parts: initial file reading, matrix generation, solution

by the LDL method, and answer file output.

Initial file reading takes about 0.2 seconds. The calculation order for matrix generation is $O(n^2/p)$, where n is the number of patches and p is the number of cells. This part consumes 10 % to 30 % of the computation time. The calculation order of the LDL solution is $O(n^3/p)$. This part consumes 70 % to 90 % of the computation time. The results are collected by the host from cells and written to the answer file. The gather function is used to collect the answer. If the gather function is not used, the host communicates $\sqrt{p}$ times with the diagonal cells which contain the answer, where p is the number of cells. This part consumes about 5 % of the computation time. Table 5 shows the analysis of library calls.

Regular in Fig. 11 shows the results when only ordinary message communication functions are used. Gath shows the results when the gather function is used. *XY + Gath* shows the results for XY-transfer when the gather function is used. *ALL* shows the results when all functions, including the broadcasting function, are used. The problem size is set to the number of patches that *ALL* can calculate in about 60 seconds.

Without the gather function, we cannot increase the number of patches by adding cells because the amount of answer data becomes excessive and host-cell interaction increases.

When the number of patches is increased, XY-transfer is used to increase the computation speedup ratio to compensate for the consequent increase in the amount of communication. Since broadcasting does not consume much time, the contribution by the broadcast function is not significant. However, exchanging the Regular and XY-transfer simulation functions makes a significant change. This is also true of other benchmarks.

Table 6 summarizes the benchmark results.

Table 5. Analysis of SLALOM benchmark

Cells	Size	*BRD*	*XY*	*Gath*	*L* (%)	*T* (s)
16	1 163	7 800	2 400	1		58
		5.8	1.8	0.007	14	69
64	1 943	12 387	2 200	1		59
		10	2.8	0.004	22	68
256	3 110	19 243	1 850	1		58
		16	4.1	0.003	31	72

5. Conclusion

We evaluated the effects of AP1000 architectural support of application programs by running several benchmarks.

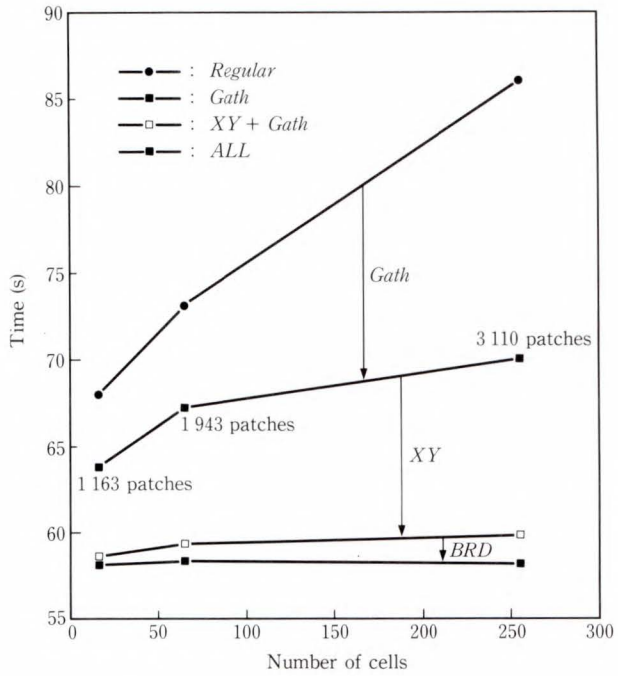


Fig. 11 – SLALOM performance.

Table 6. Benchmark summary

Benchmark	Cells	Problem size	Results (MFLOPS)	Time (s)
LINPACK	8 × 8	1 000	190	3.51
	8 × 8	2 000	237	22.5
	8 × 8	4 000	268	159
	16 × 16	1 000	505	1.36
	16 × 16	2 000	755	7.07
	16 × 16	4 000	951	44.9
SCG	16 × 16	100	212	0.251
	16 × 16	200	341	1.26
	16 × 16	400	412	8.32
	16 × 32	100	267	0.200
	16 × 32	200	536	0.799
	16 × 32	400	678	5.06
SLALOM	4 × 4	1 163	27.8	58.0
	8 × 8	1 943	119	58.7
	16 × 16	3 110	474	57.9

We showed that minimizing the sending and receiving overhead reduces the computation time. When a problem is small, the percentage of total time consumed by message passing is large. Therefore, reducing the overhead for message passing makes a significant improvement. When the number of cells increases, the amount of communication increases, messages become small, and the message passing setup time can affect performance. Broadcast functions have an effect similar to a reduction in the amount of communication, and therefore reduce execution time.

To solve a large problem, good communication performance between cells and a host is very important during initialization and output. Since the AP1000 has scatter and gather functions on the B-net, these processes can be done efficiently. Speed is limited, however, by the B-net's bandwidth (50 Mbytes/s) and the memory bandwidth of the host interface (host-to-cell: 25 Mbytes/s, cell-to-host: 40 Mbytes/s). If we use local disks and frame buffers on each cell as option boards, these limitations can be removed.

Fast barrier synchronization is achieved, regardless of the number of cells. The benchmarks we evaluated do not explicitly use barrier synchronization, and we did not evaluate its effect on applications. However, synchronization mechanisms do allow message communication to be isolated from the network, enabling the easy isolation of problem synchronization and messages. In fact, in program development and debugging, synchronization is extensively used and a fast barrier is very valuable.

Low-latency communication, broadcasting, and the gather and scatter functions supported by the AP1000 are used for fast execution of applications, especially when an increase in the number of cells increases the effects of these mechanisms and therefore maintains scalability. Without these support mechanisms, some applications would not speed up, even with an increased number of cells.

The programs we evaluated are good from the perspectives of load balancing and the balance of calculation and communication times. Benchmark program performance can be increased for large problems. However, to assess the performance of a wide range of applications, we evaluated various kinds of overhead. Accordingly, we limited the problem size to keep the overhead components visible.

The architectural supports provided by the AP1000 environment clearly facilitate efficient execution of parallel programs. Also, these supports can speed up a program for a variety of applications.

References

1) Ramune Arlauskas: iPSC/2 System: A Second-Generation Hypercube. Proc. 3rd Conf. Hypercube Concurrent Comput. Appl., 1988, pp. 33-36.
2) Ishihata, H., Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Process., 1991, pp. 13-16.
3) Ishihata, H., Horie, T., Inano, S., Shimizu, T., Kato, S., and Ikesaka, M.: Third Generation Message Passing Computer AP1000. Proc. Int. Symp. Supercomput., 1991, pp. 46-55.
4) Horie, T., Ishihata, H., Shimizu, T., Kato, S., Inano, S., and Ikesaka, M.: AP1000 Architecture and Performance of LU Decomposition. Proc. 1991 Int. Conf. Parallel Process., 1991, pp. 634-635.
5) Horie, T., Ishihata, H., and Ikesaka, M.: Design and Implementation of an Interconnection Network for the AP1000. Information Processing 92, **I**, North-Holland, Elsevier Science Publishers B. V., 1992, pp. 555-561.
6) Shimizu, T., Ishihata, H., and Horie, T.: Low-Latency Message Communication Support for the AP1000. Proc. 19th Annual Int. Symp. Comput. Architecture, 1992, pp. 288-297.
7) Hockney, R.: Performance parameters and benchmarking of supercomputers. *Parallel computing*, **17**, 10 & 11, pp. 1111-1130 (1991).
8) Brent, R. P.: The LINPACK benchmark on the AP1000. Proc. 4 th Symp. Frontiers Mas-

sively Parallel Computation, 1992.
9) Dongarra, J. J.: Performance of Various Computers Using Standard Linear Equations Software. Technical report, Univ. Tennessee, 1991.
10) Gustafson, J. et al.: SLALOM UPDATE. *Supercomputing Review*, pp. 56-61 (1991).

Toshiyuki Shimizu

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Physics Eng.
Tokyo Institute of Technology 1986
Master of Computer Sience
Tokyo Institute of Technology 1988
Specializing in Parallel Computing
and Architecture

Takeshi Horie

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
The University of Tokyo 1984
Master of Electrical Eng.
The University of Tokyo 1986
Specializing in Computer Architecture
and Parallel Computing

Hiroaki Ishihata

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
Waseda University 1980
Specializing in Computer Architecture
and Parallel Computing

UDC 681.3.06

AP1000 Software Environment for Parallel Programming

● Takeshi Horie ● Morio Ikesaka

(Manuscript received August 11, 1992)

Programs for massively parallel computers are very difficult to write, debug, and tune. Most of the tools developed to support parallel program development are for parallel systems with fewer than 100 processors. The runtime monitor shows the processor status and load at run time. The performance analyzer evaluates program execution statistics and presents a graphic display of program behavior. The parallel debugger is used to debug programs running simultaneously on multiple processors. This paper describes the software environment of the AP1000, a programming model for the AP1000, and tools that support parallel programming.

1. Introduction

Programs for massively parallel computers are very difficult to write, understand, and debug. One of the performance tuning and program debugging tools developed to support parallel programming is Malony and Reed's HYPERMON hardware system, which captures and records software performance traces generated on the Intel iPSC/2 hypercube[1),2)]. Malony and Reed have also studied the instrumentation perturbations of software event tracing on the Alliant FX/80 vector multiprocessor[3)]. Sharma has developed a prototype run-time performance monitoring environment for the Cedar multiprocessor[4)]. Rover et al.[5)] have demonstrated the utility of performance visualization for fine-tuning algorithms and for the study of phenomena using software tools developed for distributed memory machines. Then, they applied their findings to the SLALOM program. Couch has described a way to organize and present debugger output[6)].

Parallel computers with more than 100 processors, for example, the iPSC/2[7)], the parallel computers developed by the Touchstone project[8)], nCUBE/2[9)], and the AP1000[10)] have been used in diverse applications. However, most of the performance tuning tools and debuggers concentrate on systems with fewer than 100 processors, and their application to larger systems seems to be difficult.

The following problems must be considered when applying performance and debugging tools to parallel systems with more than 100 processors:

1) Displaying the processor status and load on the fly necessitates on-the-fly information gathering from the processors. This is likely to cause perturbations in execution because data gathering makes heavy demands on the network.
2) Performance data gathering and analysis for performance tuning is time-consuming because of the large amounts of trace data that must be accumulated. This data cannot be loaded to the host computer. For example, even if each processor produces only 100 Kbytes of trace data, the amount of data produced by 512 processor is 51 Mbytes. Therefore, trace data must be collected and analyzed as efficiently as possible.
3) It is possible to display the data for debugging and performance tuning for small parallel systems. For large systems, however, we cannot display all the data at once to easily find bottlenecks and bugs in parallel programs. Therefore, we must extract and display only data which will ensure meaningful performance analysis and debugging.

We developed the runtime monitor and

performance analyzer for the AP1000. The runtime monitor shows the processor status and load at run time and is used for debugging and rough evaluation of program execution. The performance analyzer evaluates trace data recorded during execution and presents a graphic display through the X Window interface. The information displayed includes the processor status, the number of active processors, and the message transfer ratio. These displays make it easier to determine the location of bottlenecks so that performance can be optimized.

To debug parallel programs at run time, we must consider how to display the values of variables. We developed a parallel source debugger for the AP1000 which is based on GNU gdb. This debugger can display multiple data and allows breakpoints to be set in all processors at the same time.

In this paper we describe the software environment in which parallel programs are developed on the AP1000. In Chap. 2, we give an overview of the software environment for the AP1000. Chapter 3 describes the design and implementation of the runtime monitor, performance analyzer, and parallel debugger on the AP1000.

2. AP1000 architecture

2.1 Hardware

The AP1000 is a distributed-memory parallel computer that uses 16 to 1 024 processing elements (PEs) (see Fig. 1). The host is a SUN workstation. The AP 1000 has the following three independent networks:

1) The T-net, which provides point-to-point and local broadcast communication.
2) The B-net, which provides broadcast communication and data distribution-collection between processors or between the host and processors.
3) The S-net, which provides barrier synchronization and status checking.

To reduce the overhead for message sending and receiving, the message controller interfaces between the networks and processors.

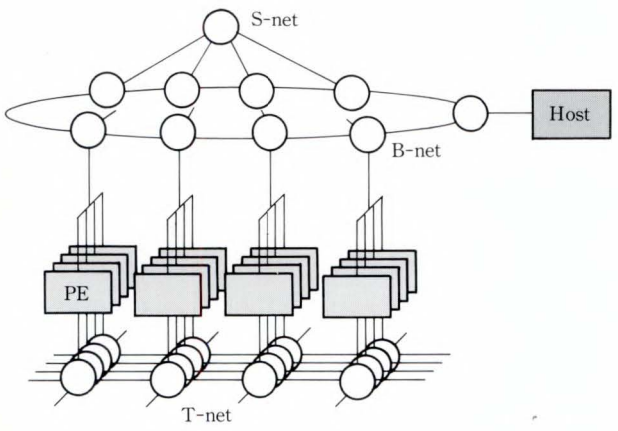


Fig. 1 – AP1000 configuration.

Performance analysis tools on parallel systems need a global hardware clock for all processors to ensure precise analysis of the interactions between processors. Malony developed the special HYPERMON hardware because iPSC/2 does not have a global clock[1]. The AP1000 operates synchronously and each processor has a timer resolution of 1.28 μs. This timer is synchronized among all processors and allows us to efficiently implement performance analyzer tools.

2.2 Software

This section describes a parallel programming model and the software configuration of the AP1000. The CASIM software simulator is also described.

2.2.1 Programming model

To develop a wide range of parallel applications, a flexible parallel programming interface must be prepared so that application programmers can implement a variety of parallel algorithms. We therefore adopted a computation interface model based on message passing.

Application programmers develop parallel programs for the AP1000 in the following sequence:

1) Parallel algorithms are translated into parallel programs based on the AP1000 parallel programming interface.
2) Programs are excuted, debugged, and evaluated on the workstation using the AP1000 parallel software simulator, CASIM.
3) The application is executed, debugged, and evaluated under the AP1000 operating sys-

tem.

The AP1000 provides a computation model that is focused on task and message communication. A task is a basic unit which can be executed in parallel, can memorize its own internal states, and can act according to received messages. Tasks cooperate to solve problems by exchanging messages.

For high-speed processing, a strategy is needed for task division. Parallel processing based on data parallelism is an important strategy, especially in large-scale, parallel systems. The AP1000 has a message communication interface to facilitate divided data access.

Application programmers create their parallel algorithms based on the above parallel programming interface. Application programs can be written in C or Fortran. Parallel processing functions, for example, communication and synchronization, can be represented by the AP1000 parallel library, which supports various types of message communication for easy implementation of data parallel algorithms.

2.2.2 Software configuration

Figure 2 shows the software configuration for the AP1000. A user program consists of a process on the host computer and tasks in each processor. CAREN is a server which initializes the AP1000 hardware, sets the execution environment, creates tasks, and controls message transfer. Processes such as a user host program, debugger, runtime monitor, and UIO (a process for user interaction) communicate with the server process using a UNIX pipe.

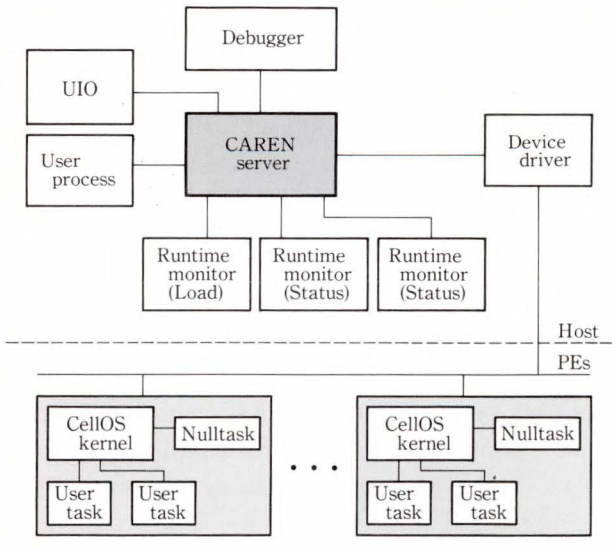


Fig. 2 – AP1000 software configuration.

A CellOS, light-weight, message-based operating system was developed to support parallel program execution on the AP1000. CellOS resides in each processor and has the following three features:

1) Support of intertask communications, both within processors and between processors.
2) Performance of multitask processing within a processor.
3) Support of basic functions for debugging and evaluation.

CellOS provides a multitask processing environment for the processors so that they can execute different functions. Trace data for the performance analyzer is recorded in CellOS.

Tasks are activated by messages received from other tasks, and procedures based on these messages are executed. After completing a procedure, a processor becomes inactive and remains so until it is activated by another message. Tasks are scheduled at each message reception or synchronization request. Each task has a priority, and CellOS gives the excution right to the highest priority task. This task scheduling makes CellOS simple and light-weight by decreasing the overhead for context switches.

2.3 Software simulator: CASIM

We developed a parallel software simulator, CASIM, to support the development of parallel programs on the AP1000. CASIM has the following three features:

1) Configuration can be done on a workstation so that application programmers can use a wide variety of programming tools, for example, useful window systems and powerful debuggers provided by the workstations.
2) Simulation of parallel processing supported by the AP1000 parallel library. Application programmers can verify parallel processing functions such as message communication and synchronization.
3) Guarantee of a parallel programming interface, providing the same parallel program-

ming interface as the AP1000, so that parallel programs developed using CASIM can be executed on the AP1000 without changing source programs.

2.3.1 CASIM configuration

CASIM runs on Sun workstations (see Fig. 3). Tasks on the AP1000 correspond to workstation processes. CellOS functions are supported by the CASIM server process. The CASIM server handles communication, synchronization, and other service requests to CellOS. CASIM uses interprocess communication by TCP/IP to establish an interface between tasks and CellOS.

2.3.2 Specifying the process execution window

CASIM executes task processes in an X Window environment. The standard output from processes is displayed on the windows corresponding to each process. Processes are debugged using source debuggers such as dbx and gdb. CASIM can also be accessed through the network from remote hosts (i.e. from other than the workstation where CASIM is installed). Process execution is checked using the windows on the remote host.

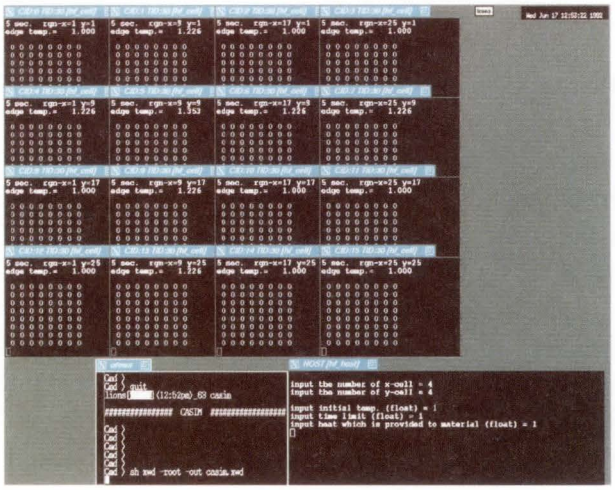


Fig. 3 – CASIM.

3. Software development tools

This chapter describes our three software development tools: the runtime monitor, performance analyzer, and parallel debugger.

3.1 Runtime monitor

The runtime monitor presents a graphic runtime display of the process status and load of each processor. This tool is used to debug and evaluate parallel programs on the AP1000.

In each processor, CellOS sends processor information to the host, which gives the information at runtime on the X Window system. The following information is displayed:

1) The status of tasks in processors (e.g. RUN, READY, and WAIT)
2) The load for each processor (in graphic form)
3) The messages written to the standard output of processors

The runtime displays of each processor's status make it easy to grasp the behavior of multiple processors during execution. Graphics capability greatly aids programmers in the debugging of parallel programs, especially in large-scale parallel systems.

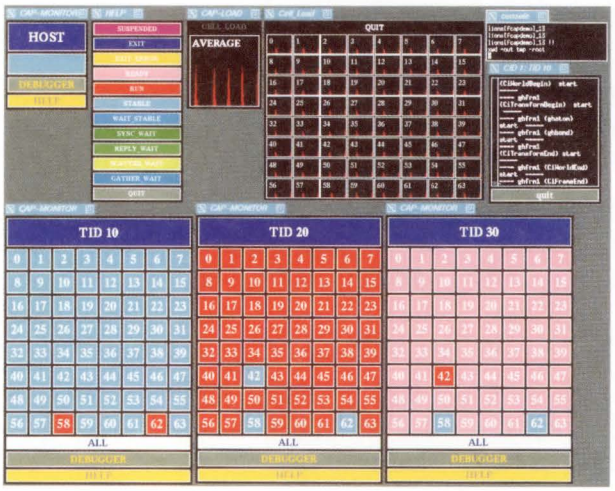


Fig. 4 – Runtime monitor.

The runtime monitor has two windows: one for the processor status and one for the load. The processor status window shows the status of each task, and the load window shows the load of each processor and the average processor load. If there is more than one task in each processor, several status windows are displayed. Figure 4 shows an example of a runtime monitor display. The boxes in this display represent the processors, and the colors of the boxes indicate the status of each processor. There are three tasks (task IDs 10, 20, and 30) in each processor.

In addition to the processor status, the runtime monitor displays messages written to the standard output of the processors and invokes a parallel debugger. The load window shows the load of each processor and the average processor load.

When gathering information from processors, the runtime monitor uses the AP1000 B-net, which has special hardware for data distribution and collection. Without this hardware, each processor would send information independently, resulting in excessive data being sent to the host computer. Processor data is gathered through the B-net. The host receives this data as a single message, so the overhead for data reception at the host does not increase with the number of processors. We believe that large parallel systems require hardware support when gathering data from processors.

3.2 Performance analyzer

The performance analyzer evaluates trace data recorded during execution and displays it in graphic form. We developed two types of performance analyzer. The first type stores trace data in the processor memory and then analyzes it. The second type calculates trace-data statistics during execution. The second type is useful when extracting only statistical trace information such as the number of messages sent. It can also be used for extensive executions because it does not store trace information. The drawback with this type of analyzer is that it has a large trace overhead and does not provide a graphic output. We will therefore focus on the first type of analyzer.

During execution of an application program, all event traces are recorded. After execution, instead of being sent to the host, the traces are left in processor memory because it is too expensive to collect them. For example, the traces recorded by a LINPACK benchmark execution using 512 processors constitute 657 Mbytes of data. Our performance analyzer runs on the processors and the host. The analyzers running on the processors read the trace data directly and analyze it in parallel, enabling fast analysis. The host analyzer gathers the results from the processors and displays it in an X Window.

The performance analyzer displays the following information:

1) Tabular statistical information (minimum, maximum, and average)
   i) Event and time
   ii) Times for execution, interrupts, and idling
   iii) Communication distance
   iv) Transferred message size
   v) Network activity ratio
   vi) Time for message and barrier synchronization wait
2) Graphs
   i) Processor activities
   ii) Overhead (e.g. idle time, communication time)
   iii) Processor status
   iv) Message information (e.g. data amount, communication distance)

3.2.1 Trace data

We use two kinds of events for trace data. The first type includes the start and end of communication library use, the start and end of communication interruption, and task switching.

The second event is mainly used for message communication. When a message is sent, the following are recorded: the destination processor ID and task ID, type, size, and source processor ID and task ID of a message and the time the message was sent. When a message is received, the following are recorded: the source processor ID and task ID, type, size, and destination processor ID and task ID of a message and the time the message was received and the message waiting time. Barrier synchronization events and their times are also recorded.

3.2.2 Graphic display

A graphic representation of execution performance can provide a greater insight than tables of statistics. Figure 5 shows a performance analyzer display. This display is for a parallelized LINPACK benchmark program that was executed using 512 processors. The execution took 1.081 seconds. The distribution of the execution, idling, and communication time is 50.97, 22.61, and 26.42 percent, respectively.

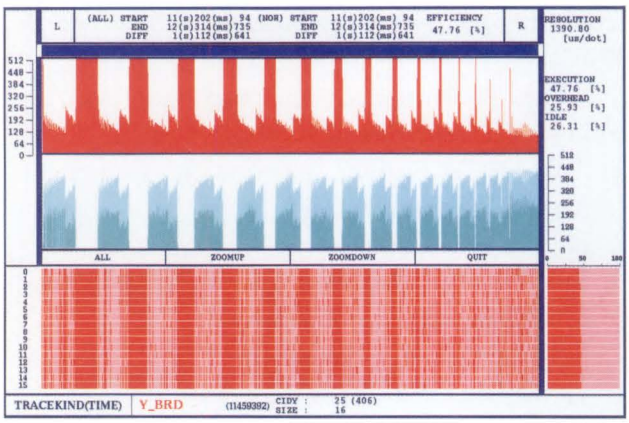


Fig. 5 – Performance analyzer display.

In this display, the graphs in the three main windows use the same scale for the horizontal time axis and show (from top to bottom) the number of active processors, the number of idling or communicating processors, and the status of each processor at each execution step. The number of active processors drops dramatically at fairly regular intervals because the program uses a blocked Gaussian elimination method. This display clearly illustrates the behavior of a program.

In the middle graph, idling processors are shown in dark blue and communicating processors are shown in light blue. In the bottom graph, various colors are used to indicate the status of each processor, for example, execution, communication, or interrupt. The example shown in Fig. 5 shows the status of only 16 processors. The graph to the right of the status window shows the total amounts of time each processor spends in each status.

The bottom window shows event information. This information is specified by clicking on the status window. In this example, the trace event is Y_BRD, in which processor 25 broadcasts a 16-byte message along the row axis. Users can change the time scale of the display.

Message information can be displayed in the overhead window. Figure 6 shows the display for the size of messages in each transfer. This display shows how the message size decreases as the decomposed matrix becomes smaller.

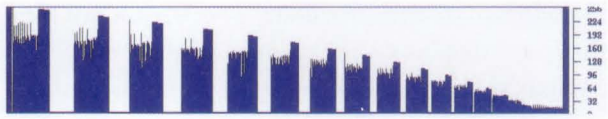


Fig. 6 – Message size.

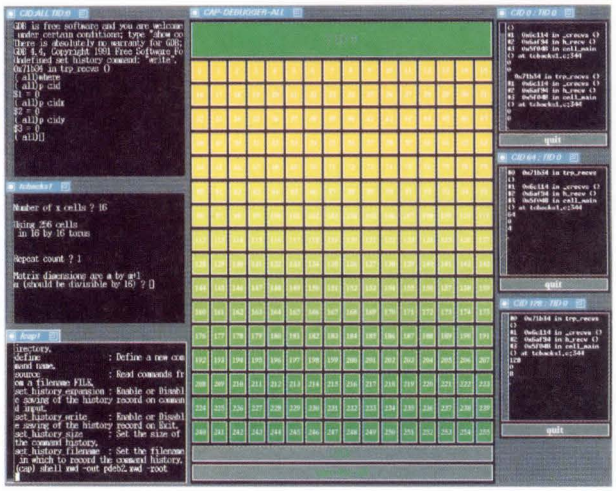


Fig. 7 – Parallel debugger.

3.3 Parallel debugger

The source debugger we developed for debugging tasks running in parallel is implemented using the remote facilities of GNU gdb, so all GNU gdb functions are accessible. In addition to debugging programs running on one processor, the parallel debugger can be used to set breakpoints and display the values of variables simultaneously for multiple processors. The command for the parallel debugger is the same as that for the debugger of a single processor. The parallel debugger has two modes. In the first mode the debugger commands are effective for all processors. In the second mode the commands are effective only for one specified processor.

The parallel debugger displays the values of a specified variable and the point where the program stops. Figure 7 shows an example display of the parallel debugger. The boxes in this display represent the processors, and the colors of the boxes indicate the values of the specified variable for each processor. The data maximums are green and the minimums are yellow. If all data sets except one are the same, the odd one out is clearly distinguishable.

The WHERE command prints a backtrace of the entire stack. If all processors stop at the same point, the color of each box is the same.

4. Conclusion

This paper described the AP1000 software environment. The parallel programing interface of the AP1000 is based on message transfer and allows application programmers to easily write parallel programs.

The runtime monitor shows the processor status and load during execution to enable program debugging and performance tuning. The performance analyzer analyzes program execution statistics and presents a graphic display of program behavior during execution using traces recorded during application execution. The parallel debugger is used to debug programs running simultaneously on multiple processors. The parallel software simulator, CASIM, supports parallel software development for the AP1000. CASIM runs on workstations and simulates the AP1000 parallel library.

The AP1000 provides a high-speed processing environment for applications and research.

References

1) Malony, A. D., and Reed, D. A.: A Hardware -Based Performance Monitor for the Intel iPSC/2. Proc. Int. Conf. Supercomput., 1990, pp. 213-226.
2) Malony, A. D., Reed, D. A., and Rudolph, D. C.: "Integrating Performance Data Collection, Analysis, and Visualization". *Parallel Computer Systems: Performance Instrumentation and Visualization*, Addison-Wesley Publishing Company, 1990.
3) Malony, A. D., Reed, D. A., and Wijshoff, H.: "Performance Measurement Intrusion and Perturbation Analysis". Tech. Rep. CSRD No. 923, Univ. of Illinois, 1989, pp. 1 -35.
4) Sharma, S., Malony, A. D., and Berry, M. W.: Run-Time Monitoring of Concurrent Programs on the Cedar Multiprocessor. Proc. Supercomputing '91, 1991, pp. 784 -793.
5) Rover, D. T., Carter, M. B., and Gustafson, J. L.: Performance Visualization of SLALOM, Proc. 6th Distributed Memory Computing Conf., 1991, pp. 543-550.
6) Couch, A. L., and Krumme, D. W.: Multidimensional Spreadsheets in a Graphical Symbolic Debugger. Proc. 6th Distributed Memory Computing Conf., 1991, pp. 205-208.
7) Arlauskas, R.: iPSC/2 system: a second generation hypercube. Proc. 3rd Conf. Hypercube Concurrent Comput. Appl., 1988, pp. 38-42.
8) Lillevik, S. L.: The Touchstone 30 Gigaflop DELTA Prototype. Proc. 6th Distributed Memory Computing Conf., 1991, pp. 543 -550.
9) nCUBE2 Supercomputers: Technical Overview, 1990.
10) Ishihata, H., Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Processing, 1991, pp. 13 -16.

Takeshi Horie

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
The University of Tokyo 1984
Master of Electrical Eng.
The University of Tokyo 1986
Specializing in Computer Architecture and Parallel Computing

Morio Ikesaka

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Mathematical Eng.
Kyoto University 1978
Master of Mathematical Eng.
Kyoto University 1980
Specializing in Parallel Processing

UDC 681.326.7

Towards Debugging and Analysis Tools for Kilo-Processor Computers

● Paul Brian Thistlewaite ● Christopher William Johnson

(Manuscript received October 12, 1992)

The new class of MIMD computers with a thousand or more processors (*kilo-processor machines*) is over-reaching the available program development support tools. One such machine is the Fujitsu AP1000. The LERP project at the Australian National University aims to assit the AP1000 programmer with essential monitoring and debugging tools. LERP is based on off-line analysis and replay of an event trace which includes both low level system events and user-defined events. In this paper we address the issues of extracting humanly usable information from large event traces with a new compound event analysis tool, suited to machines of the AP1000 scale.

1. Introduction

The Fujitsu AP1000 is a pioneering member of a class of MIMD wormhole message-passing machines with many hundreds of processors (each of which is a general purpose microprocessor with significant speed and memory in its own right), connected by three communications networks[1]. We may term this class the "kilo-processor" machines[Note]. It provides significant challenges to programmers who need to develop correct and efficient multiprocessor parallel programs. As it is a new machine, it lacks extensive software debugging aids for the programmer. The LERP project at the Australian National University aims to assist the programmer by producing the essential monitoring and debugging tools for the AP1000.

Traditional debugging methods that are used on sequential machine programs are less effective on multiple processes and distributed memory. The usual methods for parallel debugging fall into three classes: *playback*, *breakpoint*, and *static analysis*[2]. LERP is based on off-line analysis and replay of an event trace. Preliminary stages in its design are described in previous papers[3,4].

Most parallel debuggers and performance analysers are basically visualisation aids: they present an animation or complete timeline view of one or many aspects of program execution such as message passing events, processor busy/idle state, etc. {see UPSHOT and PICL/PARAGRAPH, for example[5-7]}. To make use of them the programmer must replay or display the program behaviour, possibly filtered, and extract anomalous or significant behaviours from the animation by observation. The tools become markedly less effective as the number of processors rises past 30 or so: the graphics devices and the human eye both become overloaded, as does the human ability to integrate and differentiate time-based behaviours or spatial patterns.

An alternative model of debugging is provided in the Belvedere system[8], which expects the user to specify expected behaviours in a formal Event Description Language[9]. It is apparent that many programmers do not have such a formal view of their program's expected

This work is supported by the ANU-Fujitsu Joint Research Agreement.

Note: The relationship to the term "killer-micros" to describe these machines is not accidental.

behaviour, and that such facilities are not useful to them. On the other hand, a description of program behaviour in these terms is a useful start to debugging and understanding the program's actual behaviour. In this paper we describe a method for extracting significant compound events automatically, with acceptable efficiency, which has application to useful classes of programs.

Tools for debugging and performance monitoring and analysis exist to aid programmers in the task of forming and testing hypotheses about the program's behaviour. Kilo-processor machines present the specific problem of compressing the information that is available from the machine to humanly usable quantity. By attaching existing conventional debugging and monitoring tools to the AP1000 we are able to support part of the debugging activity, but at the same time we demonstrate these tool's inadequacy for large numbers of processes. Debugging on such machines requires extending the abstractive power of the tools further along the spectrum from detailed internal state examination (**dbx**-like), and beyond internal and external state transition and message flow (like PARAGRAPH and UPSHOT). The LERP compound event recognition tool, and the associated ability to cluster process behaviours, are an advance along this spectrum that provides the user with a more abstract gestalt view of the program's behaviour.

The problem is not only in the large number of processes but in the shift in viewpoint that in needed, from considering internal program states alone in sequential debugging, to seeing both internal and external states in distributed parallel programs, up to broader pictures of state changes and behaviour. Conventional debugging systems are unable to handle the larger sequential programs of today, such as a complete X-windows application, or layered OSI application, through all their layers, multi-parameter interfaces, and semi-hidden internal states. It is not surprising that they are also unable to cope with the kilo-processor.

Our answer is to provide two kinds of selective focus to the user:

1) *focus in time or space* taking subsets of the execution timeline, and of the set of tasks;
2) *defocusing on detail* abstracting whole program behaviour to reduce the information load.

Programmers use several layers of abstraction in constructing their programs, both explicitly represented in higher-level language constructs and implicitly contained in their coding. The compound event recogniser is the first stage in attempts to capture some abstract patterns in program behaviour and present them in ways that the users can relate to their designs. The alternative of requiring programmers to specify formal event descriptions ahead of time is unproductive: programmers rarely formulate their designs precisely enough to be able to give exact specifications of the expected behaviour. By viewing behavioural abstractions extracted from actual behaviour programmers can be alerted to discrepancies between processors, and can gain new understanding of their programs to help them refine their algorithms and implementations.

2. Trace collection

LERP provides a common event trace format for both low level system events and user-defined events. The event trace makes no assumptions about the style of programming: each event is associated with a particular task and cell. Every event logged in the trace includes a cell-based timestamp, the type of event (send, receive, broadcast, configure, synchronise, user-event etc.), and in some cases the identifier of a second cell that participates in the event (e.g. a receive event also records the sender). We have implemented event trace collection by modifying the AP1000 cell operating system, both in the kernel and in the library code. Events are logged into cell memory for selected operating system traps. The accumulated event log is sent to the host for collection into a combined disk file, either at the end of processing or at synchronised regular intervals during execution.

The events that are traced are those we term *external* to a process: sending or receiving

a message, synchronisation, network configuration, etc. A typical program's event trace thus includes some tens or hundreds of events per processor per second.

LERP presents traces to the user through a growing set of both graphical and text-based tools, controlled by a common graphical driver. Selective focusing on subranges of time and of processes will be by user-selected filters on the trace. The tools include both static analysis of the execution that is recorded in the trace, and some animated tools that allow the user to visualise the flow of events, message traffic, and computation load, over time.

The static tools include *virtual channel analysis* of message traffic and *compound event recognition* to reduce the trace to a humanly manageable volume of meaningful information. This volume reduction can be achieved for traces both within individual tasks, by replacing sequences of atomic events with derived simple regular expressions, and within a collection of tasks, by imposing appropriate equivalence relations on the collection. The dynamic replay tools include our own replay debugger.

A number of event trace analysis tools are available for other parallel computing systems. In the case of two existing tools, PICL/PARAGRAPH and UPSHOT, the program model is close enough to the AP1000 that it is simple and worthwhile to interface them to the AP1000 event trace format. The interaction of these tools is described elsewhere[10].

3. Compound event trace analysis

3.1 Motivation

Existing tools such as PARAGRAPH and UPSHOT are ineffective for viewing or analysing logs of parallel programs with a large number of tasks/processes. Indeed, many existing tools have soft-wired limits to the number of tasks/processes that can be viewed (for example, 16). Moreover, an extensive search of the literature together with discussions with a number of other researchers in the area of parallel program debugging and performance monitoring indicates that few researchers have considered the problems associated with massively parallel architectures where hundreds or even thousands of processors might be involved – the kilo-processors.

When the number of processors exceeds some small number the user of a parallel debugging or performance monitoring tool faces the problem of "bandwidth", or of "not being able to see the forest for the trees" or *vice versa*. The mass of information makes it easy to miss important individual events (e.g., anomalous or pathological events), and makes it hard to see an overall execution pattern (e.g., topology) amongst the processes that make up the program.

Symbolic debuggers are especially useful for understanding the exact nature of an anomaly, and in the case of exception-induced process termination (e.g., fatal errors), the approximate location of an anomaly. But in the case of parallel systems, especially message-passing architectures, non-fatal or indirectly fatal errors can reside in processes other than the one that terminates with an error condition.

Animation tools provide approximate, impressionistic depictions of "patterns" within and between event streams, but do not provide pattern descriptions which exactly describe event streams. Human visual processing can detect only a few of these patterns; the human must be assisted by automatic pattern recognition.

Such exact descriptions can be used for the deduction of similarities between processes and parts of a single process, and these can lead, in turn, to the detection of anomalous individual events within a stream, the analysis of the topology of a program, and the selection of a manageable number of "representative" tasks for closer examination.

3.2 Compound event recognition

3.2.1 Algorithm

In a LERP log the events that constitute a particular processor / task event stream are dispersed throughout the log. However, for the purposes of compound event recognition, we will separate individual processor / task event streams, and we will order events within a

particular stream according to time.

Assume then that we have logged a program using p processors each having t tasks per processor giving a total of $n = p \times t$ separate event streams. Each event stream, n_i, is a time-ordered sequence of events. Where $n_j = <e_j^1, \cdots, e_j^k>$, we can now define a *compound event* as a complete subsequence of an event stream, $<e_j^c, \cdots, e_j^{c+d}>$. A *compound event* is a structure having the following fields:

1) a *type-identifier*, which maps to the sequence of primitive event types that constitute that subsequence
2) a *start-time*, being the timestamp of the event e_i^c
3) an *end-time*, being the timestamp of the event e_i^{c+d}
4) a *count*, the meaning and use of which will be explained later.

Note that in our terminology a compound event does not span or relate events from different tasks (e.g., a SEND from task A and its matching RECEIVE in task B).

However, we do not want just any arbitrary subsequence of an event stream to constitute a compound event. Ideally, we want to select those subsequences that correspond, in some meaningful sense, with actual "modules" or "blocks" within a task; for example, a SEND followed by an ACKNOWLEDGEMENT followed by a CUE. In the absence of clues provided by the user, we can use two criteria for determining which subsequences form natural blocks:

1) the subsequence recurs frequently within a particular event stream, or between different event streams
2) the subsequence is bounded by another compound event, or lies exactly between other compound events, or occurs repeatedly so as to indicate a loop.

In addition, we would like to recognise larger sequences as compound events before smaller sequences, everything else being equal.

These considerations lead to assigning a value to every subsequence of every event stream, where this value is determined by the formula:

$$Value = (\textit{total frequency of occurrence}) \times \log_{10}(\textit{length of subsequence}) \times \textit{repeat\_count}$$

where the repeat_count is the number of times the subsequence is immediately followed in an event stream by an instance of itself.

Compound event recognition proceeds by identifying subsequences of events in each task's event stream, between specified length limits, and computing *adjacency counts* for repeated adjacent instances of subsequences. The highest *Valued* subsequence is chosen as a compound event, given the start- and end-times of its first and last constituent events, and subsequence identification is repeated after substituting a compound event and count, for the constituent subsequences. This is called *compaction*. So, the algorithm for selecting a subsequence as a compound event can be stated as follows:

1) partition the LERP event log into separate event streams for each processor / task
2) scan each stream, identifying subsequences of events having length, L, between certain minimum and maximum values
3) calculate the frequency of occurrence for each such subsequence within all streams; call this value A
4) calculate the number of times each subsequence occurs adjacent to itself; call this value B, the *adjacency value* of a subsequence
5) assign a value, V, to each subsequence, equal to $A \times B \times \log_{10}(L)$
6) select the sequence, $S = <e_1, \cdots, e_n>$, with the highest value of V as a compound event C_i
7) replace all occurrences of S within all event streams by a single event, of a new *event-type* C_i, and assign a *start-time* being the timestamp of the first event, e_1 in this instance of subsequence S and an *end-time* being the timestamp of the last event, e_n, in this instance of S
8) if an occurrence, S_i, of S is immediately followed in a sequence by another occurrence of S, S_j, then (i) increment the *count* field of S_i, (ii) replace the *end-time* of S_i with the *end-time* of S_j, and (iii) delete S_j. Let us call this process *compaction*.

This algorithm is repeatedly applied to successively refined event stream sequences, and a new highest-valued subsequence is selected as the next compound event for substitution and compaction within the event stream sequences. Note that, because of compaction, eventually an iteration of the algorithm will fail to find any subsequence with a non-zero adjacency value. Consequently, at this iteration, all values of V for all subsequences will be 0, and we can terminate iterating the algorithm.

At termination, we will have recognised a small number of compound event types. These compound event types represent frequent patterns of adjacent event subsequences.

Users can control the efficiency and power of this process by setting parameters such as minimum and maximum subsequence length, whether compound events can themselves be compounded, or can only be a compound of primitive event types, and terminating when a given number of the best compound events have been found.

An example of two primitive event traces from tasks in the one program and the compound events that are recognised by this algorithm is in Fig. 1. Note that the timestamp values have been idealised.

3.2.2 Implementation details

Scanning an event stream for all subsequences of lengh between *min-length* and *max-length* is computationally expensive, but for all practical purposes can be done for actual LERP logs in reasonable time (e.g., less than 5 seconds per scan in a 1.5 Mbyte LERP log). These speeds are achieved by keeping all subsequence data (such as the list of primitive event types in the subsequence, the frequency of occurrence, the adjacency value, the length and the merit-value) in hashed records, where the hash value is a function of the primitive event types in the subsequence.

Primitive event sequence-Cell 2		Compound event sequence-Cell 2
Event	Time	
Iinfo	0001	
Cue	0004	Event : CE2 = (Iinfo, Cue,
Recvs	0006	Recvs, Recvd,
Recvd	0012	Cue, Sync, Cue)
Cue	0015	Start : 0001
Sync	0030	End : 0035
Cue	0035	Count : 1
Fsend	0100	
Ack	0110	
Recvs	0134	
Recvd	0178	
Ack	0213	
Fsend	0311	Event : CE1 = (Fsend, Ack,
Ack	0320	Recvs, Recvd, Ack)
Recvs	0349	Start : 0100
Recvd	0401	End : 0791
Ack	0429	Count : 3
Fsend	0498	
Ack	0523	
Recvs	0538	
Recvd	0556	
Ack	0791	
Pstat	0923	Event : CE3 = (Pstat, Ack,
Ack	0932	Csend, Ack, Exit)
Csend	0955	Start : 0923
Ack	1020	End : 1093
Exit	1093	Count : 1

Signature : CE2, CE1(3), CE3

Primitive event sequence-Cell 6		Compound event sequence-Cell 6
Event	Time	
Iinfo	0002	
Cue	0004	Event : CE2 = (Iinfo, Cue,
Recvs	0006	Recvs, Recvd,
Recvd	0012	Cue, Sync, Cue)
Cue	0015	Start : 0002
Sync	0030	End : 0038
Cue	0038	Count : 1
Fsend	0105	Event : CE1 = (Fsend, Ack,
Ack	0110	Recvs, Recvd, Ack)
Recvs	0134	Start : 0105
Recvd	0178	End : 0218
Ack	0218	Count : 1
Fsend	0317	Event : CE4 = (Fsend, Ack,
Ack	0320	Recvs, Recvd, Cue)
Recvs	0349	Start : 0317
Recvd	0401	End : 0428
Cue	0428	Count : 1
Fsend	0493	Event : CE1
Ack	0523	Start : 0493
Recvs	0538	End : 0797
Recvd	0556	Count : 1
Ack	0797	
Pstat	0923	Event : CE3 = (Pstat, Ack,
Ack	0932	Csend, Ack, Exit)
Csend	0955	Start : 0923
Ack	1020	End : 1094
Exit	1094	Count : 1

Signature : CE2, CE1, CE4, CE1, CE3

Fig.1 – Corresponding primitive and compound events (idealised timestamps).

3.3 Generalised compound events

A simple extension of the compound event recognition capabilities permits some recognition and analysis of compound events that span across task streams, given some indication of the expected geometry of interprocessor communications. The technique involves extending the number of types of primitive events by sub-categorising existing primitive event types. For example, rather than just have a CSEND and an ISEND event type, we identify the source and destination of the message within the event type as a new type such as SEND (**s_type**, **from**, **to**), where **s_type** is the type of send (e.g. CSEND or ISEND), **from** is the task identifier for the sending task and **to** is the task identifier for the receiving tasks (similarly, for receive event types).

The user can then specify whether SENDs with different **s_type** values should be distinguished as different event types or not, whether the **from** or **to** fields should distinguish, and if so, whether on the basis of the **from** and **to** values taken literally or via some mapping based on a user-supplied topology {e.g. in an 8×82 dimensional mesh topology, both SEND (CSEND, 3, 11) and SEND (ISEND, 6, 14) become SEND (Any, Self, South)}. Many programs use one of a small set of topologies (linear, ring, mesh, torus, hypercube) that make it feasible to use an enumerative approach to specify this; the CCONFIG event in the AP1000 system contains additional indication of the mesh dimensionality, for instance, and we may add further information about other significant structure (such as that column or row relationships matter in the program, or that some subset of processors can be regarded as equivalent, e.g. Tuple server cells in a distributed Linda implementation).

3.4 Partitioning tasks into equivalence classes

After recognising compound events, we have reduced the event stream for each task to a more compact, yet exact, description of the original, by substituting compound events for large subsequences of other events, and compacting adjacent instances of a compound event into the one record.

If we ignore the time information in each event stream, and explicitly tag each occurrence of a compound event with its count, the resulting event stream description is a regular expression representing the sequence of event types of events in the original event stream. Let us call such a description the *signature* of an event stream for a given task. Signatures for the two tasks are shown underneath Fig. 1.

Now, we can partition the set of tasks associated with a particular program using equivalence relations defined in terms of task signatures. Where two tasks have identical signatures, we call them identical tasks. We can also ignore the count tag in the signatures, thereby getting a description which we call the *reduced signature* of a task. Where two tasks have identical *reduced signatures*, we call them *cognate tasks*. Clearly, these identities give rise to equivalence classes.

In the example there are two equivalence classes of identical tasks among the whole group of tasks (the signatures of the other tasks were not shown above):

SIGNATURE for cells
0···5, 7···13, 15···21, 23···29, 31···37,
39···45, 47···53, 55···61, 63
: CE2 CE1 (3) CE3

SIGNATURE for cells
6, 14, 22, 30, 38, 46, 54, 62
: CE2 CE1 CE4 CE1 CE3

Having automatically partitioned all tasks into equivalence classes, the user can:

1) determine whether the topology induced by the set of equivalence classes corresponds to the intended topology.
2) identify tasks which may have anomalous events by examining singleton equivalence classes.
3) examine the signature for a task or set of tasks to see whether the "block" structure appears to reflect the intended substructuring of the task.
4) select a subset of tasks, typically one or two from each equivalence class, for viewing using available tools (such as UPSHOT or

those provided via PARAGRAPH), or for examining using a symbolic debugger.

In our example the topology of the identical tasks corresponds to a single column of an 8 × 8 cell array, as shown here.

0	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	33	34	35	36	37	**38**	39
40	41	42	43	44	45	**46**	47
48	49	50	51	52	53	**54**	55
56	57	58	59	60	61	**62**	63
						⇧	

The topology of processor classes is clearly that the column of cells 6, 14, 22, etc. behaves differently from the rest. We can offer no explanation in this case.

3.5 XCERT – an X-based tool for topology-specification, compound event recognition, and topology analysis

The capabilities discussed in the previous sections have been integrated into a single tool, XCERT (X-based Compound Event Recognition Tool).

XCERT is an X-based tool that enables a user to specify a LERP log and to graphically specify the topology that the user expected the program that generated the log to obey (known as the *expected topology*). Various run-time options for compound event analysis can also be set. Compound events are then derived, along with the topological equivalence classes that these lead to.

In the example outlined above the expected topology {2 dimensional, 8 × 8 configuration, relative addressed (mesh topology)} is indicated by a menu selection. Three selections are currently implemented in XCERT, each allowing a further choice of communications addressing:

1-dimensional	2-dimensional	Other

Relative	Absolute

The other topologies mentioned in Sec. 3.3 have not yet been implemented.

XCERT displays a graphic map of the cells in this topology. The configured size is obtained automatically by inspecting the event trace.

Selecting **Other** allows users to specify the expected topology by partitioning the cells into two or more classes. A user-defined short label can be given to each class for mnemonic purposes, and the display reacts to show the partitioning as the user develops it. In the ANU C-Linda implementation, for example[11], there are two classes of cells that are expected to behave quite differently: a small number of cells are chosen to act as "Tuple Space servers", the remainder as "evaluation servers". The XCERT user can select the five to eight Tuple servers as a class marked **T**, the evaluation servers as another, marked **E**.

After the user has set the other run-time options mentioned in Sec. 3.3, the tool runs the CERT algorithm over the event trace. As well as displaying these derived compound events, task signatures and equivalence classes, XCERT displays the derived topology actually exhibited by the program. These equivalence classes of cells are displayed on an overlaid rendition of the graphic cell map, with the equivalence classes shown as colours. The user can compare the derived and expected topologies visually, and reset the CERT parameters to run further analyses if necessary. The resulting matches – and mismatches – are a valuable debugging aid to understanding actual program behaviour. Where the derived topology does not match the expected the user can turn to other tools in the LERP suite to investigate the anomaly.

As well as displaying these derived compound events, task signatures and equivalence classes, XCERT displays the derived topology actually exhibited by the program. It does this by graphically overlaying the *derived topology* over the expected topology, thereby allowing the user to easily compare expected program behaviour with actual program behaviour.

3.6 Determining phases of a process

Reduced signatures, to the extent that they each reflect a natural, abstracted "block"

structure within each task, also reflect the computational "phases" of that task, with phase boundaries corresponding to abstracted compound event boundaries. With cognate tasks, these phases will be common. We can view the phases of a class of cognate tasks (or, indeed, of all tasks) by representing a particular phase as an UPSHOT state, and inserting dummy events into the log to mark the boundaries of each phase.

4. Conclusion

Future directions for this work will extend the approach that we have taken in providing a multi-layer, multi-view toolset, with automatic behavioural analysis, guiding manual selective focusing. We identify the need to include the whole program view in debugging and analysis: reductionism will not be successful in kilo-processing.

Extensions planned in the near future include:

1) recognising and highlighting clustering in patterns of decomposition and event behaviour
2) discriminating between otherwise similar compound events on the basis of significantly different duration over time (classical Clustering Analysis techniques can then be applied).
3) determining equivalence classes of tasks, and the phases of tasks, using signatures and reduced signatures, more generally by taking relations other than identity. This will require more powerful pattern analysis methods such as local optimisation (e.g. using simulated annealing techniques); some of the techniques of Genome Analysis can be applied. Certain types of events (such as SYNC) can thus be given more significance than others in an inexact comparison of similarity and in the derivation of computational phases.

References

1) Horie, T., Ishihata, H., Shimizu, T., Kato, S., Inano, S., and Ikesaka, M.: AP1000 architecture and performance of LU decomposition. Proc. 1991 Int. Conf. Parallel Processing, **vol. 1**, C. Wu, ed., CRC Press Inc, Boca Raton, 1991, pp. I-634-635.
2) McDowell, C. E., and Helmbold, D. P.: Debugging Concurrent Programs. *ACM Computing Surveys*, **21**, pp. 593-622 (1989).
3) Johnson, C. W., and Mackerras, P.: Architecture of an Extensible Parallel Debugger. Proc. 1991 Int. Conf. Parallel Processing, **vol. 2**, H. D. Schwetman, ed., CRC Press Inc, Boca Raton, 1991, pp. II-262-263.
4) Johnson, C. W., and Mackerras, P.: Design of a Replay Debugger for a Large Cellular Array Processor. Proc. Australian Software Engineering Conference, 1991, pp. 189-201.
5) Herrarte, V., and Lusk, E.: Studying program behavior with upshot. Math. Comput. Sci. Div. Argonne Natl. Lab., Illinois, 1991.
6) Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H.: A users' guide to PICL: a portable instrumented communication library. Oak Ridge Natl. Lab., Tennessee, ORNL / TM-11616, 1990.
7) Heath, M. T., and Etheridge, J. A.: Visualizing performance of parallel programs. *IEEE Software*, **8**, 9, (1991).
8) Hough, A. A., and Cuny, J. E.: Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. Proc. 1987 Int. Conf. Parallel Processing, University Park, Pennsylvania, 1987, pp. 735-738.
9) Bates, P. C., and Wileden, J. C.: EDL: A Basis for Distributed System Debugging Tools. Proc. 15th Hawaii Int. Conf. Syst. Sci., 1982, pp. 86-93.
10) Johnson, C. W., Thistlewaite, P. B., Walsh, D., and Zellner, M.: Developing monitoring and debugging tools for the AP1000 array multiprocessor. Proc. 2nd Fujitsu-ANU CAP Workshop, R. P. Brent, ed., CAP Workshop Organisation, Australian National University, Canberra, 1991, pp. C-1-C-14.
11) Cohen, R., and Molinari, B.: Implementatoin of C-Linda for the AP1000. Proc. 2nd Fujitsu-ANU CAP Workshop, R. P. Brent, ed., CAP Workshop Organisation, Australian National University, 1991, pp. T-1-T-10.

Paul Brian Thistlewaite

Department of Computer Science
Australian National University
Bachelor of Arts
University of Queensland 1978
Ph.D. in Computational Logic
Australian National University 1984
Specializing in Information Systems Design

Christopher William Johnson

Department of Computer Science
Australian National University
Bachelor of Science
Monash University 1973
Ph.D. in Computer Science
Australian National University 1983
Specializing in Software Development, Environments and Programming Languages

UDC 681.3.02

Parallel Visual Computing on the AP1000: Hardware and Software

● Hiroyuki Sato ● Satoshi Inao ● Hideaki Yoshijima

(Manuscript received October 9, 1992)

To use the highly parallel AP1000 computer in visual computing, we developed video hardware having a 400 Mbytes/s peak transfer rate and the ability to display HDTV images generated by processors at 30 frames/s, and parallel disk hardware that uses a 3.5-inch disk drive for each cell, providing a large capacity and high aggregate transfer speed. We also developed parallel visualization software CaVis, implementing it with a pipeline of task programs, each of which represents a visualization phase. We applied this to the visual simulation of molecular dynamics, global climates, and ray tracing.

1. Introduction

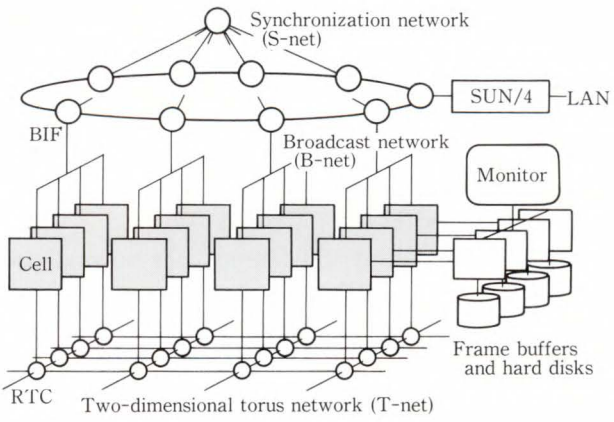
Large amounts of date are often best understood visually. "Scientific visualization"—a major topic of interest in supercomputing—involves converting multidimensional arrays of numeric data to visual images. Visualization cannot be accomplished efficiently with vector processing due to irregular data structures and complex calculations. Massively parallel computers such as the AP1000 should help speed up visualization, since parallelizing the computation and visualization tasks and running them concurrently on the same machine eliminates data transfer between the computing and visualization machines. Users can also interact with the programs more easily than when using separate systems.

To improve the AP1000's I/O performance and apply it to visual computing, we developed a video and parallel disk hardware option. The video hardware outputs HDTV-resolution images distributed in processor memory in real-time, and the parallel disk hardware processes large amounts of data at high speed. Visualization software CaVis uses parallel processing to visualize at high speed.

2. Video and parallel disk option hardware

The distributed-memory, highly parallel AP1000[1),2)] (see Fig. 1) can be configured from 16 to 1 024 cell processors. To take advantage of this parallelism, the I/O subsystems for the AP1000 should be scalable. When I/O subsystems are connected to a single cell processor or a front-end workstation, multiple cell access to I/O devices is serialized, often becoming a bottleneck. To avoid this, we designed a distributed I/O subsystem for image display and data access to disks.

For the video subsystem, image memory is distributed to processors, each of which directly



Cell : Cell processor RTC : Routing controller
BIF : Broadcast network interface

Fig. 1 – AP1000 architecture.

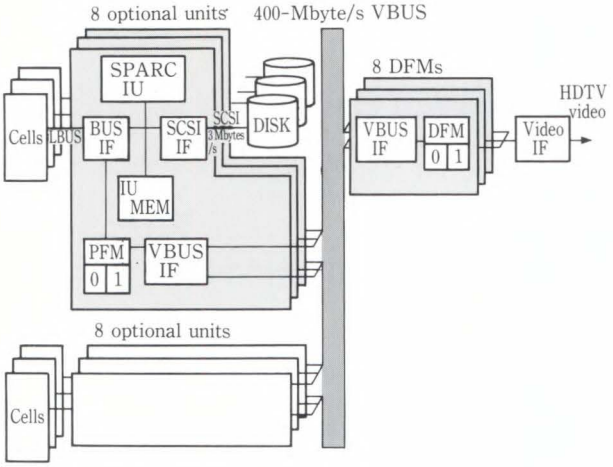


Fig. 2 – Video and disk option.

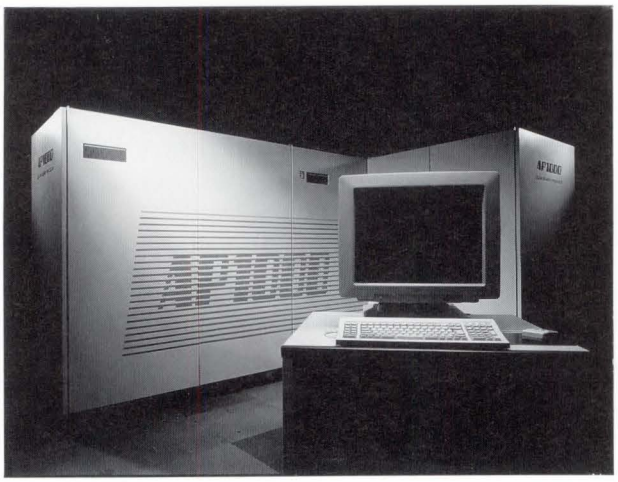


Fig. 3 – AP1000 with parallel disk frame.

accesses its own image memory. To output distributed image data to a high-resolution monitor, we designed special hardware for collecting images. Disks are also distributed to processors to obtain scalable data transfer speed.

The hardware we developed provides the basic AP1000 with:

1) Real-time output (30 frames per second) of HDTV-and NTSC-resolution data
2) Large-capacity, high-speed parallel disks
3) Animated display of consecutive image data partitioned and stored on parallel disks

The hardware consists of optional units, disk drives, and display frame memory (DFM) units (see Fig. 2). Each optional unit is connected to a cell via the cell's local bus (LBUS), a disk drive via an SCSI bus, and DFM units via the video bus (VBUS). Specifications of the video and disk hardware are listed in Table 1. Figure 3 is the AP1000 with the option hardware with a cell cabinet (right) and a disk cabinet (left). This basic cabinet set holds up to 128 cells and 128 optional units.

Table 1. Video and disk specification

Parameter	Specifications	
Display resolution (pixels)	NTSC	720×486
	HDTV	1 920×1 035
Video interface	NTSC	D-1 component digital RGBA component digital RGBAS analog
	HDTV	Component digital RGBA component digital RGBAS analog
Nunber of outputs	One or two	
Pixel gradation	8 bits per r, g, b, and a attribute (total of 32 bits/pixel)	
Display mode	Single/double buffer	
Display refresh rate (Mbyte/s)	PFM to DFM: 400 (peak)	
	Disk to PFM: 3 per drive	
Disk drivers (Mbyte)	500 each	
System configuration	16, 32, 64, 128, 256, and 512 options (# options must be less than or equal to # cells)	

2.1 Parallel disk

Each cell has access to a 500-Mbyte 3.5-inch hard disk. The SPARC integer unit (IU) in each optional unit controls the SCSI interface (SCSI-IF) for the drive. The maximum transfer rate is 3 Mbytes/s per drive. Data transfer is possible between a disk, the cell processor's main memory, partitioned frame memory (PFM), and IU memory.

2.2 Video

PFM contains image data generated by each

cell, which reads from and writes to its own PFM. The image distributed among the PFMs is written to display frame memory (DFM) via the VBUS. Each PFM consists of two 1-Mbyte banks of three-port memory used for double buffering. One bank is accessed by the cell and the other transfers image data to DFM. When only one bank is used, it acts as a single buffer.

The 8-channel, VBUS ring data bus transfers data between PFM and DFM. Each channel has a 50 Mbytes/s peak transfer rate. The VBUS has a maximum transfer rate of 400 Mbytes/s, which is fast enough to display HDTV resolution (1 920×1 035 pixels) images in realtime (30 frames/s).

The DFM has 16 Mbytes of memory divided among 8 groups, each consisting of two 1-Mbyte memory banks used for double buffering. One bank is accessed by the VBUS, and the other is used for display. The video interface provides analog and digital image signals in HDTV and NTSC formats.

2.3 Image display

The hardware image display is diagrammed in Fig. 4. Images are distributed to PFMs in vertical lines as shown in the figure. The Nth of P processors handles Xth vertical lines such that $N = X$ modulo P. In Fig. 4, P is 16. Processor 0 thus handles vertical lines of X addresses for 0, 16, 32,···. Processor N handles of N, $N + 16$, $N + 32$, ···.

The DFM consists of 8 channels. An image

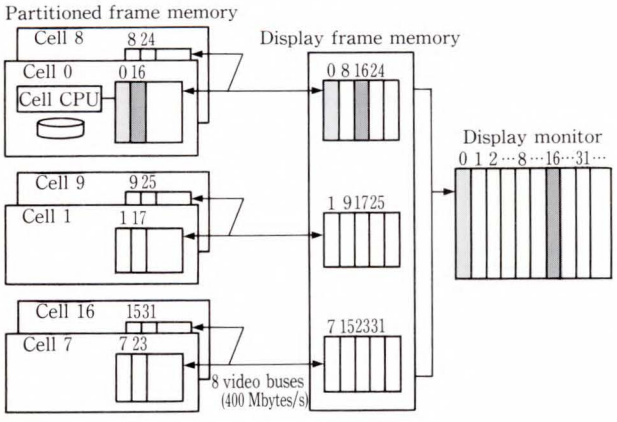


Fig. 4 – Hardware image display.

frame is distributed by vertical line division with fixed 8-line intervals.

Display is accomplished as follows.

1) Writing image to PFM

Each cell generates a part of the vertically divided frame image and writes image data to its own PFM. PFMs are double buffered with the foreground (writable) sides used for writing.

2) Data transfer from PFM to DFM

When a display request is issued by all processors, data is transferred in parallel from PFM to DFM using the 8 channels. In each channel, data is transferred from the first to the last processor in the VBUS ring. In Fig. 4, for example, cells 0 and 8 are grouped and connected in the channel 0 ring. Cell 0 first transfers data to DFM through the VBUS, then cell 8 transfers data. This is done simultaneously in all 8 channels.

3) Image display

After a new image frame is written to DFM, the foreground (writable) and background (readable) sides of DFMs are exchanged to display a new image frame. Each DFM has a first-in-first-out (FIFO) buffer. Image data is read from the DFM and written to these buffers. This data is then read in display sequence at the appropriate clock timing and multiplexed every 8 pixels. Output data is then converted to analog video signals.

4) Animated display using the parallel disk

Image data distributed among cells is stored on the parallel disk. The images are broken down into vertical lines in the same way as that for image display. Image data in each cell is written in continuous sectors on each disk to obtain the maximum I/O speed. When display commands are issued from all cells, IUs in optional units read image data from disks and transfer it to PFM using DMA. Images can then be displayed. This is done for successive image frames in animation. NTSC resolution has a 30-frame/s display rate with 16 options. HDTV resolution has a 15-frame/s display rate with 64 options, or a 30-frame/s rate with 128 options.

2.4 Basic video and parallel disk software

Software includes optional system, image,

and standard disk I/O libraries.

The optional system library is a collection of low-level functions not available to application programmers.

The image library is a set of functions called by application programs in cells. All cell programs are assumed to call the same function at the same time. Functions for displaying, saving, and loading image data are supported.

The standard disk I/O library is a set of UNIX-compatible I/O functions. Each cell program opens, closes, reads from, and writes to its own disk drive.

3. CaVis

The scientific visualization software called CaVis we are developing has the same set of programs residing in all cells and generates images in parallel. These images are displayed and stored on the parallel disk.

CaVis design objectives are:

1) High-performance visualization using data parallel algorithms

Visualization is speeded up by partitioning and distributing the data space to all the processors based on algorithms suited for massively parallel processing.

2) Creation of programs that are easy to develop, maintain, and use

Visualization programs tend to be complex and are difficult to use and maintain. CaVis uses modular programming[3] to minimize these problems.

3) Integration with computational programs

The visualization program is required to run concurrently with computational programs in all cells. Integrating computation and visualization makes it possible to construct interactive visual simulation systems where users control the simulation. Animation is also easier.

3.1 Program structure

3.1.1 Visualization process flow model

In visual computing, the numeric computation phase produces arrays of numeric values (see Fig. 5).

The data conversion phase converts numeric data to geometric primitives such as polygons and lines renderable using scanline conversion and a hidden-surface removal algorithm based on a depth buffer[4]. If the physical value at each grid point is a scalar, such as temperature, pressure, or density, numeric data is often converted to contour lines for two-dimensional field data. For three-dimensional data, displaying a cross section is the simplest way to visualize volume data. Another common method is to generate isosurfaces from a scalar field as proposed in Ref. 5.

3.1.2 Visualization task pipeline

We mapped the visual computing flow onto a cell task sequence which we call the visualization pipeline. This is somewhat similar to a UNIX command pipe. Data is fed to the first task, processed, and passed on to subsequent tasks (see Fig. 6).

The rendering phase is split into object and image tasks to render geometric objects using a z-buffer-based hidden-surface algorithm. Various visualization pipelines can be constructed using available tasks. Data transfer from one task to another sometimes include interprocessor communication.

3.2 Visualization pipeline implementation

The functions and implementation of individual CaVis tasks are explained in the sections that follow.

3.2.1 Conversion task

The conversion task produces geometric primitives such as vectors, polygons, and spheres

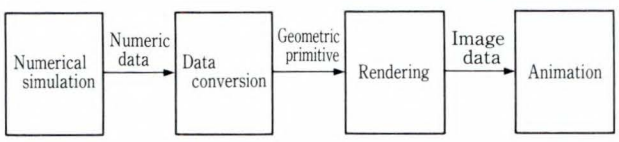


Fig. 5 – Visual computing.

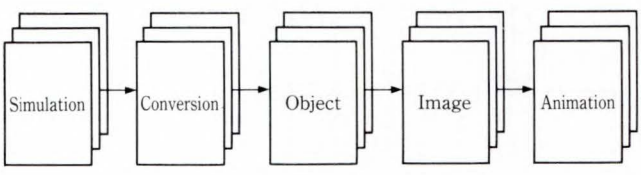
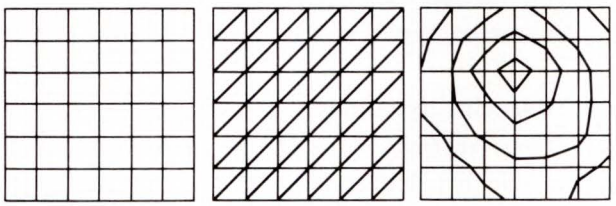


Fig. 6 – Example of visualization pipeline tasks.

to represent two- or three-dimensional field data with contour lines and isosurfaces. Two-dimensional scalar field data can be visually represented in many ways {see Fig. 7a)}. The conversion task supports the following:

1) Color: Triangles are formed from two-dimensional grids {see Fig. 7b)}. Color values are given to the vertices to be rendered in the object task. The pixels inside each triangle are produced by interpolating values at the vertices.
2) Contour line: Contour lines are generated{see Fig. 7c)}.
3) Filled contour line: The areas between contour lines are filled with a color.
4) Height field: Physical values are translated into height values. Each vertex coordinate of the triangles has an x and y coordinate plus a z value corresponding to its physical value.
5) Translucency: Physical values are translated into color and opacity values at the vertices. A lower physical value results in a lower color intensity and opacity value and reveals hidden objects when rendered with other objects.



a) Two-dimensional data b) Triangles c) Contour lines

Fig. 7 – Conversion task functions.

Isosurfaces for three-dimensional scalar field data are extracted using the marching cube algorithm[5]. Conversion task is designed so that unit data elements are processed independently. For two-dimensional array data, a rectangle with four corner-grid points is processed as a unit. For three-dimensional volume data, a voxel with eight corner-grid points is processed as a unit (see Fig. 8). This makes data distribution very flexible.

3.2.2 Object task

The object task generates subimages for geometric objects such as vectors, polygons, and spheres. The subimages are made up of pixels which have the attributes of color intensity, coverage, opacity (called an alpha value), and a z value, which is the distance between the eye and surface points corresponding to the pixel[6]. The coverage is an 8 by 4 bit mask which represents subpixel coverage of the object on that pixel and is used for antialiasing in the image task.

A CaVis task consists of a sequence of lower level parts called filters. These are similar to tasks but are implemented as C functions and pass data by function parameters.

The object task consists of many filters. The start and end filters have special functions related to data distribution. The optional selector filter at the top selects a part of the geometric primitive sent from either the previous task or the host process. When all geometric data is sent from a previous state, as when all data is broadcast from the host, this selector filter is used to select part of the primitives for each cell. Primitives are selected cyclically by processors, that is, when $N = O$ modulo P, the object is selected, where N is the processor

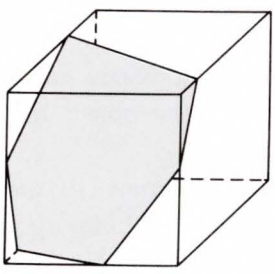


Fig. 8 – Example of an isosurface for a voxel.

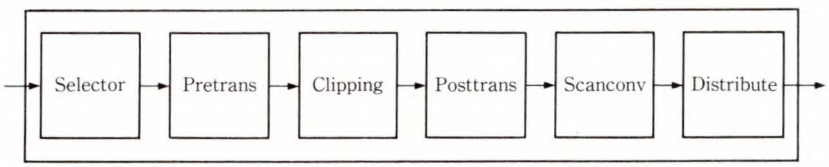


Fig. 9 – Filter pipeline in object task.

number, O the object primitive number, and P the total number of processors.

The subimages generated in the object task are divided into horizontal segments and are distributed among processors by horizontal scanline. The distribute filter in the object task buffers pixel data and sends it when the buffer becomes full or is flushed by an end-of-frame command. The torus network of the AP1000 is used to transfer data.

Intermediate filters are standard graphic pipeline functions for rendering geometric primitives such as polygons, vectors, and spheres[4]. The pretrans filter translates vertex coordinates into a normalized coordinate system before the clipping filter clips the screen windows. Coordinates of vertices are then transformed into screen coordinates by the posttrans filter. The scanconv filter then performs scan conversion with a shading calculation for each pixel.

3.2.3 Image task

In the image task, hidden-surface removal, translucent blending between surfaces, and antialiasing is done pixel by pixel using an alpha buffer algorithm[6].

There are many possibilities for dividing image space (see Fig. 10). CaVis uses either horizontal or vertical lines for the images because it is easy and effectively balances the load and minimizes computation cost. When video hardware is used, the vertical line mode is used to suit PFM.

The image task stores pixel information in an alpha buffer. Pixel segment data, i.e., intensity, depth, opacity, coverage, and object number, is sorted by depth. After all objects are processed, the final visible intensity is calculated. When the pixel segment nearest the eye is opaque, that pixel segment is visible and its color intensity is output to the frame buffer. When pixel segments are translucent, they are merged with later pixel segments. Antialiasing takes into account the coverage of each pixel segment.

3.2.4 Animation task

The animation task controls displaying images stored on the parallel disk. A control panel on the front-end workstation window is used to interactively control animated display.

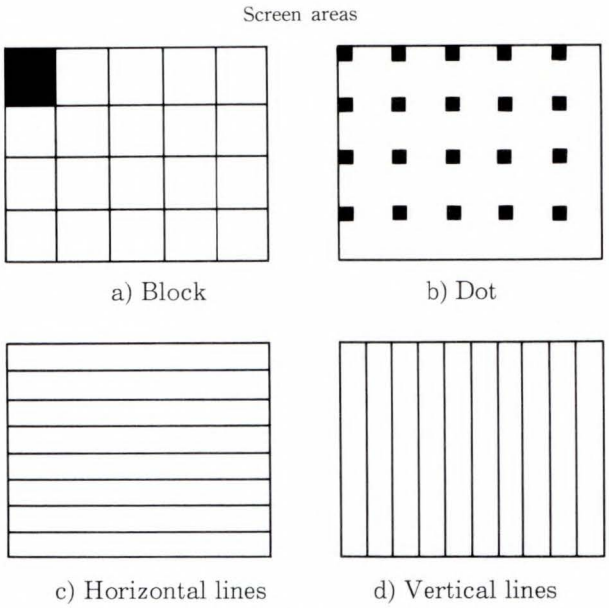


Fig. 10 – Examples of image space division.

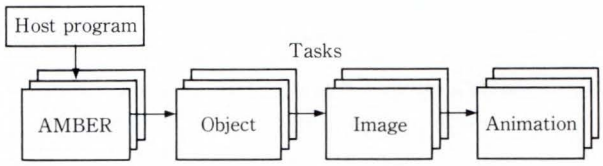


Fig. 11 – AMBER molecular dynamics simulation.

4. Application examples

Three examples of visualization follow.

4.1 Molecular dynamics

The AMBER[7] molecular dynamics program, which calculates the dynamics of protein molecules, was parallelized for the AP1000[8]. We used protein molecules soaked in water for test data. One of the objectives of the simulation was to study the protein dynamics at a microscopic level. Protein motion is observed by animation to understand its behavior.

The AMBER task calls CaVis to display atoms as spheres and bonds as vectors; object data is then fed to the object and image task pipeline (see Fig. 11).

AMBER uses particle division in which atoms are distributed randomly to processors for balancing the load. Each task receives data from the AMBER task of the same cell. Figure 12 is an example of a rendered image.

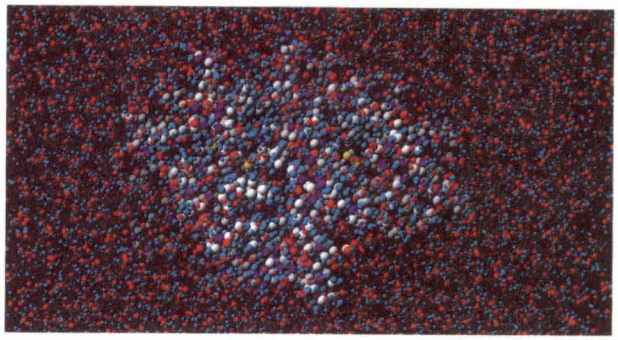


Fig. 12 – A rendered image of a lysozyme protein in water.

This model contains a human lysozyme with 130 amino acid residues (2 041 atoms) at the center of a pressure control cube (7.0 × 5.9 × 5.4 nm) filled with 6 208 water molecules.

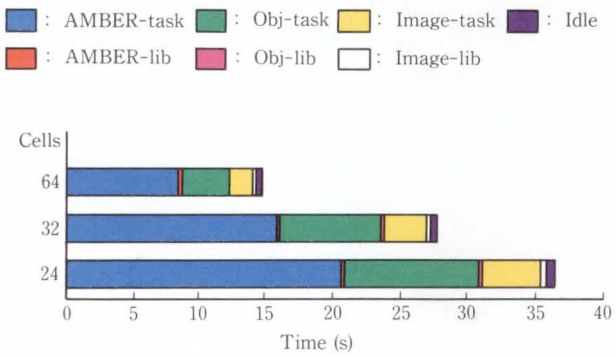


Fig. 13 – Visual simulation timing for molecular dynamics.

Figure 13 shows results for the visual simulation for AMBER. The time is for 5 steps of molecular dynamics calculation. A 1 920 by 1 035 pixel resolution image is rendered for each time step. All 20 665 atoms and bonds are rendered in each frame. Task and communication library run times are shown for each task. For example, amber-task time is the AMBER task run time and amber-lib time is the communication library time by the AMBER task. The null-task time shows the idle time. Overall task time is decreased by increasing the number of cells and the total performance is improved. Communication overhead is very small.

4.2 Global climate simulation

Global climate simulation was executed on the Fujitsu VPX240 vector computer and

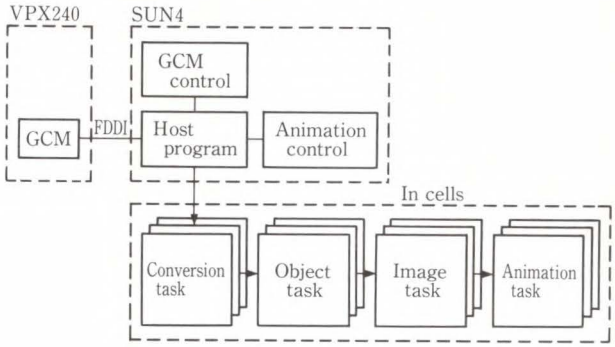


Fig. 14 – Global climate simulation.

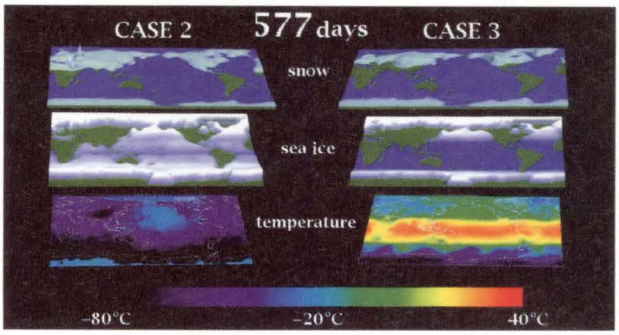


Fig. 15 – A rendered image of global climate simulation.

From the top, snow on the ground, sea ice, and the ground temperature are displayed. The results of two simulations with different initial conditions are displayed in one image frame. An animated sequence containing 577 HDTV image frames was made. The picture is the last image frame, after 19 months of simulation.

visualization was done on the AP1000 (see Fig. 14). The VPX240 and the AP1000 host workstation are connected via the FDDI link. The global circulation model (GCM) program and AP1000 host processes communicate via a UNIX socket interface. VPX240 simulation is monitored visually on the AP1000. The user controls the GCM program by issuing various commands from the SUN4 workstation.

The GCM program took 2.7 seconds for every one-hour step calculation on the VPX240. It took 2.9 seconds to transfer 1.7 Mbytes of data from the VPX240 to the SUN4 workstation. Using 64 cells, it took 1.2 seconds to render the earth's ground temperature distribution in a frame of 1 920 by 1 035 pixels with antialiasing. The time includes receiving data from the

Table 2. Timing results of ray tracing

Scene	Execution time (s) (Performance ratio)				
	SUN4/260	Transputer-32	AP1000-64	AP1000-512	Speed increase
Water drop 1	9 159.8 (1.0)	1 093.8 (8.3)	37.1 (246.9)	15.0 (611.0)	129.3
Water drop 2	5 421.1 (1.0)	956.8 (5.7)	98.9 (54.8)	19.4 (280.0)	305.2
Quark	113 872.9 (1.0)	6 780.4 (16.8)	1 095.1 (104.0)	155.2 (733.7)	437.5

Execution time includes loading geometric data and saving the image to the host.
The screen resolution is 1 024×1 024 pixels.

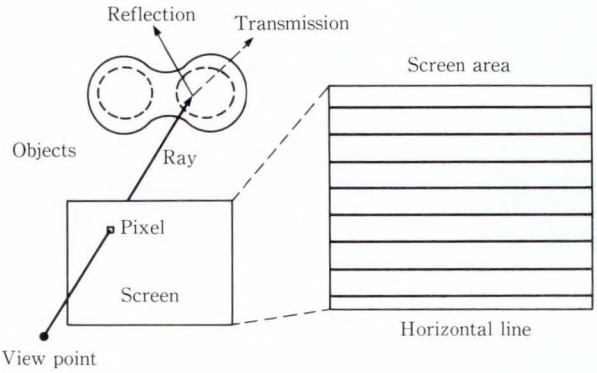


Fig. 16 – Ray tracing.

For each pixel on the screen, rays are traced along the directions from which the light comes. The first point on the object surface that the ray's strike is the visible point and the intensity for this pixel is calculated taking into account lighting, reflection, and transmission.

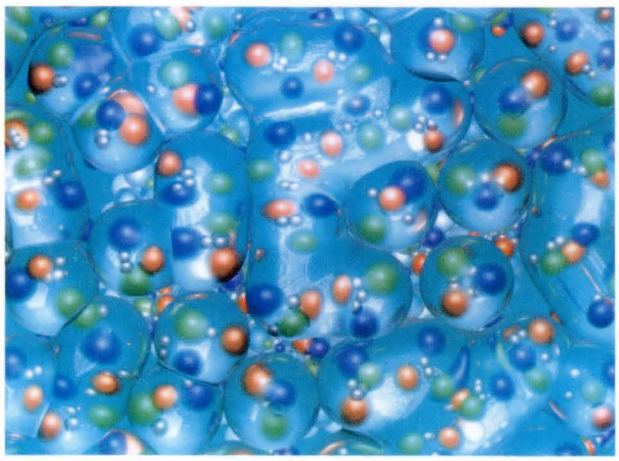


Fig. 17 – Quark scene generated by ray tracing.

The screen explains the creation of a "black hall". Nucleus of atoms are destroyed by strong gravity and split into quarks. Colored spheres represent a variety of "quarks".

workstation. A frame contains 7 200 triangles and 7 000 vectors. The VPX240 and AP1000 that were run in pipelined and ground-temperature images for one-hour step calculations were updated every 5.2 seconds when 64 cells were used.

Figure 15 is the result of global climate simulation using MRI-GCM.

4.3 Ray tracing

A ray tracing program developed by Fujita et al[9]. was ported to the AP1000. The screen space is divided in horizontal line mode (see Fig. 16). Table 2 lists the timing results of the ray tracing program for three examples. Figure 17 shows an image for a quark scene.

With 512 cells, the AP1000 performed ray tracing 280 to 734 times faster than the SUN4/260. The speed increase ratio, compared with a single processor, is 129.3 to 437.5 for the same scene.

5. Conclusion

The video and parallel hardware we developed enables high-speed image data output and efficient handling of large amounts of data. The data transfer speed between disks and cell processors is scalable with the number of disk-connected cells.

CaVis software speeds up visualization using parallel processing. Task pipelining provides a flexible way to build up task configurations. Two application examples, molecular dynamics and global climate simulation, were implemented with slightly different task pipeline configurations.

With the video and parallel disk hardware and CaVis visualization software, the AP1000 is well suited for visual computing. Parallel disk hardware is useful for any application which handles large amounts of data.

6. Acknowledgment

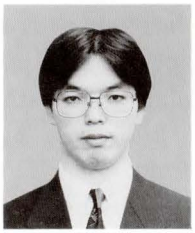
We thank Dr. Tatsushi Tokioka of the Meteorological Research Institute for providing the MRI-GCM code with data and for his advice on using it.

References

1) Ishihata, H. Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Processing, 1991, pp.13-16.
2) Ishihata, H., Horie, T., Inano, S., Shimizu, T., Kato, S., and Ikesaka, M.: Third Generation Message Passing Computer AP1000. Proc. Int. Symp. Supercomputing, 1991, pp.46-55.
3) Potmesil, M., and Foffert, E. M.: FRAMES: Software Tools for Modeling, Rendering, and Animation of 3D Scenes. *ACM Comput. Graph.* **21**, 4, pp. 85-93 (1987).
4) Foley, J. D., and Van Dam, A.: Fundamentals of Interactive Computer Graphics. Addison-Wesley, 1983, pp. 267-318.
5) Lorensen, W. E., and Cline, H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM Comput. Graph.*, **21**, 4, pp. 163-169 (1987).
6) Carpenter, L.: The A-Buffer, an Antialiased Hidden Surface Method. *ACM Comput. Graph.* **18**, 3, pp. 103-108 (1984).
7) Weiner, P. K., and Kollman, P. A.: AMBER: Assisted Model Building with Energy Refinement. A General Program for Modeling Molecules and Their Interactions. *J. Comput. Chem.*, **2**, 3, pp. 287-303 (1981).
8) Sato, H., Yasumasa, T., Iwama, H., Kawakita, K., Saito, M., Morikami, K., Yao, T., and Tsutsumi, S.: Parallelization of AMBER Molecular Dynamics Program For the AP1000 Highly Parallel Computer. Proc. Scalable High Perform. Comput. Conf. (SHPCC-92), 1992, pp. 113-120.
9) Fujita, T., Hirota, K., and Murakami, K.: Computer Graphics for Splashing Water. *Denshi Tokyo,* No. 29 1990, IEEE Tokyo Section, 1991, pp. 70-74.

Hiroyuki Sato

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electrical Eng.
Waseda University 1975
Master of Electrical Eng.
Waseda University 1977
Specializing in Parallel Processing and Computer Graphics



Hideaki Yoshijima

System Dept.
FUJITSU KYUSHU SYSTEM ENGINEERING
Bachelor of Science
Science University of Tokyo 1990
Specializing in Parallel Computer Graphic

Satoshi Inano

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Ibaraki Technical High School 1980
Specializing in Parallel Computer Hardware and Visual Systems

UDC 007.52:681.322

Visualizing 3-Dimensional Data on the AP1000

● Paul Mackerras ● Brian Corrie

(Manuscript received October 12, 1992)

Recent advances in computer technology have made it feasible for scientists and engineers to work with sampled data representing the variation of physical quantities throughout a 3-dimensional volume. The FUJITSU AP1000 has the memory capacity and processing speed to handle large 3-dimensional data sets and generate images from these data sets at interactive or near-interactive rates. We have implemented parallel versions for the AP1000 of two basic techniques for visualizing 3-dimensional scalar data sets: volume rendering and isosurface generation. The key issues in parallelizing these algorithms are the data distribution and work distribution. This paper describes the distribution methods used and the tradeoffs involved, and presents results obtained with example data sets.

1. Introduction

Many scientific and engineering disciplines generate large masses of data describing the variation of quantities as a function of position within a three-dimensional volume. This data can come from physical measurements or computational simulations, and from a wide variety of fields including atmospheric studies, aeronautics, medicine, geophysics, and many others. A volume data set consists of quantized values of one or more scalar fields that have been sampled at positions throughout a 3-dimensional volume. The sampling can be performed on a regular or irregular grid, with data set sizes ranging from kilobytes to gigabytes. These large masses of data are far too complex to be understood by examining the individual sample values. Visualization, the process of translating the data from its numeric form to a visual form, can be of immense assistance in the process of understanding the data.

The size and number of volume data sets that are being produced today, and will be produced in the future, present a challenge to current rendering architectures and techniques. The demand for better and faster algorithms increases faster than the speed and sophistication of the hardware and software. The demand for interactive visualization also drives the need for better and faster algorithms, as researchers need to use visualization techniques for steering their simulations and exploring their results. Batch oriented visualization is no longer a feasible option. One solution to this problem is to use scalable, massively parallel architectures to perform the visualization tasks. By using a scalable architecture, increases in the size and number of the volume data sets can be dealt with by scaling the architecture with the problem size.

Two of the most common techniques for visualizing three-dimensional scientific data are *volume modeling* and *volume rendering*. Volume modeling is the creation of geometric models of areas of interest within a three-dimensional volume. One common technique for this is the Marching Cubes isosurface extraction technique described by Lorensen and Cline[1]. An isosurface is analogous to a contour line on a map – it has the characteristic that the field value is the same at all points on the isosurface. Isosurface display is particularly useful in situations where values of the scalar field can be

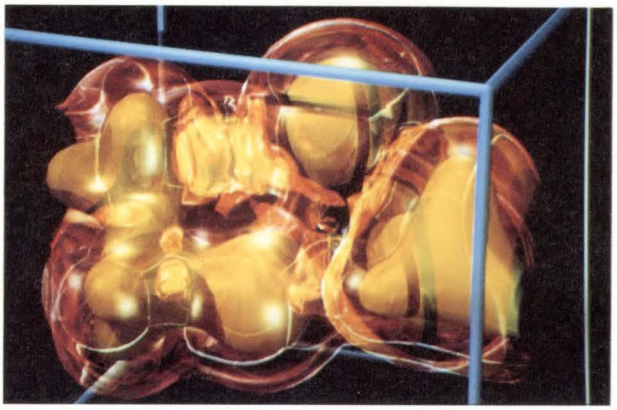


Fig.1 – Two nested isosurfaces of an electron density function.

Fig.2 – Volume-rendered image of atmospheric carbon-dioxide concentration data.

related to discrete classifications of the material being represented, for example, different tissue types in a medical Magnetic Resonance Imaging (MRI) image. Figure 1 is an image of two nested isosurfaces from an electron density function of a simple molecule. The outer isosurface is rendered as semi-transparent so that the inner isosurface can be seen.

Volume rendering is the process of generating an image directly from the volume data without the generation of an intermediate geometric model[2)-7)]. Typically this is done by mapping the sample values in the volume to the colour and opacity of an imaginary semi-transparent material, and then rendering an image of this material. Sample values of interest can be assigned high opacity values and a specific colour to highlight their location within the volume while other sample values can be assigned low opacity values to reduce their visual importance. It is also possible to render geometric and volumetric primitives together, allowing the inclusion of geometric primitives such as coordinate axes and reference objects such as isosurfaces. This technique is useful for displaying relationships between areas of interest that are not well defined in a geometric sense. It is also useful for displaying the volume around areas of geometric interest, such as the volume near an isosurface.

Figure 2 is a volume-rendered image of a data set obtained from a global atmospheric simulation model. The model simulates the emission, transport and absorption of carbon dioxide (CO_2) in the atmosphere, and the image represents the distribution of carbon dioxide in the atmosphere at a single time point in a full year's simulation. Moderate to high concentrations of CO_2 are rendered as yellow to red, while low concentrations of CO_2 are rendered as blue. Intermediate values are rendered as transparent so that the areas of high and low concentration are visible. A geometric reference plane containing a polygonal map of the earth is rendered beneath the volume to give recognizable reference points to the data.

2. Volume rendering using ray casting

Although volume rendering using ray casting is one of the most compute-intensive methods for rendering three-dimensional data, it is still a very widely used technique. The main reasons for this are that it generates high quality images, and that it makes it possible to render both volume data and geometric primitives. Through the use of parallel architectures, it is possible to perform volume rendering using ray casting at interactive rates. The computational power and memory capacity of the AP1000[8),9)] makes it ideal for rendering very large three-dimensional data sets using this technique.

2.1 The ray-casting algorithm

Performing volume rendering with ray casting is conceptually straightforward. A ray is cast from the view point, through the pixels in the image plane, and into the volume data. As the ray passes through the volume it is sampled

at evenly spaced intervals. The value at a sample point is obtained by trilinear interpolation of the values at the corners of the volume element that contains that point. (Trilinear interpolation involves interpolating linearly between sample points along each coordinate axis.) The sample values are then translated to colours and opacities through a mapping function. The contributions of the samples are accumulated until the opacity level becomes sufficiently close to unity or the ray leaves the volume. At this point a final colour for the ray, and therefore the pixel that the ray passed through, can be computed. We have implemented a ray-casting volume renderer as an extension to an existing ray-tracing geometry renderer. Ray tracing is more sophisticated than ray casting, as additional rays are traced to simulate shadows and the light interactions with complex materials. In this fashion we can include geometrically defined structures such as coordinate axes and reference objects that make the volume data easier to understand. More details on the basic structure of the renderer can be found in other work[10),11)].

2.2 Work distribution and load balancing

The ray-tracing program has been parallelized using an *image space* subdivision and the *worker-farm* paradigm, in which a master process (the host or a cell) issues work items (rectangular blocks of pixels) to worker processes (the cells). Initially the master assigns each worker cell a work item to render. As a cell completes its assigned work item it transmits it back to the master, which then sends that cell the next work item to be rendered. Below we present an algorithm that is based on earlier work performed on the AP1000[10)], except that instead of using scan lines as the work items, square pixel blocks are used. Square pixel blocks are preferable to scan lines because of the potential for exploiting *data coherency* while rendering. Data coherency refers to the fact that nearby pixels tend to use much the same data in the rendering process, particularly when rendering volume data. Similarly, various acceleration techniques used in ray tracing make use of cached results to avoid duplicated computation[12)]. These caches are much more likely to contain useful data when rays are traced in contiguous blocks.

Just as adjacent pixels are likely to use much the same data in the rendering process, they are also likely to require a similar amount of computation. Thus there is an unavoidable trade-off between exploiting data coherency and obtaining a good load balance. There is also a trade-off between load balance and communication. Small work items require more communication (since two messages are required for each work item) but give better load balance. Large square work items require less communication and exploit data coherency effectively, but can result in extremely poor load balance. To obtain a better load balance using square pixel blocks, we have developed an adaptive extension to the worker-farm algorithm, using timeouts for each pixel block. If a cell is rendering a pixel block and the timeout period expires, the cell sends the master a message describing how much of the block has been completed. If there are no more pixel blocks to be rendered and there are idle cells, the master subdivides the pixel block among the idle cells (including the cell that just completed the block). If there are no idle cells, the master simply sends the block back to the cell for completion.

2.3 Data distribution and distributed virtual memory

With the work distribution system described above it is not possible to determine *a priori* which pixel blocks a cell will be required to render. It is therefore necessary for each cell to be able to render any portion of the image, which in turn makes it necessary for each cell to have access to any part of the volume data at any time. Since volume data sets can be very large and the memory on each cell is limited, we have implemented a distributed virtual memory system to provide that access. During the rendering process, when a cell requires volume data which it does not have, it requests it from another cell. This increases the amount of communication required for the rendering proc-

ess and results in a slower overall rendering time. The implementation goal of the distributed virtual memory system is to minimize this communication.

The implementation of distributed virtual memory described below is loosely based on the work performed by Green and Paddon[13),14)]. The implementation for our renderer divides the AP1000 into a set of neighbourhoods of one or more adjacent processing cells. The volume data set is subdivided into approximately square subblocks of volume data of less than 256 Kbytes. Each neighbourhood contains the entire volume data set, with the data set distributed over the cells in that neighbourhood. The algorithm used to divide an AP1000 with N cells into neighbourhoods is described below:

Given a parallel machine with N cells, divide it up into N neighbourhoods of 1 cell each. Determine whether the entire data set will fit in one of these neighbourhoods. If it does, distribute the data set across all of the cells in each neighbourhood. If the data set does not fit, double the size of each neighbourhood (halving the number of neighbourhoods). Repeat this process until the entire data set fits in the available memory of a neighbourhood.

When rendering is being performed, if a given cell needs a portion of the data set that it does not have locally, it can retrieve the required data from one of the cells in its neighbourhood. Neighbourhoods minimize the communication required by keeping as much of the data set on each cell as possible (ideally all data is kept on each cell so no communication is needed) and by localizing the communication required (to reduce network contention).

In addition to the volume subblocks that each cell serves, each cell contains a Least Recently Used (LRU) cache of volume subblocks that it needs to perform the rendering it has been assigned. This LRU scheme exploits the data coherency of the volume data very effectively. When a cell is rendering a square pixel block, most of the rays cast through adjacent pixels will require approximately the same volume blocks to render those rays, resulting in fewer cache misses and therefore less communication overhead. See Corrie and Mackerras[11)] for more details on this caching algorithm.

2.4 Results

The two key issues in developing a parallel volume rendering system are the work and data distributions. These two issues involve very important tradeoffs in general, and are especially important when considering an image space work distribution of a ray-casting renderer combined with a distributed virtual memory. It has been shown that *dot mode* work distribution is useful for obtaining a good load balance[15)]. This technique does not work well with our data distribution technique, as it does not take advantage of the data coherence that is available in rendering a volume data set.

In Table I we see the statistics obtained while rendering a geometric object database of approximately 10 000 polygons (image resolution of 512 × 512 pixels, rendered on a 128-cell AP1000). Note that with a work item *timeout* of one second a very low *load imbalance* results for all pixel block sizes. In both Table 1 and Table 2, the load imbalance is the average time cells spend idle waiting for the last cell to finish. In general, the load imbalance will be less than the block timeout value, except in extreme cases. The slightly higher *item delay* overheads (the average time spent waiting for work items) for 2 × 2, 30 × 30, and 40 × 40 pixel blocks (1 second timeout) are a result of the large number of work items that are generated for these cases. The 2 × 2 pixel block case generates a large number of work items because of the small pixel block size. The 30 × 30 and 40 × 40 pixel block cases generate a large number of work items because of the large amount of pixel block subdivision that must occur at the end of the rendering to achieve a good load balance. The *rendering time* is the average time it takes to perform the actual rendering, and should be approximately the same for all situations. The important factor to note is that even with the item delay overheads, the *total times* (the time from when the rendering starts until the last cell finishes) are very close for all block sizes that use a one second timeout. Note that when the

Table 1. Rendering statistics for work item timeouts

Work item size	Timeout (seconds)	Item delay (seconds)	Load imbalance (seconds)	Rendering (seconds)	Total (seconds)
2 × 2	1	8.3	0.0	13.2	25.2
5 × 5	1	0.5	0.0	12.6	13.1
10×10	1	0.1	0.2	12.5	13.1
20×20	1	0.6	0.1	12.4	13.2
30×30	1	2.0	0.0	12.4	14.5
40×40	1	4.8	0.2	12.4	17.8
10×10	∞	0.1	0.2	12.5	12.8
20×20	∞	0.0	2.9	12.4	15.4
30×30	∞	0.0	9.4	12.4	22.1
40×40	∞	0.0	29.7	12.4	42.1
1 ×100	∞	0.1	0.0	12.7	12.9
2 ×200	∞	0.0	0.3	12.5	12.9
3 ×300	∞	0.0	1.5	12.5	14.0
4 ×400	∞	0.0	9.8	12.5	22.4

Table 2. MRI rendering statistics

Work item	Render (seconds)	Cache (seconds)	Item delay (seconds)	Load imbalance (seconds)	Total (seconds)	Cache misses
Dot	144.6	116.2	1.8	6.4	270.4	559
2 × 2	139.6	164.8	22.9	0.2	327.6	835
5 × 5	144.0	41.0	2.6	0.4	188.2	187
10×10	144.4	18.1	1.4	0.8	164.8	84
20×20	144.4	12.1	1.2	1.7	160.8	57
40×40	144.0	18.1	1.3	1.5	164.9	86
1 × 4	137.0	258.7	31.6	0.2	430.6	1 298
1 ×25	143.5	79.5	5.0	0.2	229.6	380
1 ×100	144.6	41.7	2.1	0.4	188.8	197
1 ×400	144.6	31.1	1.6	0.7	178.1	147
4 ×400	144.5	19.4	1.6	2.0	167.5	91

adaptive work item subdivision is disabled (timeout = ∞), high load imbalance results are associated with large pixel blocks, resulting in long total rendering times.

Determining an effective compromise between work and data distribution techniques is essential to the implementation of a practical parallel volume rendering system. Minimizing the amount of data communication that is required for data distribution is done by exploiting the data coherency of volume data. The top half of Table 2 shows the rendering times obtained using both the dot mode and square adaptive pixel block algorithms on a 256 × 256 × 109 volume data set. It is a standard data set from the University of North Carolina at Chapel Hill's volume rendering test data set, and is a Magnetic Resonance Imagery (MRI) study of a human skull. The volume was stored as 32 bit floating point data, resulting in over 27 Mbytes of volume data. Note that dot mode and 2 × 2 pixel block work items have a large number of *cache misses* and a large amount of *cache time* dedicated to serving those misses. The number of cache misses is the average number of times (per cell) that a cell has to request volume data from a cell in its neighbourhood. The cache time is the average time a cell spends either

waiting for cache miss to be serviced or serving a cache miss from another cell. The load balance of all block sizes (including dot mode) is quite good, and the factors that determine the overall rendering performance are the time spent servicing cache misses and the item delay overhead involved in dealing with a large number of small work items (i.e. the 2 × 2 work items have a large item delay overhead).

The importance of data coherency is demonstrated in the timings from Table 2, as we can see that small pixel blocks that are not adjacent to one another (dot mode, 2 × 2, 1 × 4, 5 × 5, 1 × 25 pixel blocks) result in relatively large cache time overheads. In the bottom half of Table 2, we render the data set with an elongated pixel block work distribution with pixel block sizes equal to the square pixel block sizes from the top half of Table 2. Note that the large thin pixel blocks have relatively high cache miss results, as a long line of pixels covers a large number of volume subblocks. The 4 × 400 pixel block gives the best results since it is more than one pixel in width and therefore takes advantage of the data coherency to a higher degree than the other elongated pixel blocks. Even when large pixel blocks such as these are used for rendering, the adaptive work distribution algorithm provides a reasonably small load imbalance. The marginally longer load imbalance time of the 4 × 400 block (1.3 seconds greater than 1 × 400 pixel blocks) is outweighed by the savings in decreased cache time (11.7 seconds less than 1 × 400 pixel blocks).

3. Isosurface generation

Isosurfaces are usually represented as geometric objects (typically a mesh of small triangles) which can be rendered using standard techniques[16),17)]. Such an isosurface representation does not depend on the direction of view – having generated the isosurface representation, the user can render an image of it from any desired point of view. The 'Marching Cubes' algorithm[1)] is an efficient algorithm for generating a polygonal description of an isosurface of 3-dimensional data sampled on a rectangular grid. The polygons are in general non-planar, but can be converted into a triangle mesh for rendering.

We have implemented a parallel version of the Marching Cubes algorithm for the AP1000, together with a simple parallel Z-buffered renderer for viewing the results. This section describes our parallel Marching Cubes implementation and briefly outlines our parallel Z-buffered renderer implementation. For further details, the reader is referred to Mackerras[18)].

3.1 Outline of basic algorithm

Our parallel isosurface program for the AP1000 is based on an efficient serial implementation of a modified version of the Marching Cubes algorithm. The sampled volume data is divided into one or more contiguous blocks. Each cell processor of the AP1000 is allocated one or more blocks of data, and runs the serial Marching Cubes code (essentially unchanged) on each block, producing a list of triangles. The complete surface is obtained simply by concatenating these lists of triangles.

The Marching Cubes algorithm considers each unit cube defined by a 2 × 2 × 2 grid of sample points. If some of the samples at the corners of the cube are above the isosurface value and some are below, then the isosurface passes through the cube. For all such cubes, the algorithm uses linear interpolation to find the intersections of the surface with the edges of the cube, and then connects the intersections up into polygons. These polygons are then converted into a triangle mesh. In addition, the gradient[Note)] of the 3D scalar data field is optionally calculated (using a simple central-difference formula) at each triangle vertex for use as the surface normal in the renderer's shading calculations.

Except at the boundaries of the volume, each sample point is shared between eight cubes,each edge between four cubes, and each face between two cubes. The Marching Cubes algorithm makes use of these facts to minimize

Note: The gradient of a scalar field is a vector in the direction in which the field increases most rapidly.

the amount of computation required; data computed for the corners, edges and faces of one cube are used in the adjacent cubes without recomputation.

The original Marching Cubes algorithm can generate surfaces with incorrect topology (surfaces with holes or other flaws) in some circumstances. Various modifications to the algorithm have been suggested to overcome this deficiency[19),20)]. Our implementation ensures that the generated isosurface is self-consistent and topologically similar to the isosurface of the field obtained by trilinear interpolation between the sample points.

3.2 Data and work distribution

The unit of work in the Marching Cubes algorithm is the unit cube defined by a grid of $2 \times 2 \times 2$ sample points; the algorithm makes decisions about the topology of the isosurface within the cube based on the values of all 8 sample points. Furthermore, the choice of topology within one unit cube does not influence the topology within adjacent cubes, except insofar as sample values are shared.

Thus a parallel implementation can assign any set of unit cubes to each processor. Each processor will need the sample points defining the unit cubes which it has been assigned (plus some neighbouring samples if gradient values are to be calculated), and will calculate the portion of the isosurface which lies within those unit cubes. The complete isosurface is obtained by aggregating the polygons from each of the unit cubes, just as it is in the serial Marching Cubes algorithm.

The optimizations which are possible in the Marching Cubes algorithm represent a kind of data coherency which makes it more efficient to allocate a connected volume of unit cubes to a processor than to allocate the same number of unit cubes in a scattered fashion. When a given processor is allocated adjacent cubes, it can optimize by storing each sample point only once, instead of once for each unit cube which requires that sample point. Similarly, values which are computed for the faces, edges and sample points of one cube can easily be used for adjacent cubes without recomputation if the adjacent cubes are allocated to the same processor. (If the adjacent cubes are allocated to a different processor, it is possible to avoid recomputation by transmitting the results from one procerssor to another; however, the cost of the communication required would be likely to be greater than the cost of re-computing the values. Also, the algorithm would be complicated by the need for each processor to process its cubes in an order which both avoids lengthy waits for data from other processors, and makes data available to other processors in a timely fashion.)

These considerations would lead to a data and work subdivision in which each processor is assigned a single large block of the volume. However, such a subdivision can have poor load-balance properties. Typical volume data sets often have a region of interest near the middle of the volume, with the result that cubes near the center of the volume are more likely to be intersected by the isosurface than cubes near the boundaries of the volume. Since the processing for a cube which is intersected by the isosurface is more expensive than for one which is not, this can lead to considerable load imbalance.

To overcome this problem, we divide the volume into more blocks than there are cell processors, and assign a small number of blocks to each processor, allocated such that each processor has some blocks near the middle of the volume and some near the boundaries. This is done in a dot-mode fashion: our implementation divides the X-Y plane into $n_X C_X \times n_Y C_Y$ blocks for an AP1000 processor array of dimension $C_X \times C_Y$, where n_X and n_Y are the 'degrees of interleaving' in the X and Y dimensions respectively (the Z dimension is not divided). These are assigned in an interleaved fashion to the processors; cell processor (i, j) receives blocks $(i + lC_X, j + mC_Y)$, for $0 \leqq l < n_X$, $0 \leqq m < n_Y$.

In our AP1000 implementation, the volume data are broadcast to the cell processors by the host as a series of X-Y planes of data. Each cell receives each complete plane and copies those samples which are required for processing the cubes which it has been assigned. This was

found to be considerably more flexible than the AP1000 scatter operation, and no slower. The triangles generated can be sent to the host processor for storage in a disk file, or alternatively rendered by the cell processors using the Z-buffer renderer described in the next section.

3.3 Z-buffer renderer

We have found in practice that while the AP1000 can generate millions of triangles in a few seconds, it takes much longer (up to 100 times longer) to transmit these triangles to the host processor for storage, display or other processing. Clearly it is necessary to implement as much as possible of the further processing of the triangles on the cell processors. To this end, we have implemented a simple distributed Z-buffered triangle renderer on the cell processors. The rendered image still has to be transmitted to the host processor for display, but this is often a much smaller amount of data than the triangles.

Our renderer uses an adaptive hybrid image-space / object-space subdivison technique to parallelize the rendering process. Image-space subdivision requires each processor to have the full list of triangles, whereas object-space subdivision requires each processor to have a full-sized Z buffer. Our subdivision technique divides the processors into groups, each of which has (collectively) a full-sized Z-buffer. Object-space subdivision is used between groups, and image-space subdivision within the group. When each group has finished rendering its triangles into its Z buffer, the Z buffers are merged between groups in a bitonic Z-buffer merge phase. The image is then gathered to the host processor for display. Further details are given in Mackerras[18].

3.4 Results

The performance of our parallel Marching Cubes implementation has been measured on several example volume data sets using the following measures:

1) Total time taken to generate the isosurface (excluding time to load the data). This is the time taken by the slowest cell processor.
2) Average time taken by the cell processors to generate their portion of the isosurface.
3) Load balance: the ratio of 2) to 1).
4) Speedup: the total time taken by a single cell processor divided by 1).

Figure 3 shows how the total time and load balance vary with the degree of interleaving for a typical volume data set. The times shown are for generating an isosurface of 46 266 triangles from a volume of $256 \times 256 \times 40$ samples on 8×16 processor AP1000. The height of each column shows the total time taken for the cell processors to perform the isosurface operation {item 1) above}, including gradient calculations, and synchronize. The darker portion of each column shows the average time per processor taken to compute the isosurface (excluding synchronization time). The lighter portion shows the load imbalance, measured as the average time spent waiting for synchronization. The minimum time taken to synchronize was 0.55 ms in each case. The results were quite consistent over several runs.

The results show an interesting effect: a modest degree of interleaving reduces the total time for the isosurface generation although it increases the average time per cell, which is a measure of the total amount of computation performed. That is, assigning a small number of blocks (around 8) to each cell improves the load balance to a sufficient extent to outweigh the extra computation required by the division into

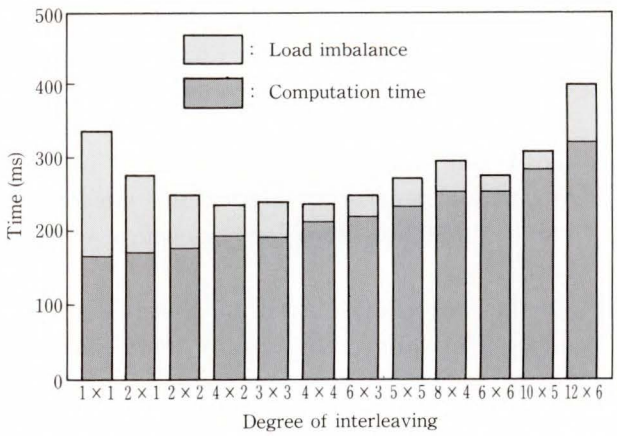


Fig.3 – Measured execution times as a function of interlerving.

smaller blocks.

The time taken for a single cell processor to perform the isosurface operation was 30.1 seconds, giving a speedup of 133 for the fastest case ($n_X \times n_Y = 4 \times 2$). This is greater than the number of processors due, we believe, to cache effects: the complete machine has a total of 16 Mbytes of cache memory, enabling it to keep more of the volume data in cache than a single cell processor can.

4. Discussion

The parallel implementations of volume rendering and isosurface generation described here represent contrasting approaches to parallelization: one uses dynamic work distribution with a distributed virtual memory scheme to make the necessary data available when required; the other uses a static work distribution with a static data distribution. Yet there are similarities between the requirements of the algorithms: they both have quite a small unit of work (one pixel, one unit cube), and they both have significant optimizations which are available when adjacent work units are assigned to the same processor. Both operate on volume data which is potentially large enough that it cannot be stored on a single processor. Why then have we employed such different parallelization schemes ?

The answer lies in some quite significant differences in the requirements of the two algorithms.

1) A static load balance is more effective for the isosurface generation algorithm than for the volume rendering algorithm, because the difficulty of individual work units varies less. The ratio of longest to shortest times for an individual work unit is less than 9 for the isosurface generation algorithm, whereas it can be 10 000 or more for ray-casting volume rendering.
2) In the isosurface algorithm, there is a very straightforward relationship between the work units which a processor has been assigned and the set of data which it therefore requires. Furthermore, a relatively small amount of that data will overlap with other cell processors. The static work distribution we have used enables us to use a static data distribution which is simple, efficient – no time is consumed transmitting data from one cell to another, and scalable – volume data approaching the total memory size of the machine can be handled. In contrast, the relationship between a pixel's position and the set of volume data which are required to render that pixel is a complex, viewpoint-dependent function. This makes a distributed virtual memory scheme attractive as a way to make the necessary data available when required without requiring a complex and time-consuming *a priori* analysis of data requirements, which would probably take a substantial fraction of the rendering time. Thus a static work distribution offers no advantage in terms of a simple data distribution. The dynamic work distribution scheme described above is relatively simple to implement and provides much better load balance than a static work distribution.

5. Conclusion

In this paper we have described two approaches to visualizing volume data: ray-casting volume rendering and isosurface generation and display. Implemented on the AP1000, these techniques enable the user to interactively explore large scale volume data sets. The concept of data coherency was used to motivate the implementation of an adaptive worker-farm work distribution and a distributed virtual memory. The results presented above show that this is an efficient method of utilizing the inherent data coherency between neighbouring pixels that exists in rendering three-dimensional volume data sets. Using this technique, it should be possible to directly render large volume data sets that could not be rendered on a normal workstation. A simple technique was used to parallelize the isosurface generation algorithm; a dot-mode distribution of relatively large blocks of volume data gives reasonable load balance while still exploiting the data coherence available.

Our implementations demonstrate that the AP1000 architecture is well-suited to handling the problems of visualizing large 3-dimensional data sets. In particular, they demonstrate that both volume rendering (using ray casting) and isosurface generation and display can be parallelized effectively on distributed-memory MIMD machines such as the AP1000. Furthermore, they provide evidence that these machines will scale well on this class of problems. Both the processing speed and the data-set size which can be handled increase approximately linearly with the number of processors; in some cases we have actually observed super-linear speedup, due we believe to the increase in total cache memory size with the number of processors.

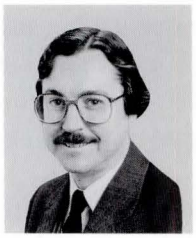
6. Acknowledgement

We would like to thank our colleagues at Fujitsu Laboratories for many interesting discussions. Mr. Drew Whitehouse of the ANU Supercomputer Facility constructed an initial serial version of the volume rendering algorithm. We would like to thank Dr. John Taylor of the Center for Resource and Environmental Studies, ANU, for the use of the atmospheric carbon-dioxide simulation data.

References

1) Lorensen, W. E., and Cline, H. E.: Marching cubes: a high resolution 3D surface construction algorithm. *ACM Computer Graphics*, **21**, 4, pp. 163-169 (1987).
2) Drebin, R. A., Carpenter, L., and Hanrahan, P.: Volume rendering. *ACM Computer Graphics*, **22**, 4, pp. 65-74 (1988).
3) Kajiya, J. T., and Von Herzen, B. P.: Ray tracing volume densities. *ACM Computer Graphics*, **18**, 3, pp. 165-173 (1984).
4) Laur, D., and Hanrahan, P.: Hierarchical splatting: a progressive refinement algorithm for volume rendering. *ACM Computer Graphics*, **25**, 4, pp. 285-288 (1991).
5) Sabella, P.: A rendering algorithm for visualizing 3D scalar fields. *ACM Computer Graphics*, **22**, 4, pp. 51-58 (1988).
6) Upson, C., and Keeler, M.: V-BUFFER: Visible volume rendering. *ACM Computer Graphics*, **22**, 4, pp. 59-64 (1988).
7) Wilhelms, J., and Van Gelder, A.: A Coherent projection approach for direct volume rendering. *ACM Computer Graphics*, **25**, 4, pp. 275-284 (1991).
8) Ishihata, H., Horie, T., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Processing, 1991, pp. 13-16.
9) Ishii, M., Sato, H., Murakami, K., Ikesawa, M., and Ishihata, H.: Cellular Array Processor and its Applications. *J. VLSI Signal Processing*, **1**, pp. 57-67 (1989).
10) Mackerras, P.: Scientific Visualization Algorithms on the AP1000. Proc. Second ANU-Fujitsu CAP Workshop, 1991, pp. Q-1-Q-9.
11) Corrie, B., and Mackerras, P.: Parallel Volume Rendering and Data Coherence on the AP1000. Tech. Rep. TR-CS-92-11, Dept. Computer Science, The Australian National University, 1992.
12) Arvo, J., and Kirk, D.: "A Survey of Ray Tracing Acceleration Techniques". An Introduction to Ray Tracing. A. S. Glassner, ed., Academic Press, London, 1989.
13) Green, S. A., and Paddon, D. J.: A Highly Flexible Multiprocessor Solution for Ray Tracing. Tech. Rep. TR-89-02, Computer Science Dept., University of Bristol, 1989.
14) Green, S. A., and Paddon, D. J.: Exploiting Coherence for Multiprocessor Ray Tracing. *IEEE Computer Graphics and Applications*, **9**, 6, pp. 12-26 (1989).
15) Ishii, M., Goto, G., and Hatano, Y.: Cellular Array Processor CAP and its Application to Computer Graphics. *FUJITSU Sci. Tech. J.*, **23**, pp. 397-390 (1987).
16) Foley, J. A., van Dam, A., Feiner, S. K., and Hughes, J. F.: Computer Graphics: Principles and Practice. Second Edition, Addison-Wesley, Reading, Massachusetts, 1990.
17) Carpenter, L.: The A-buffer, an antialiased hidden surface method. *ACM Computer Graphics*, **18**, 3, pp. 103-108 (1984).
18) Mackerras, P.: A Fast Parallel Marching Cubes Implementation on the Fujitsu AP1000

Tech. Rep. TR-CS-92-10, Dept. Computer Science, The Australian National University, 1992.
19) Wilhelms, J., and Van Gelder, A.: Topological Considerations in Isosurface Generation. (Extended Abstract), ACM Computer Graphics, **24**, 5, pp. 79-86 (1990).
20) Wyvil, G., McPheeters, C., and Wyvil, B.: Data structure for soft objects. *The Visual Computer*, **2**, 4, pp. 227-234 (1986).



Paul Mackerras

Department of Computer Science
Australian National University
Bachelor of Science
University of Queensland 1982
Bachelor of Engineering(Electrical)
University of Queensland 1982
Ph.D. in Computer Science
Australian National University 1988
Specializing in Scientific Visualization

Brian Corrie

Department of Computer Science
Australian National University
Bachelor of Science
University of Victoria 1988
Master of Science
University of Victoria 1990
Specializing in Scientific Visualization

UDC 681.322.001.4

Implementation of the BLAS Level 3 and LINPACK Benchmark on the AP1000

● Richard P. Brent ● Peter E. Strazdins

(Manuscript received October 12, 1992)

This paper describes an implementation of Level 3 of the Basic Linear Algebra Subprogram (BLAS-3) library and the LINPACK benchmark on the Fujitsu AP1000. The performance of these applications is regarded as important for distributed memory architectures such as the AP1000. We discuss the techniques involved in optimizing these applications without significantly sacrificing numerical stability. Many of these techniques may also be applied to other numerical applications. They include the use of software pipelining and loop unrolling to optimize scalar processor computation, the utilization of fast communication primitives on the AP1000 (particularly row and column broadcasting using wormhole routing), blocking and partitioning methods, and 'fast' algorithms (using reduced floating point operations). These techniques enable a performance of 85-90 % of the AP1000's theoretical peak speed for the BLAS Level 3 procedures and up to 80 % for the LINPACK benchmark.

1. Introduction

The Basic Linear Algebra Subprogram (BLAS) library is widely used in many supercomputing applications, and is used to implement more extensive linear algebra subroutine libraries, such as LINPACK and LAPACK. To take advantage of the high degree of parallelism of architectures such as the Fujitsu AP1000, BLAS Level 3 routines (matrix-matrix operations) should be used where possible.

The LINPACK benchmark involves the solution of a nonsingular system of n linear equations in n unknowns, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. The performance of the LINPACK benchmark and the BLAS-3 are both regarded as good indicators of a parallel computer's potential in numeric applications.

The AP1000[1),2)] is a distributed memory MIMD machine with up to 1 024 independent SPARC processors which are called cells connected via a toroidal topology using wormhole routing. Each processor has a 128 Kbyte direct-mapped copy-back cache, 16 Mbytes of memory and a FPU of theoretical peak of 8.3 MFLOPs (single precision) and 5.6 MFLOPs (double precision). Details of the AP1000 architecture and software environment are discussed in this issue[3),4)].

High level design issues, most importantly the distribution of matrices over the AP1000, are discussed in Chap. 2. Techniques for the optimization of matrix computations on single AP1000 cells are given in Chap. 3. Chapter 4 describes the implementation of parallel matrix multiply-add operations on the AP1000, discussing issues such as communication, cache, non-square matrix shapes, and so-called 'fast' multiplication methods. The implementation of the LINPACK benchmark is discussed in Chap. 5, emphasizing the need for 'blocking' together small computations into larger ones. The application of this and techniques from Chap. 4 to the similar problem of BLAS-3 triangular matrix update is given in Sec. 5.3. Conclusion is given in Chap. 6.

1.1 The BLAS Level 3

The BLAS Level 3[5)] implement matrix-matrix operations, which, for $n \times n$ matrices, involve $O(n^3)$ arithmetic operations on $O(n^2)$

data items. This yields a higher ratio of arithmetic operations to data than for the BLAS Level 2 (BLAS-2)[6], although degenerate cases of the BLAS-3 routines yield all BLAS-2 routines. The use of BLAS-3 is attractive on parallel machines such as the AP1000 because the cost of a data transfer may be amortized over the cost of $O(n)$ arithmetic operations.

The BLAS-3 perform multiply-add operations of the form:

$$C \leftarrow \alpha \tilde{A}\tilde{B} + \beta C$$

where $\tilde{A}$ can be either A or A^T (and similarly for $\tilde{B}$); multiply-add operations for symmetric matrices, eg.:

$$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$$

where C is symmetric; and triangular matrix update operations of the form:

$$B \leftarrow \alpha \tilde{A}B,\ B \leftarrow \alpha B\tilde{A}$$

where A is triangular and $\tilde{A}$ can be A, A^T, A^{-1} or A^{-T}. Matrices may be general rectangular, symmetric or triangular but there is no special form of "packed" storage for symmetric or triangular matrices.

1.2 The LINPACK benchmark

The LINPACK benchmark involves the solution of a nonsingular system of n linear equations in n unknowns, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. Three cases are considered:

1) $n = 100$: the original benchmark.
2) $n = 1\,000$: gives more opportunity for vector pipeline machines (and to some extent parallel machines) to exhibit high performance.
3) n as large as desired: gives maximum opportunity for vector pipeline and parallel machines to exhibit high performance.

We are only concerned with the cases 2) and 3) here, since case 1) is trivial to solve on a machine as powerful as the AP1000.

1.3 Conventions and restrictions

We use the C language for implementation, as it permits better access to the low-level details of the AP1000, which is useful for optimizations. Thus, we use C conventions for matrices, stored in row-major ordering with indices starting from 0. Associated with the row-major storage scheme for an $m \times n$ (cell sub-) matrix A is the *last dimension* of A, denoted ld_A, where $n <= \mathrm{ld}_A$. This enables A to be identified as sub-matrix of a larger $m' \times \mathrm{ld}_A$ matrix A', where $m <= m'$. Let $A_{i,}$ denote the ith row and $A_{,j}$ denote the jth column of the matrix A.

Let $N_x(N_y)$ be the number of cells across a row (column) of an AP1000 configuration, and $P = N_y N_x$ be the total number of processors. Our algorithms will be first described for a square ($N_x \times N_x$) AP1000, and then generalizations to other AP1000 configurations will be given. A minor restriction is that for an $m \times n$ matrix to be distributed over the AP1000, we must have $N_y \mid m$ and $N_x \mid n$ (if necessary, matrices can be padded with 'dummy' rows and columns to satisfy this restriction).

2. High-level design issues

On non-distributed memory machines, calls to the BLAS-3 and LINPACK routines reference global matrices; to achieve the same effect on a distributed memory machine, we must have all AP1000 cells calling, in SPMD mode, the corresponding routine with references to the cell's respective sub-matrix of the global matrix. This unfortunately means that uniprocessor codes involving these routines cannot be directly ported to AP1000 cell programs.

To consider the optimal matrix distribution strategy, let us first consider what communication patterns are needed for these applications. These include, most importantly, (grouped) row/column broadcasts, row/column send/receive (for pivot row interchange for LINPACK and matrix rotation for 'systolic' matrix multiply) and finally row/column scan (e.g. vector maximum for LINPACK).

For reasons of symmetry, high bandwidth for the row and column broadcasts, and good load balancing (especially for operation on triangular matrices and contiguous sub-blocks of larger global matrices), matrices are distributed over AP1000 cells by the *cut-and-pile* or *scattered* strategy, rather than storage by rows, by col-

umns, or by contiguous blocks. In the scattered strategy, matrix element $a_{i,j}$ is stored in cell (i mod N_y, j mod N_x), assuming that there are $N_y \times N_x$ cells in the AP1000 array.

A generalization of all these distribution strategies is the 'blocked panel-wrapped' strategy, which is sufficient for all dense linear algebra applications in practice[7]. We have not implemented our algorithms for this more general strategy, as it introduces considerable coding difficulties. Also, due to the relatively low communication startup overheads on the AP1000, it would not yield significantly better performance.

3. Optimizing computation on SPARC processors

To optimize floating point computation on AP1000 cells, we have implemented kernels which are essentially a subset of uniprocessor BLAS-2 and BLAS-3 routines, optimized for the SPARC architecture used in AP1000 cells and written in SPARC assembly language. For this purpose, the following techniques were used:

1) write SPARC "leaf" routines to minimize procedure call overheads[8].
2) keep all variables and array elements in registers, to re-use as much as possible; this enables a low load/store to floating point operation ratio (denoted R).
3) use *software pipelining*, i.e. separate loads, multiplies, adds, and stores which depend on each other by a sufficiently large number of instructions so that their operands are always available when needed.

Techniques 2) and 3) were achieved by using typically a 4×4 (for single precision) and a 4×2 (for double precision) loop unrolling.

The most important of such kernels was the Level 3 UpdateRect ( ) routine which, for single precision, involves a matrix multiply-add $C \leftarrow C + AB$ where A is $4 \times k$ and B is $k \times 4$. This routine would initially load C into the FPU registers, and, upon the ith iteration, update it using $A_{,i}$ $B_{i,}$, $0 \leq i < k$.

UpdateRect ( ) has $R = 0.375$ (double precision) and $R = 0.25$ (single precision); the latter can be effectively reduced further to $R = 1/6$ using the SPARC load double word instruction. When used to perform an $n \times n$ matrix multiplication (with a 'warm' cache), UpdateRect ( ) can sustain 7.7 MFLOPs ($80 \leq n \leq 160$) for single precision, and 5.1 MFLOPs ($56 \leq n \leq 120$) for double precision.

The next most important kernel is the Level 2 Rank1Update ( ), which implements the multiply-add $C \leftarrow C + ab$ where a is $m \times 1$ and b is $1 \times n$. A naive implementation would have R = 1.5; however, for single precision, using a 4×4 loop unrolling, this can be reduced to 1.125, again effectively reducible to slightly less than unity using the load double word instruction. Rank1Update ( ) can sustain 5.9 MFLOPs ($64 \leq n \leq 128$) for single precision, and 4.0 MFLOPs ($48 \leq n \leq 100$) for double precision.

The other Level 2 routines, vector-matrix multiply and matrix-vector multiply, can sustain 7.3 MFLOPs (single precision) and 5.0 MFLOPs (double precision) for matrix multiplication.

These routines can achieve about the same percentage of the theoretical peak on the SPARC Station 1+ and SPARC 2 processors. An exception is Rank1Update ( ), which operates about 25 % slower on these architectures, due to their 'write-through' caches.

The implications of these results for the following sections are as follows:

1) use UpdateRect ( ) wherever possible, even if it requires re-organization of the algorithm.
2) using UpdateRect ( ) to perform $C \leftarrow C + AB$ means that only A and B are significant with respect to the cache. This make good cache utilization much easier. For parallel algorithms, it is better to choose an algorithm not involving the communication of C, as message receipt of C may then displace either A or B from the cache.

4. Implementing BLAS-3 parallel matrix multiply-adds

In this chapter, parallel matrix multiply-add operations, e.g. $C \leftarrow C + AB$ where A, B, C are matrices distributed over the AP1000 cells, are considered, firstly for an $N_x \times N_x$ AP1000, and then for a general AP1000 configuration (Sec. 4.4). The simplest parallel matrix multiplication

Table 1. Speed in MFLOPs / cell of parallel multiply-add methods on an 8 × 8 AP1000 with $n \times n$ matrices (single precision)

n/N_x	$C \leftarrow C+AB$ by (-systolic) method:			$C \leftarrow C+A^TB$ by (-systolic) method:	
	Full-	Semi-	Non-	Semi- (implicit)	Non- (explicit)
16	4.2	4.4	4.4	4.3	4.1
32	5.8	6.0	6.0	5.9	5.8
64	6.5	6.7	6.8	6.7	6.7
96	7.0	6.9	7.0	6.9	6.9
128	7.1	7.2	7.2	7.1	7.1
160	7.1	7.2	7.1	7.1	7.1

algorithm, which we call the 'non-systolic' method, is as follows:

for ($k = 0$, $k<N_x$, $k++$)
  y-broadcast B cell sub-block from row k;
  x-broadcast A cell sub-block from column k;
  perform local cell sub-block multiplication;

A variation is the 'semi-systolic' method[9] where B cell sub-blocks are broadcast from the kth diagonal (instead of from the kth row), and each A cell sub-block is shifted right one unit (instead of broadcast). A third variation is the 'full-systolic' method (also known as Cannon's algorithm) in which both A and B sub-blocks are rotated at each step; this however has the overhead that both A and B must be initially 'aligned'.

Table 1 indicates the relative efficiency of each method for single precision. The overhead of the initial matrix alignment of the 'full-systolic' method makes it the slowest.

To compute $C \leftarrow C + A^TB$ without using explicit transposition, variations of the 'semi-systolic ' and the 'systolic' methods can be used where C's cell sub-blocks are communicated in place of those of B (similarly for $C \leftarrow C + AB^T$). This has a small overhead in extra disturbance of the cache, as explained in Chap. 3.

For explicit matrix transposition, the simplest method of exchanging matrix sub-blocks between cells appears to be the most efficient. The bottleneck for this algorithm is at the diagonal cells, through each of which N_x-1 messages must pass and change direction, so that the time is expected to be proportional to N_x-1 (for constant n/N_x). Transposition has a communication rate of 1.4 Mbytes/s per cell for an 8 × 8 AP1000 ($64 \leq n/N_x \leq 256$), which implies a small relative overhead (for $n/N_x \simeq 128$, the overhead is about 1 %).

Table 1 indicates that for square matrices, these is little difference between the explicit and implicit methods, except for small matrices, which favour the implicit method. This is due to the high relative speed of the AP1000 communication routines, which make the choice of communication patterns less critical.

4.1 Effect of communication

Comparison of Table 1 with the results of Chap. 3 shows that the effect of communication on performance is appreciable, at least for moderate matrix sizes.

In the AP1000's xy communication routines, copying of matrices is avoided on message send; however, upon message receipt, messages are copied from a 'ring buffer' to user space. Message copying creates a twofold overhead, since message transfer (in hardware) on the AP1000 is almost as fast as a corresponding memory transfer, and message copying may disturb the cache. We made slight modifications to the xy routines so that the A and B cell sub-blocks were accessed directly from the ring buffer, thus avoiding the copy.

The performance of this optimization was tested for the non-systolic multiply-add method, and generally halved the communication overhead. Thus, for $n/N_x = 128$, a performance of 7.5 MFLOPs/cell (single precision) was achieved, 90 % of the theoretical peak.

4.2 Optimizing cache utilization and partitioning

BLAS-3 routines generally operate on sub-blocks of larger matrices, rather than whole matrices as such. Using the scattered distribution strategy, these sub-blocks are generally not contiguous in memory when mapped to the AP1000 cells, which is inconvenient for both

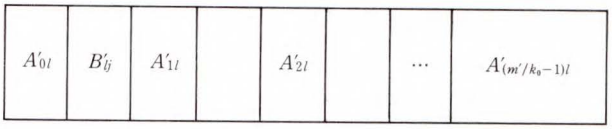


Fig. 1 – The workspace for a partitioned multiply-add operation.

Table 2. Speed in MFLOPs / cell of parallel $C \leftarrow C+AA$ using the non-systolic method on an 4×4 AP1000 with $n \times n$ matrices (single precision)

n/N_x	Partitioning		Strassen's method
	Yes	No	
128	7.18	7.23	7.2 (7.2)
256	7.44	5.4	6.9 (7.9)
384	7.52	5.7	–
512	7.58	5.5	6.8 (8.8)
640	7.60	–	–
728	7.59	–	–
896	7.63	–	–
1 024	7.65	–	6.7 (10.0)

message passing and cache management. Furthermore, the matrix multiply-add operation may involve scaling by constants α and β. Finally, distributed implementations of the BLAS-3 $C \leftarrow \alpha AA^T + \beta C$ imply copying of A cell sub-blocks, even if $\alpha = 1$.

These problems can be most easily overcome by copying (parts of) the A, B and in some cases C sub-blocks into contiguous blocks in a BLAS-3 'workspace' area, where they may then be scaled if necessary. However, the workspace need not be $O(n^2)$ for $n \times n$ matrices; below we present an 'outer product'-based $O(n)$ workspace partitioning method, capable of high asymptotic performance by full utilization of the cache.

Consider an $m \times k$ global matrix A having an $m' \times k'$ (sub-) matrix A' on a particular AP1000 cell, where $m' = m/N_y$, $k' = k/N_x$. Partition A' into $k_0 \times k_0$ sub-blocks denoted A'_{ij} where $0 \leq i < \lceil m'/k_0 \rceil$, $0 \leq j < \lceil k'/k_0 \rceil$ and the optimal block size $k_0 = 128$ (for single precision) is chosen from Table 1. Let B be a $k \times n$ global matrix partitioned in a similar way.

The method involves at step l copying the 'block-column' $A'_{0l}, A'_{1l}, \ldots, A'_{(m'-1)l}$ into a contiguous workspace, for $l = 0, \ldots, k'/k_0 - 1$. On the jth sub-step ($j = 0, \ldots, n'/k_0 - 1$), B'_{lj} is copied to the workspace and is multiplied by each of the $k'/k_0 A'$ sub-blocks already there. The layout of these sub-blocks in the workspace is shown in Fig. 1. Here, one can see that A'_{il} and B'_{lj} map into different areas of the AP1000's 128 Kbyte direct-mapped cache. For this reason, almost half of the workspace is unused. The total size of the workspace is $k_0(m' + n' - 1)$ words per cell, and it can be seen that the cost of copying (with scaling, if needed) a sub-block into the workspace is amortized over the k'/k_0 times it is used to perform a multiply-add.

This idea can easily be integrated into the parallel 'non-systolic' multiply-add, thus amortizing communication costs. The performance of this partitioning method is given in Table 2. As the maximum matrix size corresponds to 4 Mbytes, results for a 4×4 AP1000 are given; however the results for an 8×8 AP1000 appear identical for the corresponding matrix sizes. These results indicate the performance achievable for the BLAS-3 general multiply operation $C \leftarrow \alpha AB + \beta C$, over 90 % of the theoretical peak on the AP1000.

It is possible to use partitioning without workspaces, where the overall matrix multiply is split into a series of sub-multiplications that minimize cache conflicts[10]. However, with a direct-mapped cache this cannot always yield maximum performance (e.g. a $k \times n$ matrix B with $k\mathrm{ld}_B$ exceeding the cache size will mean that some elements in a single column of B will map into the same place in the cache).

4.3 'Fast' methods

The above implementations are all based on standard $O(n^3)$ matrix multiplication algorithms; however, with an 'acceptable' loss of numerical stability (in terms of the BLAS-3 error bounds[11]), it is possible to implement matrix multiply algorithms with a reduced number of arithmetic operations. One such algorithm, Strassen's method[11], has asymptotically $O(n^{2.81})$ operations.

In Strassen's method, matrices are split into 4 sub-matrices; products of the sums and differences of these sub-matrices may be combined in such a way that only 7 (instead of 8) sub-matrix multiplies need be computed. Thus, considerable workspace area is needed. If the matrix dimensions are powers of 2, this process can be easily repeated recursively. However, for $n \times n$ matrices, we have found it more efficient to apply only the first $\log_2\{n/(N_x k_0)\}$ stages of the method, where k_0 is defined in Sec. 4.2, and hence it is only appropriate for large matrices. Table 2 gives the results of our implementation; in parentheses are the MFLOPs rating if $2n^3$ arithmetic operations are assumed. The actual efficiency decreases primarily because the FPU can operate at no more than half speed during the matrix addition and subtraction operations.

4.4 Adaption to a general AP1000 configuration and the BLAS Level 2 limit

We now describe an implementation of $C \leftarrow C + AB$, where C is $m \times n$, A is $m \times k$ and B is $k \times n$, for general $N_y \times N_x$ AP1000 configuration; this implementation is also efficient in the cases where a matrix becomes a vector, hence the term 'BLAS Level 2 limit'.

In these cases, it is important to communicate the smaller of the matrices, so as to reduce communication costs. This may require transposition of the matrix beforehand (cf. the implicit transpose operations of Chap. 4). An efficient matrix transpose operation $A' \leftarrow A^T$ is nontrivial if $N_x \neq N_y$, and involves blocking and permuting matrix segments[10]. Our implementation, for a 1 000 × 1 000 matrix, achieves speeds on a 4 × 8, 7 × 8 and 8 × 8 configurations of (respectively) 1.02, 0.59 and 1.30 Mbytes / s per cell.

The following three algorithms, based on the 'non-systolic' multiply-add of Chap. 4, are each suited to particular matrix shape:

A (for small k) perform k rank-1 updates to C, i.e. $C \leftarrow C + \Sigma A_{\cdot j} B_{j\cdot}$. The cells in column j mod N_x of the AP1000 broadcast $A_{\cdot j}$ horizontally, the cells in row j mod N_y of the AP1000 broadcast $B_{j\cdot}$ vertically. Each cell accumulates a moderate number ω of these broadcasts and then performs a single rank-ω update. The $2k$ broadcast startup overheads involved here can be reduced by grouping if $\mathrm{GCD}(N_y, N_x) > 1$.

B (for small n) transpose B, then broadcast each row of B^T. Each cell computes a local matrix-vector product, and the vector results are summed horizontally.

C (for small m) is simply the dual of **B**

Table 3. Speed (MFLOPs / cell) of matrix multiply-add on rectangular AP1000 configurations (single precision)

m	k	n	4 × 8	7 × 8	8 × 8
1	1 000	1 000	4.1	3.5	3.8
10	1 000	1 000	4.8	4.6	4.6
1 000	1	1 000	3.1	3.0	2.9
1 000	10	1 000	5.7	5.4	5.5
1 000	1 000	1	4.2	3.5	3.8
1 000	1 000	10	5.0	4.6	4.5
1 000	1 000	1 000	6.2	6.4	6.8

In Table 3 we give speeds for the combination of methods **A**, **B**, and **C** on three different configurations. The speed exceeds 50 percent of the theoretical peak speed (8.33 MFLOPs/cell) except for the case $\min(m, n, k) = 1$.

5. Implementing the LINPACK benchmark

Suppose we want to solve a nonsingular n by n linear system:

$$Ax = b, \qquad \cdots\cdots\cdots(1)$$

on an $N_x \times N_x$ AP1000. The augmented matrix $[A \mid b]$ is stored using the scattered representation.

It is known[12),13)] that Gaussian elimination is equivalent to triangular factorization. More precisely, Gaussian elimination with partial pivoting produces an upper triangular matrix U and a lower triangular matrix L (with unit diagonal) such that:

$$PA = LU, \qquad \cdots\cdots\cdots(2)$$

where P is a permutation matrix. In the usual implementation A is overwritten by L and U (the diagonal of L need not be stored). If the same procedure is applied to the augmented matrix

$\overline{A} = [A|b]$, we obtain

$$P\overline{A} = L\overline{U}, \quad \cdots\cdots\cdots(3)$$

where $\overline{U} = [U|\overline{b}]$ and Equation (1) has been transformed into the upper triangular system

$$Ux = \overline{b}. \quad \cdots\cdots\cdots(4)$$

In the following we shall only consider the transformation of A to U, as the transformation of b to $\overline{b}$ is similar.

If A has n rows, the following steps have to be repeated $n - 1$ times, where the kth iteration completes the computation of the kth column of U:

1) Find the index of the next pivot row by finding an element of maximal absolute value in the current (kth) column, considering only elements on and below the diagonal.
2) Broadcast the pivot row vertically.
3) Exchange the pivot row with the kth row, and keep a record of the row permutation.
4) Compute the "multipliers" (elements of L) from the kth column and broadcast horizontally.
5) Perform Gaussian elimination (a rank-1 update using the portion of the pivot row and the other rows held in each cell).

We can estimate the parallel time T_P involved:

$$T_P \simeq \alpha n^3/N_x^2 + \beta n^2/N_x + \gamma n, \quad \cdots\cdots\cdots(5)$$

where the first term is due to the $2n^3/3 + O(n^2)$ floating point operations, the second term is due to the total volume of communication, and the third due to the communication startup (e.g. $O(n)$ row/column broadcasts). The terms are additive as it is difficult to overlap computation with the AP1000's xy communication. As we would expect the time on a single cell to be $T \simeq \alpha n^3$, the efficiency E_P is:

$$E_P \simeq \frac{1}{1+(1+\overline{\gamma}/n')\,\overline{\beta}/n'} \simeq \frac{1}{1+n_{half}/n'} \quad \cdots(6)$$

where $\overline{\beta} = \beta/\alpha$ is proportional to the ratio of computation to communication speed, $\overline{\gamma} = \gamma/\beta$ measures the importance of the communication startup time, $n' = n/N_x$, and $n_{half} = \overline{\beta}N_x$ is the problem size giving efficiency 0.5 (this approximation is valid if $\overline{\gamma}$ is negligible). From Equation (6), the efficiency is close to 1 only if $n' \gg \overline{\beta}$.

We omit details here of the "back- substitution" phase, ie. the solution of the upper triangular system Equation (4), because this can be performed in time much less than Equation (5), (see Refs. 14 and 15): For example, with n = 1 000 on an 8 × 8 AP1000, the back-substitution phase takes 0.1 s as opposed to the LU factorization phase, which takes 3.5 s. A generalization of the back-substitution phase (with the vector b becoming a matrix) will be disussed in Sec. 5.3.

To adapt this algorithm to an $N_y \times N_x$ AP1000 with $N_y = 2N_x$, our *ad hoc* solution was to simulate a $N_y \times N_y$ AP1000 by each physical AP1000 cell simulating two virtual cells in the x-direction. This ensured full processor utilization and optimal communication speed, but due to the significant costs of context switching on AP1000 cells, the simulation was hard coded rather than using two tasks per cell.

5.1 The need for blocking

As discussed in Chap. 3, peak performance cannot be reached using rank-1 updates. It is possible to reformulate Gaussian elimination so that most of the floating-point arithmetic is performed in matrix-matrix multiplications, without compromising the error analysis. Partial pivoting introduces some difficulties, but they are surmountable. The idea is to introduce a "blocksize" or "bandwidth" parameter ω. Gaussian elimination is performed via rank-1 updates in vertical strips of width ω. Once ω pivots have been chosen, a horizontal strip of height ω can be updated. At this point, a matrix-matrix multiplication can be used to update the lower right corner of A. The optimal choice of ω is best determined by experiment, but

$$\omega \simeq n^{1/2}$$

is a reasonable choice, with ω a multiple of N_x.

Here, we take advantage of each AP1000 cell's relatively large memory (16 Mbytes) and save the relevant part of each pivot row and multiplier column as it is broadcast during the horizontal and vertical strip updates. The block update step can then be performed independently in each cell, without any further communication. Each cell requires working storage of about

$2\omega n/N_x$ floating-point words, in addition to the $\{n^2 + O(n)\}/N_x^2$ words required for the cell's share of the augmented matrix and the triangular factors. If $2\omega n/N_x$ exceeds the cache size, partitioning methods for the matrix multiply need to be employed (see Sec. 4.2)

The effect of blocking is to reduce the constant α in Equation (5) at the expense of increasing the lower-order terms. Thus, a blocked implementation should be faster for sufficiently large n, but may be slower than an unblocked implementation for small n. This is what we observed – with our implementation the crossover occurs at $n \simeq 40N_x$.

5.2 Results

The benchmark programs perform Gaussian elimination with partial pivoting (and check the size of the residual). All results are for double-precision. Single-precision is about 50 percent faster.

As discussed in Table 3 of Ref. 16, a gain in efficiency of up to 40 % is achieved by blocking over non-blocking for large matrices. Also, a version of the blocked algorithm was implemented where the AP1000's hardware-supported row/column broadcast and scan operations were simulated in software. This version ran 7 % slower even for large matrices, indicating the need for hardware support for these operations.

The results in Table 4 are for n = 1 000 and should be compared with those in Table 2 of Ref. 17. The results in Table 5 are for n almost as large as possible (constrained by the storage of 16 Mbytes/cell), and should be compared with those in Table 3 of Ref. 17.

The results for the AP1000 are good when compared with reported results for other distributed memory MIMD machines such as the nCUBE, Intel iPSC/860, and Intel Delta, if allowance is made for the different theoretical peak speeds. For example, the 1 024-cell nCUBE 2 achieves 2.59 s for n = 1 000 and 1.91 GFLOPs for n = 21 376[17] with γ_{peak} = 2.4 GFLOPs. Our results indicate that a P-cell AP1000 is consistently faster than a $2P$-cell nCUBE 2. The 512-cell Intel Delta achieves 13.9 GFLOPs but this is less than 70 percent of its theoretical peak of 20 GFLOPs[18]. The 128-cell Intel iPSC/860 achieves 2.6 GFLOPs, slightly more than the 512-cell CAP, but this is only 52 percent of its theoretical peak of 5 GFLOPs. For large n the AP1000 consistently achieves in the range 79 to 82 percent of its theoretical peak (with the ratio slightly better when the number of cells is a perfect square, e.g. 64 or 256, than when it is not).

Table 4. LINPACK benchmark results for n = 1 000

Time for one cell	Cells	Time (s)	Speedup	Efficiency
160	512	1.10	147	0.29
160	256	1.50	108	0.42
160	128	2.42	66.5	0.52
160	64	3.51	46.0	0.72
160	32	6.71	24.0	0.75
160	16	11.5	13.9	0.87
160	8	22.6	7.12	0.89
160	4	41.3	3.90	0.97
160	2	81.4	1.98	0.99

Table 5. LINPACK benchmark results for large n

Cells	r_{max} GFLOPs	n_{max} order	n_{half} order	r_{peak} GFLOPs	r_{max}/r_{peak}
512	2.251	25 600	2 500	2.844	0.79
256	1.162	18 000	1 600	1.422	0.82
128	0.566	12 800	1 100	0.711	0.80
64	0.291	10 000	648	0.356	0.82
32	0.143	7 000	520	0.178	0.80

n_{max} : the problem size giving the best performance r_{max}

n_{half} : the problem size giving performance $r_{max}/2$

r_{peak} : the theoretical peak performance (ignoring everything but the speed of the floating-point units)

An encouraging aspect of the results is that the AP1000 has relatively low n_{half}. For example, on the 64-cell AP1000 at ANU we obtain at least half the maximum performance (i.e. at least 145 MFLOPs) for problem sizes in the wide range $648 \leq n \leq 10\,000$. (On the 64-cell Intel Delta, the corresponding range is $2\,500 \leq n \leq 8\,000$[18]). As expected from Equation (6), n_{half} is roughly proportional to $P^{1/2}$.

Because of the influence of the cache and the effect of blocking, the Equation (5) gives a

good fit to the benchmark results only if n is sufficiently small and ω is fixed (or blocking is not used).

5.3 Optimizations for BLAS-3 triangular matrix updates

If B is an $m \times n$ matrix, to form $B \leftarrow A^{-1}B$, where A is an $m \times m$ upper triangular matrix with unit diagonal, we can perform the corresponding (parallel) rank-1 updates:

$B \leftarrow B - \tilde{A}_{\cdot j} B_{j\cdot}$, for $j = m - 1, \cdots\cdots, 1$,

where $\tilde{A} = A - I$. A straightforward ('unblocked') implementation on the AP1000 uses row/column broadcasts and rank-1 updates. However, performance can be improved by grouping ω updates together, as described in Sec. 5.1.

Table 6 gives results for this computation for single precision, with $\omega = 4N_x\sqrt{n/(2N_x)}$. For the unblocked algorithm, the performance does not even approach that of Rank1Update( ), due to communication overheads (for small n) and the fact that rank-1 update is a Level 2 operation and hence makes poor use of the cache (for large n). For the blocked algorithm, performance is better but still does not approach that of UpdateRect( ), due to fact that the optimal ω is a tradeoff between seeking a higher proportion of the computation in UpdateRect( ) (needing a low ω) and seeking a high number of iterations in each call to UpdateRect( ) (needing a high ω).

Table 6. Speed in MFLOPs / cell for $B \leftarrow A^{-1}B$ for $n \times n$ matrices on the AP1000 (single precision), for $N_x = 1, 2, 8$

n/N_x	Unblocked			Blocked			Super-blocked		
	1	2	8	1	2	8	1	2	8
32	4.6	3.8	3.8	4.0	3.8	4.0	4.2	3.6	3.7
64	5.4	5.0	5.0	5.6	5.6	5.6	5.8	5.3	5.5
128	5.2	5.1	5.1	6.3	6.4	6.4	6.6	6.4	6.5
180	5.5	5.4	5.4	6.5	6.4	6.6	6.8	6.7	6.8
256	4.3	4.2	4.3	6.7	6.8	6.2	6.8	6.8	6.9
360	3.8	3.7	3.7	6.8	6.9	6.5	7.1	7.1	7.2

A value of $\omega \simeq k_0$ (Sec. 4.2) is optimal for UpdateRect( ). The tradeoff mentioned above can be overcome by recursively applying the blocking process described in Sec. 5.1, for $\omega \simeq k_0$, $k_0/2$, $k_0/4$, etc. As larger values of ω are now used, the partitioning methods of Sect. 4.2 must also be employed. The performance for moderate sized matrices of this 'super-blocked' scheme is given in Table 6; for larger matrices, performance steadily improves up to 7.3 MFLOPs for $n/N_x = 1\,024$. These results indicate that the AP1000 can perform BLAS-3 triangular matrix updates at 85 % of the its theoretical peak speed.

While the coding of such a recursive blocking scheme is complex, it could be similarly applied to the more complex LINPACK benchmark, with similar improvements in performance to be expected.

6. Conclusion

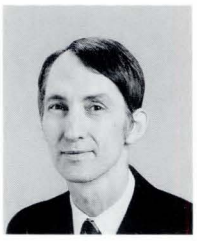
In this paper, we have described implementations of the BLAS-3 and the LINPACK benchmark on the Fujitsu AP1000. Many of the techniques presented, such as the design of SPARC BLAS-2 and BLAS-3 kernels (Chap. 3), partitioning methods for direct-mapped caches (Sec. 4.2), and blocking (Secs. 5.1 and 5.3) are also applicable to the implementation of other linear algebra applications, on the AP1000 and on similar architectures.

The LINPACK benchmark and BLAS-3 results show that the AP1000 is a good machine for numerical linear algebra, and that on moderate to large problems we can consistently achieve close to 80 % of its theoretical peak performance, for the former, and 85-90 % for the latter. They signify that the AP1000 architecture is well balanced on all levels, with respect to floating point computation. The main reason for this is the high ratio of communication speed to floating-speed compared to machines such as the Intel Delta and nCUBE. The high-bandwidth hardware row/column broadcast capability of the AP1000, extremely useful in linear algebra applications, and the low latency of the send/receive routines are also significant. As shown in Table 1, the speed of the former make the use of 'systolic' versions of linear algebra algorithms

unnecessary. The large, direct-mapped cache, while requiring extra effort for full optimization, and the large cell memory are also very important features.

References

1) Proc. CAP Workshop '91. ed. Brent, R. P., Australian Natl. Univ., Canberra, 1991.
2) Proc. 1st CAP Workshop. ed. Brent, R. P., and Ishii, M., Kawasaki, 1990.
3) Ishihara, H., Horie, T., and Shimizu, T.: An Architechure for the AP1000 Highly Parallel Computer. *FUJITSU Sci. Tech. J.*, **29**, 1 (Special Issue on Cellular Array Processor AP1000), pp. 6-14 (1993).
4) Horie, T., and Ikehara, M.: AP1000 Software Environment for Parallel Programming. *FUJITSU Sci. Tech. J.*, **29**, 1 (Special Issue on Cellular Array Processor AP1000), pp. 25-31 (1993).
5) Dongarra, J. J., and Du Croz, J. J., Hammarling, S. J., and Duff, I. S. : A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, **16**, pp. 1-17 (1990).
6) Dongarra, J. J., and Du Croz, J. J., Hammarling, S. J. and Hanson, R.: An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, **14**, pp. 1-17 (1988).
7) Dongarra, J. J.: LAPACK Working Note 34: Workshop on the BLACS. Tech. Rep., CS-91-134, Univ. Tennessee, 1991.
8) Sun Microsystems: Sun-4 Assembly Language Reference Manual. 1988.
9) Strazdins, P. E., and Brent, R. P.: Implementing BLAS level 3 on the CAP-II. Proc. 1st CAP Workshop., ed. Brent, R. P., and Ishii, M., Kawasaki, 1990.
10) Strazdins, P. E., and Brent, R. P.: Implementing BLAS level 3 on the AP1000. Proc. CAP Workshop '91. ed. Brent, R. P., Australian Natl. Univ., Canberra, 1991.
11) Demmel, J. W., and Higham, N. J.: Stability of Block Algorithms with Fast Level 3 BLAS. *ACM Trans. Math. Software*, **18**, pp. 274-291 (1992).
12) Golub, G. H., and Van Loan, C.: Matrix Computations. Johns Hopkins Press, Baltimore, Maryland, 1983.
13) Stewart, G. W.: Introduction to Matrix Computations. Academic Press, N. Y., 1973.
14) Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W.: "Solving Problems on Concurrent Processors". General Techniques and Regular Problems, **Vol. I**, Prentice-Hall, New Jersey, 1988.
15) Li, G., and Coleman, T. F.: A new method for solving triangular systems on distributed-memory message-passing multiprocessors. *SIAM J. Sci. and Statist. Computing*, **10**, pp. 382-396 (1989).
16) Brent, R. P.: The LINPACK Benchmark on the Fujitsu AP1000. Proc. Frontiers '92, IEEE, Virginia, 1992, pp. 128-135.
17) Dongarra, J. J.: Performance of various computers using standard linear equations software. Rep. CS-89-05, Comput. Sci. Dep. Univ. Tennessee, 1991.
18) van de Geijn, R. A.: Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems. Rep. TR-91-92, Dept. Comput. Sci. Univ. Texas, 1991.



Richard P. Brent

Computer Sciences Laboratory
Australian National University
Bachelor of Science
Monash University 1968
Ph. D. in Computer Science
Stanford University 1971
Specializing in Numerical Algorithms and Parallel Computation

Peter E. Strazdins

Department of Computer Science
Australian National University
Bachelor of Science
University of Wollongong 1984
Master of Science (Computation)
University of Oxford 1985
Ph. D. in Computer Science
Australian National University 1990
Specializing in Parallel Computing

UDC 621.3.049.771:681.322

Highly Parallel Circuit Simulator on the AP1000: PARACS

● Tetsuro Kage ● Junichi Niitsuma ● Kumiko Teramae

(Manuscript received August 5, 1992)

This paper presents a parallel circuit simulator called PARACS that runs on the AP1000 parallel computer. In parallel circuit simulation, a circuit is partitioned into the same number of subcircuits as processors to produce a bordered-block-diagonal (BBD) matrix. A new parallel BBD matrix solution is devised to achieve high-speed parallel circuit simulation. The interconnection block of the BBD matrix as well as the diagonal blocks are solved in parallel using all processors. The PARACS simulator implements this approach, and simulated a test circuit of 3 192 transistors on the AP1000 up to 10.6 times faster than on an S4/1+ workstation.

1. Introduction

Circuit simulation is one of the most critical, time-consuming tasks in VLSI circuit design, especially for large memory VLSI circuits. One simulation may require several hours or even days on a mainframe. Before manufacture, additional simulations must be performed to guarantee that the circuit will function correctly and meet the specifications over a wide range of process variations.

Circuit simulators can be divided into two types: direct and relaxation-based. Direct method simulators such as SPICE[1] are reliable and accurate, but slow and limited. Circuit designers are restricted by the number and size of simulations which can be performed due to the time required and the available computer resources. On the other hand, relaxation-based simulators[2] are impressive for certain classes of circuits, but are of limited use as a general-purpose simulation tool.

High-performance direct method simulation is possible with the new generation of parallel processing systems. Some experimental parallel circuit simulators have been successfully developed on parallel processing systems with less than a dozen processors[3)-5]. To obtain better performance than a mainframe, a highly parallel circuit simulator on a highly parallel processing system is needed.

A severe problem arises, however, with highly parallel circuit simulation. For parallel simulation, the circuit must be partitioned into the same number of subcircuits as processors to produce a bordered-block-diagonal (BBD) matrix. Block diagonals are easily solved in parallel, but the interconnection (IC) matrix of a BBD matrix is not so easy. Unfortunately, highly parallel circuit simulation increases the size of the IC matrix.

We have developed a parallel circuit simulator called PARACS that runs on the AP1000 parallel computer. To achieve high-speed circuit simulation, we devised an approach to solving the BBD matrix in parallel: both the diagonal blocks and the IC matrix of a BBD matrix are solved in parallel using all processors.

In this paper, first we present circuit partitions and discuss problems with highly parallel circuit simulation. Next, we describe our approach to solving a BBD matrix in parallel. The program structure of PARACS which implements this approach is introduced. Finally, some statistics on parallel simulation times for the AP1000 are shown.

2. Circuit simulation outline

2.1 Flow

A system composed of electrical elements is described by a set of nonlinear, differential algebraic equations of the form:

$$f(dx/dt, x, t) = 0. \quad \cdots\cdots(1)$$

At the new time point of analysis t_{n+1}, stiffly stable integration formulas are used to make the analysis discrete. This process yields a set of nonlinear, algebraic difference equations of the form:

$$g(x) = 0, \quad \cdots\cdots(2)$$

where x is the vector of node voltages at t_{n+1}.

These equations are solved using a damped Newton-Raphson algorithm to yield a set of sparse linear equations of the form:

$$Ax = b, \quad \cdots\cdots(3)$$

where A is a matrix related to the Jacobian of $g(x)$. In general, less than a few percent of the entries in A are nonzero. These equations are then solved using direct methods for solving linear equations such as the sparse LU (Lower and Upper triangular matrices) decomposition algorithm.

The major tasks of the simulation include model evaluation for transistors, Jacobian loading, matrix solving, and truncation error evaluation for the next time step. Of these, about 70 percent to 80 percent of the simulation time is spent on model evaluation and Jacobian loading, whereas about 10 percent is taken for solving the matrix.

As the circuit size grows, it takes longer to solve the matrix. The time required for the matrix solution phase has been found in experiments to increase as $O(N^{1.5})$, where N is the number of equations. However, the time required for model evaluation and Jacobian loading increases linearly with the number of transistors.

2.2 Node tearing approach

The node tearing approach[6] is well suited to parallel circuit simulation on a message-based parallel system, because the amount of information that must be passed between processors is relatively small. The basic idea of node tearing is to divide the circuit into a set of subcircuits,

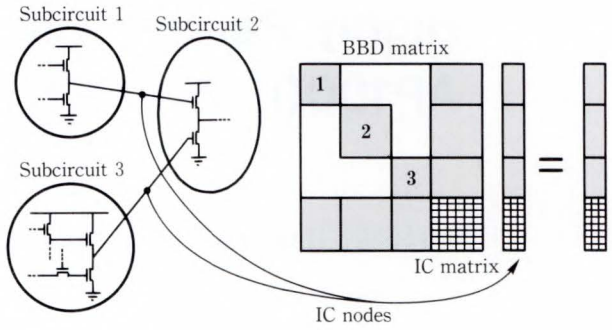


Fig. 1 – A circuit partition and its BBD matrix.

producing a BBD matrix as shown in Fig. 1. Each diagonal block represents a subcircuit, and the bottom block represents the interconnection between subcircuits.

A BBD matrix can be solved by the following formula[6]:

$$v_0 = Y_0^{-1} i_0 - Y_0^{-1} C_1 (C_2 Y_0^{-1} C_1 - Y_1)^{-1} (C_2 Y_0^{-1} i_0 - i_1), \quad \cdots\cdots(4)$$

where Y_0, C_1, C_2, and Y_1 are admittance matrices, v_0 is a node voltage vector, and i_0 *and* i_1 are current source vectors.

This formula is evaluated by the following procedure:

Step 1: Solve each subcircuit independently.
Step 2: Gather step 1 results from all processors to form an IC matrix.
Step 3: Solve the IC matrix.
Step 4: Broadcast step 3 results to all processors of each subcircuit.
Step 5: Superimpose the IC matrix effect on each subcircuit.

This procedure makes the model evaluation and Jacobian loading in step 1 run in parallel without problems. The success of parallel simulation depends on two critical considerations: the size of the IC matrix and the load balance between subcircuits. The size of the IC matrix should be kept as small as possible to reduce the number of communication tasks. The granularity of subcircuits must also be well balanced; otherwise, the parallel gain will be limited.

3. Parallel circuit simulation

3.1 Circuit partitioning

In parallel circuit simulation, a circuit

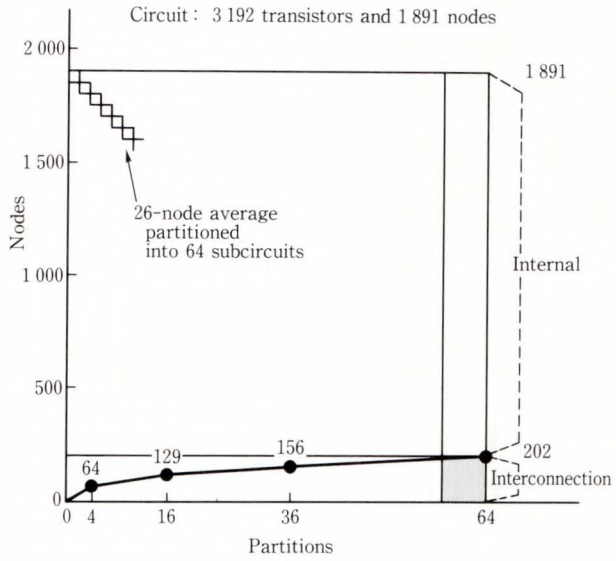
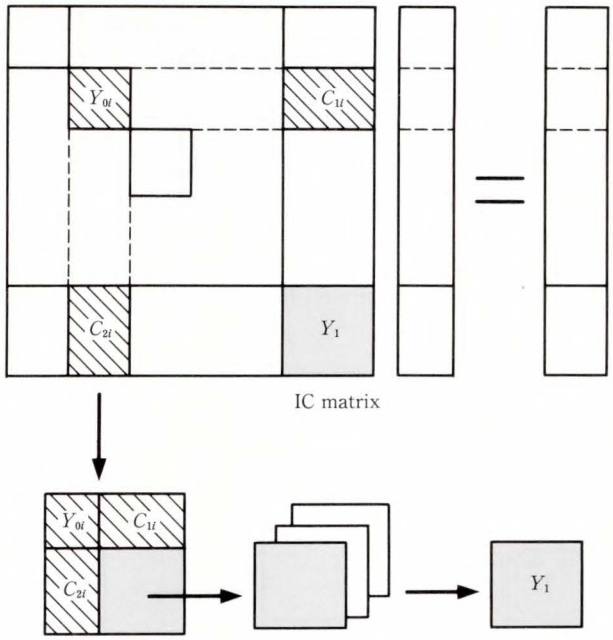


Fig. 2 – Interconnection nodes increase with circuit partitions.



a) Discrete solution of each subcircuit

b) Superimposition from all subcircuits

c) Dispersed solution of the IC matrix

Fig. 3 – Parallel BBD matrix solution

should be partitioned to minimize the IC matrix size while keeping the numbers of transistors in the subcircuits almost equal.

We developed a circuit partition program in which transistors are clustered based on interconnection strength and the number of transistors in a cluster. Once the number of clusters equals the number of processors, transistors are moved between clusters to balance the numbers of transistors in the subcircuits.

A sample circuit with 3 192 transistors and 1 891 nodes was partitioned into 4, 16, 36, and 64 subcircuits. As can be seen in Fig. 2, the number of IC nodes increases with the number of partitions. When the circuit is partitioned into 64 subcircuits, the size of the IC matrix is 202, while the average size of the diagonal matrix is 26. This implies that the time taken to solve the IC matrix as part of the BBD matrix solution is too long to be neglected, and becomes more critical in highly parallel circuit simulation.

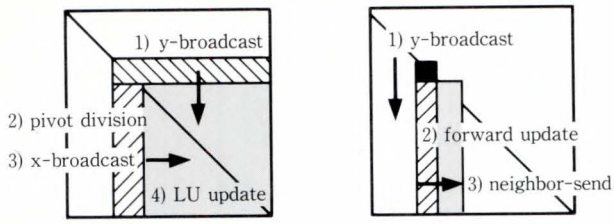
3.2 Parallel BBD matrix solution

As noted previously, the key to highly parallel circuit simulation is the parallel solution of the IC matrix of a BBD matrix. It is also important to gather the entries of the IC matrix efficiently through processor communication.

Our approach features a data structure for parallel subcircuit processing. As shown in Fig. 3, i-th processor contains a diagonal block ($Y_{0\,i}$) with its border blocks ($C_{1\,i}$, $C_{2\,i}$) by appending the IC node vector to its internal node vector. We call this a subcircuit matrix. This matrix enables us to execute subcircuit model evaluation and Jacobian loading, and to solve the subcircuit matrix independently.

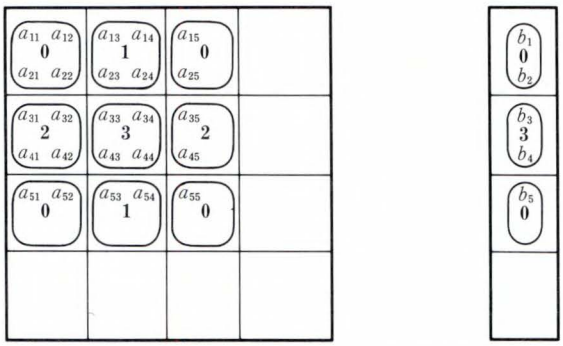
Our approach to parallel matrix solution involves two steps. In the first step, the diagonal block of a subcircuit matrix within a processor is solved using the LU-decomposition algorithm. During this procedure, the fill-ins from the border blocks to the IC matrix are sent to the designated processor asynchronously. The IC matrix is formed by superimposing the fill-ins from all subcircuits. We call this procedure "discrete parallel solution" for a subcircuit matrix.

In the second step, the IC matrix is solved by applying the "dispersed parallel solution" using all processors. Nonzero entries in the IC matrix are allocated to all processors in dot-wise order. The IC matrix is solved, pivot by pivot, by



a) LU decomposition b) Forward substitution

Fig.4 – Dispersed solution of an IC matrix.



(0, 1, 2, 3 : Processor number)

a) Interconnection matrix b) Right-hand-side vector

Fig. 5 – Multi-pivot allocation of an IC matrix into all processors.

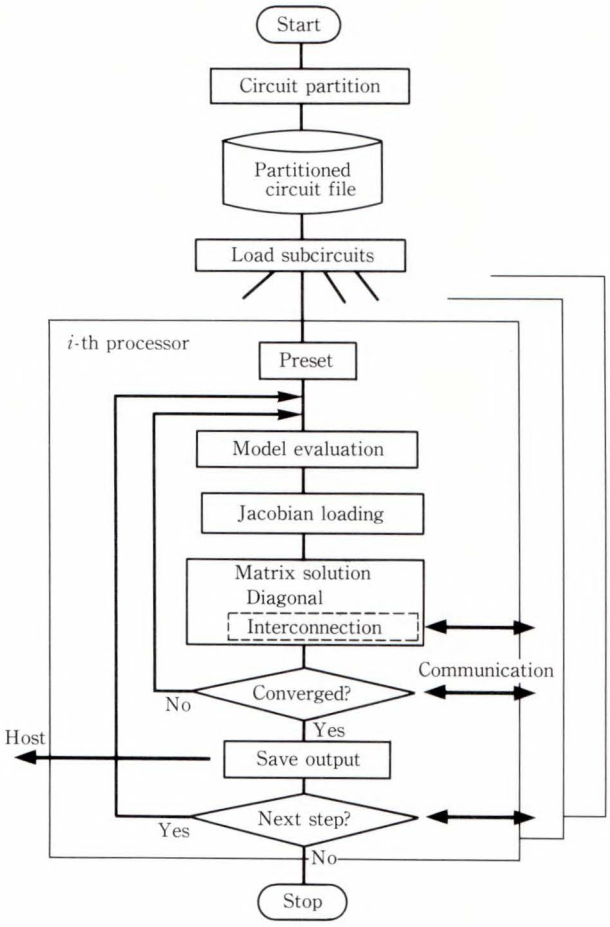


Fig. 6 – Parallel circuit simulation flow.

broadcasting a sparse row/column vector to the next row/column as shown in Fig. 4. The calculation tasks between processors must be well balanced because the entries of an IC matrix are allocated in dot-wise order.

3.3 Multi-pivot allocation

The details of the time required to solve a dense 1 000 by 1 000 matrix in a dot-wise parallel matrix solution on the AP1000 have been reported elsewhere[7]. The time required for interprocessor communication accounts for 28 percent of the total solution time during LU decomposition and 67 percent during forward and backward substitutions. In circuit simulation, an IC matrix is sparse (70-80 percent are zeros) and the size is usually in the order of several hundred. Communication speed is thus critical.

To reduce communication time, we allocate an IC matrix to processors in block-wise order as shown in Fig. 5. The IC matrix allocated in this manner is solved by using the LU-decomposition algorithm shown in Fig. 4. The calculations for LU decomposition, and forward and backward substitutions are performed on a block in a processor, and then the updated block is sent to the next processor. We call this "multi-pivot allocation."

Multi-pivot allocation reduces the communication time by a factor of between $6N$ to $7N/n$ at each matrix solution, where N is the size of the IC matrix and n is the size of the allocation blocks. The allocation block size can be changed to balance the calculation speed of a processor with the block data communication capacity of a parallel computer system.

We do not require a host or special co-processor to solve the IC matrix. Our approach uses all of the processors in parallel, and there are no communication bottlenecks.

3.4 PARACS program structure

We developed a new parallel circuit simulator, PARACS, that runs on the AP1000. The program structure of PARACS is shown in Fig. 6. The circuit to be simulated is partitioned into subcircuits automatically, and the subcircuits are loaded onto the processors for parallel processing. Each subcircuit is simulated by a direct method, general-purpose circuit simulator that was originally developed for mainframes. The simulation flow for a subcircuit is the same as a serial circuit simulation, as outlined in Sec. 2.1.

PARACS implements the parallel BBD matrix solution described above. Processors must be synchronized to solve the IC matrix, to check Newton-Raphson iteration convergence, and to select the next time step. The outputs from all processors are sent to the host at each time step.

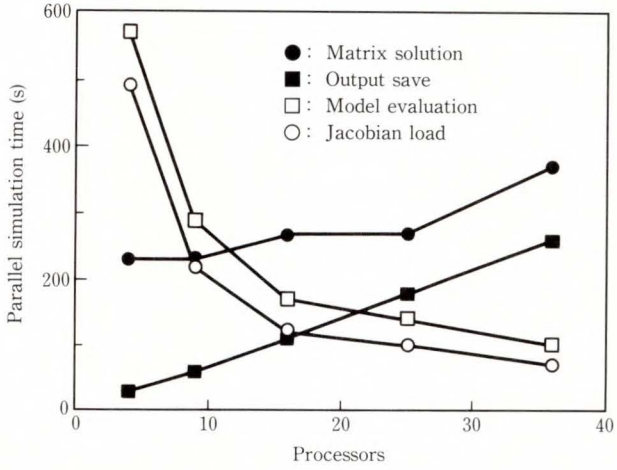
4. Results

Some test circuits were simulated by PARACS on the AP1000. The host system was a SUN4/330 workstation.

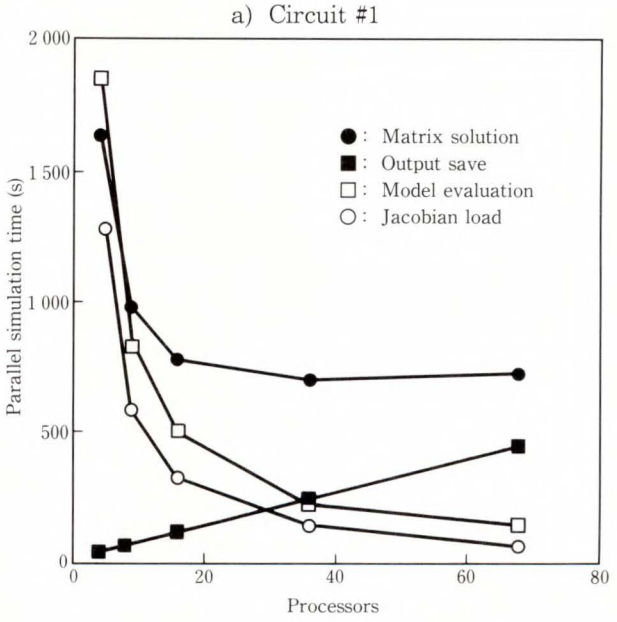
One of the test circuits, Circuit #1, was a control circuit of a memory LSI with 297 nodes. This circuit contains 544 transistors, as well as capacitors and resistors. The simulation was for an operation period of 250 ns. There were 9 502 Newton-Raphson iterations to simulate during the period. Another circuit, Circuit #2, containing 3 192 transistors was simulated for an operation period of 4 000 ns with 4 275 Newton-Raphson iterations.

The parallel simulation statistics for Circuit #1 and Circuit #2 are listed in Table 1. To provide a reference, these circuits were also simulated by the original circuit simulator using serial codes on an S4/1+ workstation. The performance of the S4/1+ is almost the same as on a single processor of the AP1000. The highest increase in speed was 10.6 times, which was obtained for Circuit #2 using 36 processors.

Figure 7 details parallel simulation times.



a) Circuit #1



b) Circuit #2

Fig. 7 – Parallel circuit simulation time details.

Table 1. Parallel simulation results

	Transistors	Iterations	Serial on S4/1+	PARACS on the AP1000 (Number of processors)					
				4	9	16	25	36	64
Circuit #1 (speedup)	544	9 502	1 h 18 m 43 s	23 m 12 s (×3.4)	14 m 57 s (×5.3)	13 m 29 s (×5.8)	14 m 21 s (×5.5)	15 m 47 s (×5.0)	
Circuit #2 (speedup)	3 192	4 275	4 h 18 m 37 s	1 h 9 m 56 s (×3.7)	42 m 35 s (×6.1)	30 m 1 s (×8.6)	27 m 10 s (×9.5)	24 m 20 s (×10.6)	26 m 42 s (×9.7)

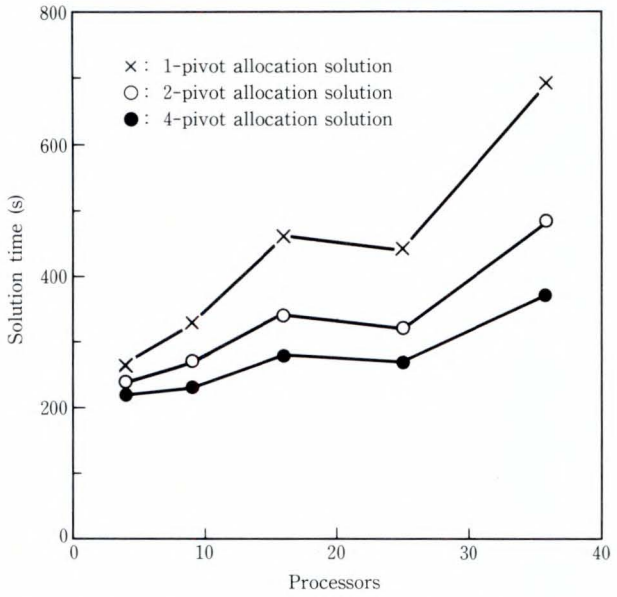


Fig. 8 – Parallel matrix solution times for Circuit #1 when the allocation size is changed.

This figure shows that the times for both model evaluation and Jacobian loading decrease with the number of processors. The effectiveness of parallel processing for solving the matrix is demonstrated by Circuit #2: the bigger the circuit, the greater the benefit, and the times for saving output increases linearly. This can be reduced by sending outputs in pipeline from processors at each time step.

The times required to solve the matrix for Circuit #1 for various allocation block sizes are shown in Fig. 8. The sizes allocated were 1 by 1, 2 by 2, and 4 by 4. This figure shows how effectively multi-pivot allocation works in solving the IC matrix, especially in highly parallel processing.

5. Conclusion

Highly parallel circuit simulation was tested on the AP1000 parallel computer by partitioning circuits into subcircuits to produce a BBD matrix. The critical part of the simulation was found to be the solution of the IC matrix of the BBD matrix.

To attain high-speed circuit simulation on a highly parallel computer, we devised a parallel BBD matrix solution which solves not only the diagonal blocks but also the IC matrix in parallel. We allocated an IC matrix block-wise into all processors to balance the calculation speed of the processors with the block data communication capacity of the parallel computer system.

A new parallel circuit simulator, PARACS, was developed to implement this approach. A test circuit with 3 192 transistors was simulated on the AP1000 (36 processors), and the simulation speed was 10.6 times faster than on an S4/1+ workstation. We plan to simulate circuits with several tens of thousands of transistors to further evaluate the benefits of highly parallel circuit simulation.

6. Acknowledgement

We would like to thank Dr. S. Watanabe of the University of Electric Communications for his advice on parallel circuit simulation.

References

1) Nagel, L. W.: SPICE2: A Computer Program to Simulate Semiconductor Circuits. Memo. No. ERL-M520, Berkeley, Univ. of Cal., 1975.
2) Newton, A. R., and Sangiovanni-Vincentelli, A. L.: Relaxation-Based Electrical Simulation. *IEEE Trans. Comput.-Aided Des.*, **CAD-3**, 4, pp. 308-331 (1984).
3) Nakata, T., Tanabe, N., Onozuka, H., Kurobe, T., and Koike, N.: A Multiprocessor System for Modular Circuit Simulation. Proc. IEEE 1987 Int. Conf. Comput. Aided Des., pp. 364-367.
4) Yuan, C. P., Lucas, R., Chan, P., and Dutton, R.: Parallel Electric Circuit Simulation Program. Proc. IEEE 1988 Custom ICs Conf., pp. 6.5.1-6.5.4.
5) Cox, P. F., Burch, R. G., Hocevar, D. E., Yang, P., and Epler, B. D.: Direct Circuit Simulation for Parallel Processing. *IEEE Trans. Comput.-Aided Des.*, **CAD-10**, 6, pp. 714-725 (1991).
6) Hachtel, G. D., and Sangiovanni-Vincentelli, A. L.: A Survey of Third-Generation Simulation Techniques. Proc. IEEE, **69**, 10, pp. 1264-1280 (1981).

7) Horie, T., Ishihata, H., Shimuzu, T., and Ikesaka, M.: AP1000 Architecture and Performance of LU Decomposition. Proc. 1991 Int. Conf. Par. Proc., pp. 634-635.

Tetsuro Kage

VLSI Systems Laboratory
FUJITSU LABORATORIES,
ATSUGI
Bachelor of Electrical Eng.
Kyushu Institute of Technology 1974
Master of Electrical Eng.
Kyushu Institute of Technology 1976
Specializing in VLSI CAD

Junichi Niitsuma

VLSI Systems Laboratory
FUJITSU LABORATORIES,
ATSUGI
Bachelor of Electrical Eng.
Hokkaido University 1984
Specializing in Circuit Simulation

Kumiko Teramae

Advanced CAD Engineering Dept.
Software Development Div.
FUJITSU LIMITED
Bachelor of Education
Yokohama National University 1984
Specializing in Circuit Simulation

UDC 621.3.049.771:681.323

LSI Mask Data Processing System: PRANCER

● Ryo Tsujimura, ● Yasuo Manabe, ● Kazumasa Morishita

(Manuscript received August 17, 1992)

A parallel mask data conversion software system to be run on AP1000 is proposed. To accomplish this, two new algorithms are introduced: one is to balance the load of each processor, and the other is to minimize communication overhead. This parallel system remarkably increases performance because all processing parts can be run in parallel except I/O and some small sequential parts. The CPU performance is over 20 times faster than existing Fujitsu M-780 systems.

1. Introduction

An electron beam exposure system is widely used for LSI mask making, or direct writing for ASICs. Prior to electron beam exposure, the pattern data has to be converted from CAD data format (GDS II stream format) to the electron beam exposure format[1].

With the increasing complexity of LSI circuits, enormous CPU time will be required for data conversion. At the current state of CAD tool performance, tasks such as mask-making have proven to be expensive bottlenecks in the VLSI design process. If the advances in the complexity of VLSI chips are to keep pace with advances made in VLSI process technology, then we must make substantial improvements to the software tools used to design those chips[2].

To accomplish mask data conversion rapidly, we developed a mask data conversion system on a multi-processor. The high performance of a 32-bit MPU is favorable because of its low cost. However, the expected high performance of the multi-processor cannot always be obtained because of its communication and I/O overhead. Furthermore, the reduction of sequential operations in parallel processing with this multi-processor presents a significant problem. Thus, to attain high performance, we introduce two new algorithms: one is to balance the load of each processor, and the other is to minimize communication overhead.

In this paper, we describe a developed mask data conversion system on a multiple instruction multiple data stream (MIMD) type parallel processor, PRANCER (PaRAllel processiNg system for mask data ConvERsion). We describe the method for a parallel mask data conversion in Chap. 2. The results of our parallel mask data conversion when applied to VLSI are shown in Chap. 3.

2. Mask data conversion on MIMD parallel processor AP1000

2.1 System overview

PRANCER works on Fujitsu's highly-parallel computer, the cellular-array processor, the AP1000. Processing elements, called cells, are configured in a two-dimensional array. The system can be increased to up to 1 024 cells.

All cells and the host computer are connected by a common bus. This enables efficient data transfer by broadcasting from the host computer to all cells or from a single cell to all the other cells. All cells are identical. Each cell is a microprocessor system with four specially designed VLSI chips, a 32-bit RISC SPARC (Scalable Processor Architecture) processor and a high-speed Floating-Point Processor Unit (FPU). Its peak performance is 15 MIPS

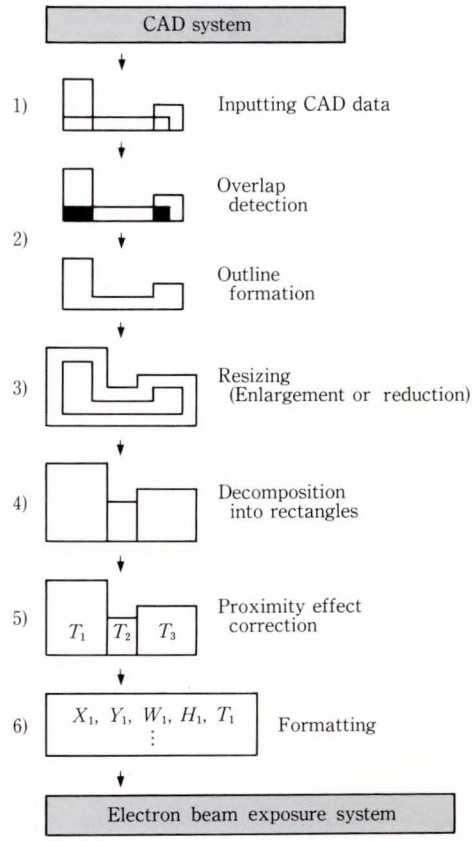


Fig. 1 – General flow of data conversion.

12.5 MFLOPS.

Mask data, as shown in Fig. 1, are converted by the following procedure;

1) inputting CAD data (GDS II),
2) removal of overlaps between shapes to prevent multiple exposure,
3) resizing, such as enlargement or reduction, to compensate for process biases,
4) decomposing polygons into rectangles for use with a rectangular beam,
5) correction of the exposure to compensate for the proximity effect[3], and
6) formatting for the exposure machine.

In general, CAD data has a hierarchical structure. CAD data is composed of a group of several figures and the nested reference of a group of several figures. Nested reference figures are placed on other figures. When a nested reference figure is placed, additional transformations such as mirroring, rotating, and scaling act on it. This action makes the CAD data compact.

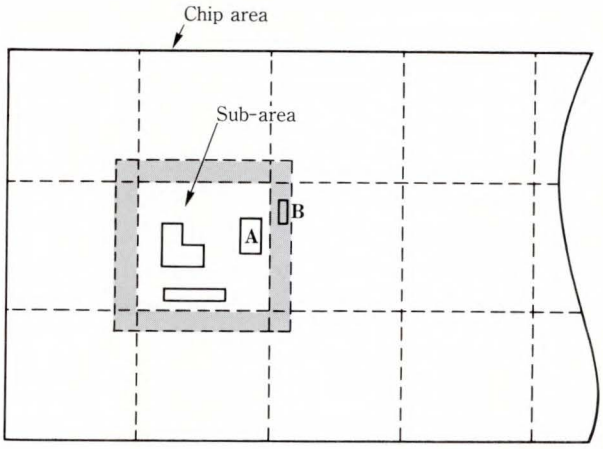


Fig. 2 – Subdivision of a chip area into sub-area

The hierarchical CAD data is input into a host processor. The host processor partially expands the hierarchical construction, and makes data groups. The data are broadcast to each cell according to the number of patterns in the data group. Each cell expands the hierarchical construction. Then, each cell executes mask data conversion as shown in Fig. 1, and broadcasts the result to the host processor. The host processor outputs the result received from each cell.

Consequently, massively parallel systems can be utilized because all processing parts can be run in parallel except communication, I/O, and a small sequential part.

2.2 Method of mask data conversion on a parallel processor

In a multi-processor system, to achieve efficient parallel processing, the loading among cells must be balanced and the communication among cells minimized.

In steps 2) to 6) in Fig. 1, to implement mask data conversion on a parallel processor, we divide the chip area into sub-areas by boundary lines as shown in Fig. 2. We divide the chip area so each cell can process one data group independently of the other cells. However, the information on the boundary lines between neighboring cells must be communicated. Otherwise, some data will not be converted correctly. For example, suppose pattern **A** exists near the shaded area around the sub-area. In

this case, the influence of neighboring pattern **B** outside the sub-area must be considered when removing the overlaps and making the proximity effect correction. As a result, communication overhead will increase.

Thus, to minimize communication and convert mask pattern data correctly, the cell handles pattern data in the overlapped area and the divided area when each cell converts the mask data in each divided area.

By this method, pattern data near the boundary line will be converted correctly. Also, overhead will be reduced since communication is minimized.

2.3 Balancing the load of each processing element

One cell processes one data group in the divided sub-area. To balance the load among cells, the data groups must be divided so each cell processor processes as many equivalent number of patterns as possible. The data groups are created by dividing the sub-area into small variable areas according to the number of pattern data in the sub-area.

2.4 Parallel processing proximity effect correction

When a resist on a substrate is exposed to an electron beam, electrons scatter in the resist and are reflected back from the substrate. This electron scattering in the resist and the substrate, commonly known as the proximity effect[3] in electron beam lithography, leads to undesired pattern sizes and is a serious problem when patterns are closely spaced.

The proximity effect must be compensated for in order to achieve accurate delineation of patterns with dimensions and spacings below 1 μm. The proximity effect is generally compensated for by adjusting the exposure[4] and pattern size[5]. The problem with proximity effect correction is that the computation process is computer resource intensive, and requires a long computing time, a large memory, and disk storage. Proximity effect correction consists of four steps as shown in Fig. 3.

Exposure intensity for each pattern is

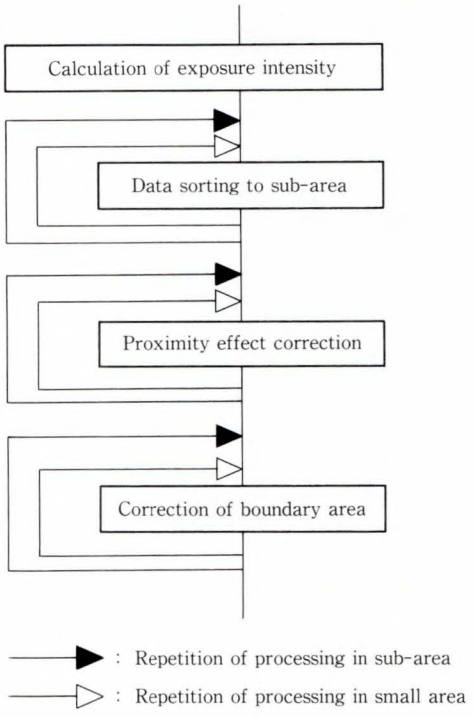


Fig. 3 – Flow of proximity effect correction.

calculated by integrating over the irradiated area using the double Gaussian proximity function[3]. The calculated exposure intensity is then tabulated[6]. The correction value for the proximity effect is obtained by this table.

The chip area is divided into sub-areas when the correction value for all patterns in the chip area is determined. The influence of neighboring patterns leads to undesired pattern sizes because of electron scattering in the resist and the substrate. Note that the influence of pattern in the boundary area must be considered when the pattern exists near the sub-area.

Figure 4 shows the structure of the paralleled proximity effect correction program (PROX-AP).

Pattern data are extracted into the correction area and the boundary area from the original data. Extracted pattern data are broadcast to the cell processor as pattern data dealt in the cell processor. The cell program is the same in all cell processors.

The proximity effect correction procedure in the PRANCER system is as follows.

1) To balance the loads among processing elements, the sub-area is divided into a small

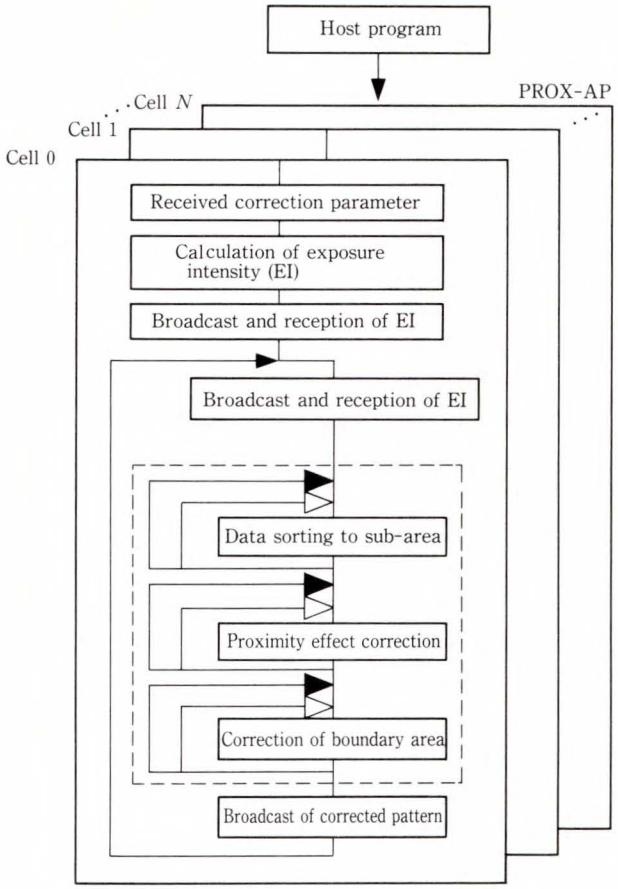


Fig. 4 – Structure of parallel the proximity effect correction program (PROX-AP).

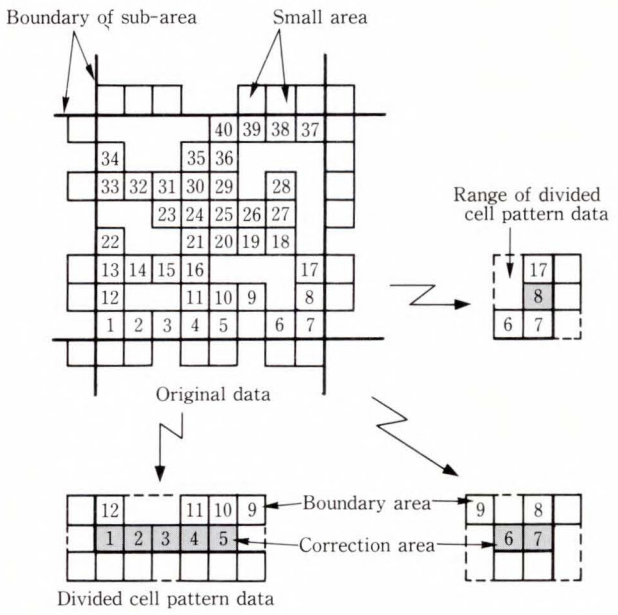


Fig. 5 – Division method of cell pattern data.

area as shown in Fig. 5. Cell pattern data, which is the pattern data in the cell processor, consist of pattern data in several small areas. The number of small areas varies (see Fig. 5). Pattern data are extracted into the correction area and the boundary area from the original data according to the number of pattern data in the small area. Extracted pattern data are broadcast to the cell processor as pattern data dealt with in the cell processor.

2) The pattern data in the cell processor are broadcast before the message from the cell processor to minimize waiting status in the cell processor arrives.
3) In this parallel algorithm method, the performance of the host program and I/O overhead are the limit of speed-up by parallel processor. As the number of cell processors decreases, the host processor has an increasing number of non-broadcast cell pattern data files, and the host processor enters the waiting status. Also, when the number of cell processors is large, the broadcast time and cell data creation in the host processor increases. Therefore, all processing parts are run in parallel except communication, I/O and small sequential part.
4) The cell processor can handle between 10^2 to 10^4 pattern data. In general, the number of pattern data exceeds the number of cell processors. So, we adopted the static assign method. In this method, the pattern data in the cell processor are first broadcast as two-cell data into all cell processors, then they are broadcast into the cell in which data processing has been completed.
5) Cell data broadcast from the host processor contain data in the small area for correction and data in the boundary area.
However, only the data in the small area for correction are broadcast from the host processor in order to minimize the communication.
6) The dedicated access functions are used to access memory.

Table 1. Application to VLSI pattern results

	SUN4/330	PRANCER				M-780
		64 cell	128 cell	256 cell	512 cell	
Phase 1 (s)	882	26	31	28	28	120
Phase 2 (s)	38 764	695	320	247	226	5 408
Total(s)	39 646	721	351	275	254	5 528
Speed-up ratio	1	54.99	112.95	144.17	156.07	7.17
Efficiency (percent)	100	86	88	56	31	

3. Results of parallel processing

The developed PRANCER system was applied to VLSI patterns to evaluate performance. Table 1 shows the results of this new method when applied to a VLSI pattern. The number of patterns for the target VLSI is 144 million. In Table 1, phase 1 shows the time needed to calculate exposure intensity and its communication. Phase 2 shows the time for pattern data sorting, proximity effect correction, boundary area correction, and communication.

The host computer of AP1000 is a SUN4/330 workstation. VLSI patterns were processed by a mask data conversion program in serial codes on a SUN4/330 for reference. The performance of the SUN4/330 is almost the same as a single processor of the AP1000. The speed-up was 156 times when 512 cells were used. This performance was over 20 times faster than the Fujitsu M-780 conversion system.

Parallel efficiency was 88 percent on 128 cells, which is a satisfactory result. The parallel efficiency was lowered to 31 percent on 512 cells for 144 million patterns. Thus, if a large number of pattern data are assigned to the small areas in Fig. 5, higher parallel efficiency may be obtained on more than 256 cells.

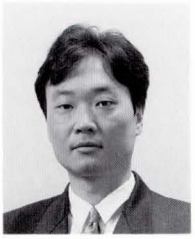
4. Conclusion

A parallel system for mask data conversion, PRANCER, has been developed and implemented on AP1000. Two new algorithms were introduced: one is to balance the load among cells, and the other is to minimize communication overhead.

In PRANCER, all processing parts can be run in parallel except I/O and some small sequential parts. It has been confirmed from a benchmark on a VLSI with 144 million patterns that the CPU performance of PRANCER on AP1000 is over 20 times faster than that of an existing Fujitsu M-780 system. This result suggests that the turn around time (TAT) for mask data conversion may be reduced from the current one week to several ten minutes, if the PRANCER system is applied to 50-100 million gate VLSI patterns.

References

1) Grobman, W. D.: An overview of pattern data preparation for vector scan electron beam lithography. *J. Vac. Sci. Technol.*, **17**, 6, pp. 1156-1162 (1980).
2) Marantz, J. D., and Brank, T.: "Exploiting Parallelism in VLSI CAD". ICCD in Computer Design, New York, 1989, pp. 442-445.
3) Chang, T. H. P.: Proximity effect in electron beam lithography. *J. Vac. Sci. Technol.*, **12**, 6, pp. 1271-1275 (1975).
4) Parikh, M.: Corrections to proximity effect in electron beam lithography. *J. Appl. Phys.*, **50**, 6, pp. 4379-4385 (1979).
5) Sewell, H.: Control of pattern dimensions. *J. Vac. Technol.*, **15**, 3, pp. 927-932 (1978).
6) Machida, Y., Nakayama, N., Furuya, S, and Yamamoto, S. : Enhanced Proximity-Effect Correction for VLSI Patterns in Electron-Beam Lithography. *IEEE Trans. Electron Device*, **ED-32**, 4, pp. 831-835 (1985).

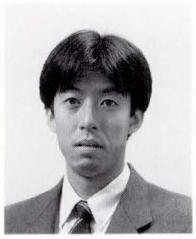


Ryo Tsujimura

LSI-CAD Engineering Dept.
FUJITSU LIMITED
Bachelor of Mechanical Eng.
Chuo University 1985
Master of Mechanical Eng.
Chuo University 1987
Specializing in LSI-CAD System

Kazumasa Morishita

CAD CAM Dept.
FUJITSU VLSI LIMITED
Kochi Higashi Technical High School
1983
Specializing in LSI-CAD System



Yasuo Manabe

LSI-CAD Engineering Dept.
FUJITSU LIMITED
Bachelor of Practical Physical Eng.
Tokai University 1985
Specializing in LSI-CAD System

UDC 530.145:681.322

A Parallelization Study of Quantum Chromodynamics Simulations on the AP1000

● Motoi Okuda ● Akemi Kawazoe ● Masahide Fujisaki

(Manuscript received 1, 1992)

Quantum Chromodynamics (QCD) simulations in the quench approximation have been performed on the AP1000 highly parallel, distributed memory computer. New parallelization techniques have been developed and studied. These techniques have provided a new parallelization algorithm for an update program using a Monte Carlo method, a renormalization program, and a hadron spectroscopy program. The details and physical results are reported here.

1. Introduction

The rapid progress in computer technology during the past decade has made it possible to realistically simulate quantum field theories having infinite degrees of freedom and non-trivial interactions. The lattice formulation by Wilson[1] is an elegant discretization of the models that realize gauge symmetry with high frequency cut-off. Numerical simulation is a powerful tool for studying non-perturbative features of lattice gauge theories. In lattice Quantum Chromodynamics (QCD) in particular, there are many successful projects that use conventional vector supercomputers. There are also several successful, but limited, QCD projects that use new parallel machines[2].

This paper reports on quench QCD calculations that were performed on the AP1000 highly parallel, distributed memory computer. These calculations were performed by the QCD on Thousand cell ARray processor for Omnipurpose research project (QCD－TARO), which is part of a three nation collaboration between Germany, Switzerland, and Japan.

Chapter 2 presents the background of the QCD simulation. Chapter 3 briefly reviews the AP1000 system and the AP1000 features that are especially relevant to the QCD calculation. Chapter 4 describes the parallelism and performance of the QCD program. Chapter 5 briefly discusses the results obtained and their physical implications. The last chapter, Chap. 6, is devoted to concluding remarks.

2. Background of QCD simulation

In the quench approximation, the Feynman path integral for the lattice QCD has the following form[1]:

$$< O > = \frac{\int O[U_l]\, e^{-S} \Pi dU_l}{\int e^{-S} \Pi dU_l} , \qquad \cdots\cdots(1)$$

$$S = \sum_{plaquette} E_{plaq.} , \qquad \cdots\cdots(2)$$

where O is an observable, and U_l is defined on links in the four-dimensional hyper cubic lattice. The action, S, is the sum of $E_{plaq.}$, which is the U_l multiples on the minimal closed loops, i.e. squares on the lattice called plaquettes[3]. To simulate the system, sets of U_l with probability density e^{-S} are successively constructed by using an update procedure, then O is measured.

$E_{plaq.}$ is a multiple of four 3×3 complex matrices, called SU(3) matrices, and requires many operations to calculate. To calculate $E_{plaq.}$, we need U_l values from only the neighboring

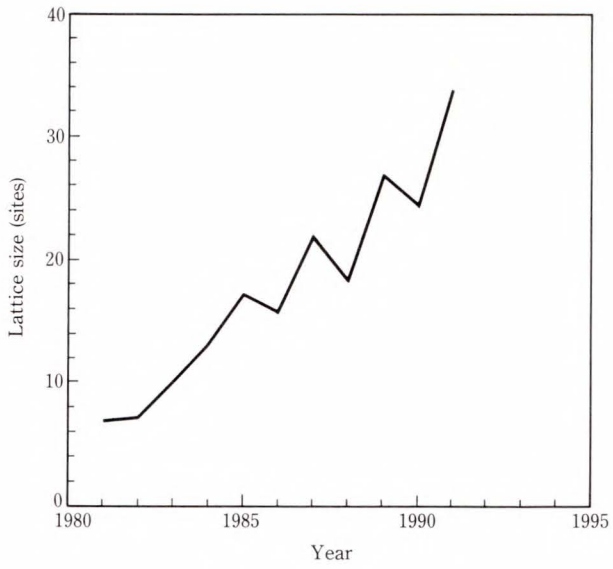


Fig. 1 – Growth of lattice size in QCD simulations.

Growth of average lattice size, L, of QCD simulations (volume is L^4) for quenched simulations. The most recent point shows the size for this QCD – TARO project.

areas, which makes its interdependency low. Therefore, if lattice link variables U_l are distributed between processors judiciously, each processor can work completely in parallel and needs to communicate only with its nearest neighbors.

In the latest QCD studies, a large lattice, i.e. a large memory, was used. Figure 1 shows how the size of the calculated lattice in quench QCD simulations grew from 1980 to 1991[4]. The most recent point in Fig. 1 shows the size for this QCD – TARO project. The memory required for all link variables on a lattice of volume V (i.e. the total number of points on a four dimensional lattice) is $4 \times 18 \times V$ words; for example, our 32^4 lattice requires 75 Mwords. The working space, which is used, for example, for the direction and residual vectors of the conjugate residual (CR) routine, takes up about the same amount of memory as the U_l values. Therefore, a total of hundreds of mega words are required for calculations with a 32^4 lattice.

These characteristics of the QCD calculation suggest that highly parallel computers with distributed memory may give higher performance than conventional vector supercomputers. Many dedicated parallel machine projects have aimed for high performance at relatively low cost and have produced useful results[5]. However, in all of these highly parallel machines, communication between widely separated processors is slow. This means that parallel computers cannot be used efficiently for some QCD calculations. For example, the conjugate gradient/residual algorithm that is now indispensable for the calculation of quark propagators involves the summation of data in all processors, i.e. global summation. Moreover, when calculating complicated physical quantities, long distance communication must not cause a bottleneck. More flexible distribution of the lattice among the processors is possible if fast communications between any two processors can be guaranteed.

The QCD – TARO project was started at the end of 1990. This project is a Monte Carlo renormalization study and a hadron spectroscopy study of large lattices using the AP1000. A Monte Carlo renormalization, which had seemed difficult on a distributed memory parallel computer, was performed by taking advantage of the AP1000's low-latency, high-throughput communication network and its large memory and flexible system configuration. Also, using the AP1000 enabled improvements to be made in the method of QCD updating and hadron spectroscopy in the quench approximation.

3. Implementation of QCD simulation on the AP1000 highly parallel computer

The AP1000 supports Fortran and C languages. In this project, most sections of each program were written in Fortran. A section which generates a task to handle files on disk was written in C. The SU(3) matrix multiplication part was written in Fortran, then its optimized code was tuned by hand using assembler language. All programs in this paper were initially developed on the CASIM AP1000 software simulator.

As discussed in Chap. 2, QCD simulations require a lot of memory. For example, at least 450 Mbytes of memory is required for a $32^3 \times 48$

lattice calculation, making it difficult for a conventional vector supercomputer to calculate. A shortage of main memory causes frequent input-output operations and results in an increase in job execution time. A highly parallel computer with a distributed memory has an advantage here. Each cell processor in the AP1000 has 16 Mbytes of memory and a user programmable memory space of 14.5 Mbytes[6]. In total, 7.4 Gbytes of memory are available to the user in a 512-cell system, which enables up to a $48^3 \times 64$ lattice calculation. In this simulation, a $32^3 \times 48$ lattice calculation is performed on the 512-cell system.

QCD calculations take at least several days and they need a system with high reliability, especially hardware reliability. If the reliability is low, data must be frequently backed up on files, which increases the computer resource requirement. High reliability is one of the most important requirements for highly and massively parallel computers and is also one of the most difficult to achieve. The high reliability of the AP1000 was proved by the fact that there were only two hardware system-downs in six months of the first stage of our project[7,8]. This reliability made it possible to produce useful results after only one and a half years.

In general, the communication schemes and communication speed available decide the algorithm and parallelization strategy for a target program. Because the QCD calculation is performed on a distributed memory parallel processor, data broadcasting and scattering between the host computer and all cell processors is necessary. Also necessary is data exchange between adjacent cell processors and global summations using all cell processors. The amount of data is enormous, for example, 450 Mbytes in the case of a $32^3 \times 48$ lattice. Therefore, data is broadcasted, and distributed and summed between more than 500 processors, which means that high speed communication is essential. Fast and random cell-cell processor communication makes complicated calculations such as renormalization possible on a parallel computer. On the AP1000, the network hardware and routing scheme (a wormhole routing with a

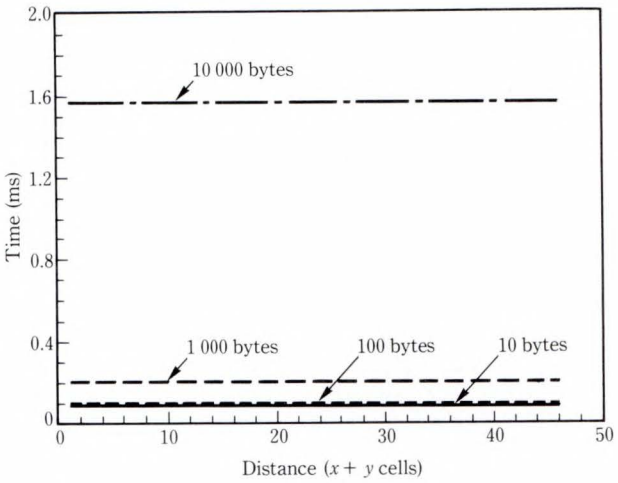


Fig. 2 – Cell communication speed (1).

Communication time as a function of communication distance for data lengths of 10, 100, 1 000, and 10 000 bytes. Data was sent from a cell at (0, 0) to another cell at (x, y). The distance, s, is $x + y$. The timers were in the Fortran program and not in the system communication libraries. Therefore, these times are effective communication times as seen by the application program.

structured buffer pool algorithm) achieves high throughput and low latency communication via a T-net[6].

Figure. 2 shows the results of a test program that measured the time required to transfer data from one cell to a distant cell. This figure clearly shows that the transfer speed is independent of distance. When 10 Kbytes of data are transferred, the transfer speed of the T-net as seen by the application program is 6.3 Mbytes/s.

Figure 3 shows the cell communication speed when all cells simultaneously exchange data with cells at the same distance. This communication pattern imitates random cell-cell communication, and is the kind of pattern found in complicated programs like the renormalization program. These results show the latency due to network congestion. The typical data length in a renormalization calculation is 1 Kbyte, and the effect of congestion is estimated to be negligible from these results.

The nature of the communication scheme makes a renormalization calculation possible on the AP1000 parallel computer. Moreover, to

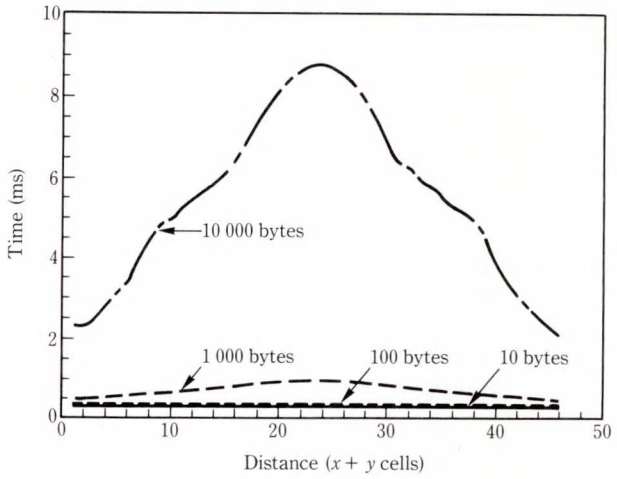


Fig. 3 – Cell communication speed (2).

Communication time as a function of communication distance for data lengths of 10, 100, 1 000, and 10 000 bytes. All cells at (x_i, y_i) simultaneously transmitted data to cells at $(x_i + \Delta x, y_i + \Delta y)$, and simultaneously received data. The distance, s, is $\Delta x + \Delta y$. For example, all cells send data to their west neighbor and receive data from their east neighbor. The distance is 1.

enable short messages to be transferred with little overhead, the AP1000 employs a new concept called line sending, in which messages are sent directly from cache memory to the network[6]. Line sending is very useful when passing link variables between cells. A fast global summation function, whose system library name is C2DSUM, helps speed up the conjugate residual algorithm[8].

Finally, a flexible cell configuration is also necessary. The AP1000 has a user programmable cell configuration interface. The user can construct any two-dimensional cell configuration $N_{cell\,1} \times N_{cell\,2}$ provided the product of $N_{cell\,1}$ and $N_{cell\,2}$ is less than the total number of cells. This means that the lattice size is not tightly constrained by the physical cell configuration.

4. Structure and performance of the QCD simulation program

In this chapter, we describe the parallelization techniques used in our current QCD simulation and describe the simulation's performance.

4.1 Overall structure of the simulation

The fundamental ingredient of lattice gauge theories is link variables, $U_\mu(x)$, where x is a site on a four dimensional lattice and μ (1, 2, 3, 4) represents the direction of the link. A set of link variables is called a configuration. The Monte Carlo steps that are performed to produce a configuration are called sweeps. In the case of lattice QCD, $U_\mu(x)$ corresponds to gluonic degrees of freedom.

The QCD simulation system consists of the Update, Blocking, and Hadron programs. The Update program makes configurations for the Blocking and Hadron programs. The Blocking program is the development program for simulating renormalization. This program blocks lattice sizes from 32^4 to 16^4, 16^4 to 8^4, 8^4 to 4^4, and 4^4 to 2^4, and calculates physical values after the Update program performs 10 sweeps. For hadron spectroscopy, the Hadron program reads the configurations roughly every thousand sweeps to avoid correlations between them, and calculates physical values. The AP1000's large memory allows numerical experiments to be performed using a $32^3 \times 48$ lattice.

4.2 Update program

This section describes the basic equation, parallelization technique, and performance of the update program.

4.2.1 Basic equation

Update uses a Monte Carlo method to produce configurations with the following probability:

$$P[U_\mu(x)] = e^{-S[U_\mu(x)]}/Z, \qquad \cdots\cdots(3)$$

where

$$Z = \int e^{-S[U_\mu(x)]} \prod_x \prod_\mu dU_\mu(x). \qquad \cdots\cdots(4)$$

In lattice gauge theories, the multiple of $U_\mu(x)$ along a minimum square, called a plaquette, is the simplest gauge invariant object. The action, S, is the sum of these terms:

$$S = \beta \sum_{plaquette} E_{plaq.}, \qquad \cdots\cdots(5)$$

where the plaquette energy, $E_{plaq.}$, is written as follows:

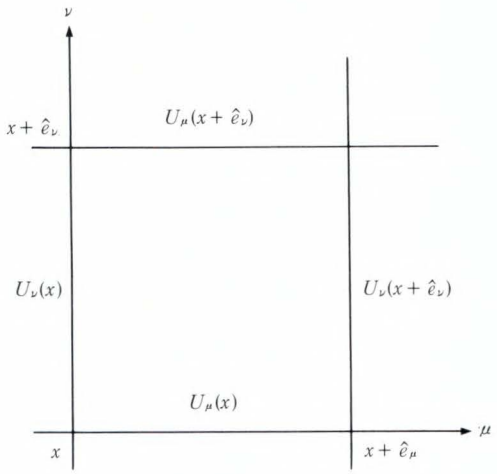


Fig. 4 – Link variables and plaquettes.

Link variables on the lattice which comprise a plaquette. A link variable belongs to twelve different plaquettes {six different planes in four dimensions ($x-y$, $x-z$, $x-t$, $y-z$, $y-t$, $z-t$)}. On each plane, $U_\mu(x)$ is the edge of two plaquettes. Therefore, a link variable couples other link variables in its neighborhood via the plaquettes, but not necessarily using the nearest links. This is an essential feature of gauge theories.

$$E_{plaq.} = \frac{1}{3} TrU_\mu(x)U_\nu(x+\hat{e}_\mu)U_\mu^\dagger(x+\hat{e}_\nu)U_\nu^\dagger(x). \quad (6)$$

Figure 4 shows how $E_{plaq.}$ is constructed on the lattice. Link variables $U_\mu(x)$ are 3×3 SU(3) matrices. The SU(2) subgroup parts are replaced by new ones with probability $P\{U_\mu(x)\}$ using a pseudo heatbath algorithm[9),10)]. Update should be executed for much longer than the auto correlation time, which depends on the update algorithm. Although, for the lattice QCD simulation without matter fields, the pseudo heatbath method is more efficient than others such as Metropolis or Langevin, it slows down drastically near the continuum limit. As a result, the auto correlation time becomes larger and more Monte Carlo steps are required. In order to minimize this effect, an over-relaxation algorithm is used[11)]. This algorithm drastically changes $U_\mu(x)$ {actually, its SU(2) subgroup parts} by maintaining the value of the action, S.

4.2.2 Parallelization

This subsection describes the data mapping method, communication method, and performance of Update.

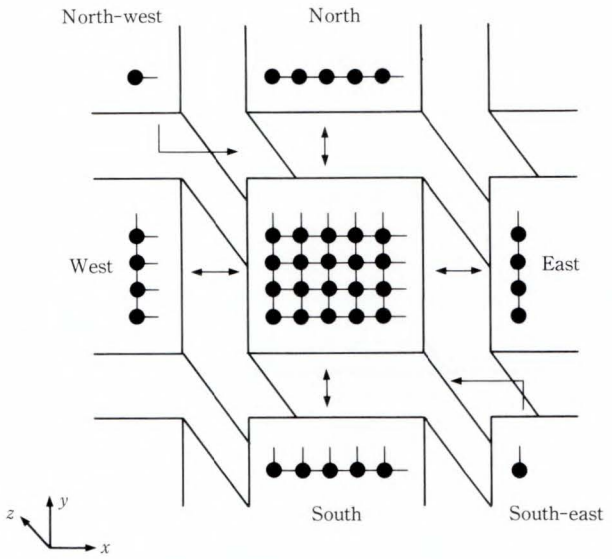


Fig. 5 – Mapping link variables onto cells.

A link variable is updated by referencing only its neighboring link variables, some of which must be fetched from the six neighboring cells. This communication is necessary on distributed memory computers. It is important to adapt the model to the network architecture to make this communication as efficient as possible.

1) Data mapping

For the hadron program, a $32^3 \times 48$ lattice is mapped onto 512 (16×32) cells, so the lattice size of one cell (x, y, z, t) is (2, 1, 32, 48). For the blocking program, a 32^4 lattice is also mapped onto 512 cells. The whole lattice is divided into pieces in the xy-plane, and each piece is mapped onto a cell. This mapping method is called xy-partition.

2) Improved method for communication

A link variable is updated by referencing only the link variables on the six neighboring cells (see Fig. 5). However, this way of defining and referencing variables does not always suit the network architecture of the AP1000. This fetch operation is therefore realized by communication with only the four nearest cells. By arranging the order of communication, the communication time can be improved. Link variables on the north-west and south-east cells are transported by the following two steps:

Table 1. Performance for one update sweep on a 512-cell system

	Pseudo heatbath (s)	Over-relaxation (s)
Calculation	8.94	5.74
Communication	1.61	1.61
(Send-receive, idle)	(0.37)	(0.37)
(Bufferring)	(1.24)	(1.24)
Total time	10.55	7.35
Performance	2.52 μs/link	1.75 μs/link

Time for one sweep of a 32^4 lattice on a 512-cell system for the pseudo heatbath over-relaxation algorithms. The calculation row shows the net calculation time. The communication row shows the communication time. The communication time is the sum of the idling time for sending and receiving data, the buffering time, and the net communication time. (The bufferring time is the time required to copy data to and from the system buffer memory for communication.)

1) Each cell sends data to its north and south cells and receives data from its north and south cells.
2) Each cell sends data to its east cell via the data of its north cell and receives data from its west cell. Sending to the west cell involves the same operation as sending to the east cell.

The communication time in the T-net increases as the number of messages on a communication channel increases. Therefore, the above communication scheme is faster than direct communication between six cells.

4.2.3 Performance

Table 1 shows the performance of one update sweep of a 32^4 lattice on the 512-cell AP1000 system. One sweep takes 10.55 s for the pseudo heatbath algorithm and 7.35 s for the over-relaxation algorithm. From this, the time required to update one link variable is calculated to be 2.52 μs for the pseudo heatbath algorithm and 1.75 μs for the overrelaxation algorithm. The performance is defined as follows:

Performance (μs/link) =
(Total time each sweep)/
(Lattice size times direction). ···(7)

The communication time for one sweep is the same for the pseudo heatbath algorithm and the over-relaxation algorithm because the data length for both is the same. The actual communication time is very short, 0.37 s, which is 3.5 percent (pseudo heatbath) and 5 percent (over-relaxation) of the total time. When the buffering time (time required to copy data in the continuous area for reception into the distributed area of the array) is included, the communication time is 1.61 s. This buffering time is a significant part of the total time and can be reduced by analyzing the system architecture.

Despite this overhead, the pseudo heatbath algorithm compares favorabley with the dedicated QCDPAX machine developed at Tsukuba University[5]. The peak performance of QCDPAX is 14 GFLOPS (10^9 floating operations per second). The peak performance of the 512-cell AP1000 is 3.3 times slower than this, but the one-link update time is only 1.8 times slower. Consequently, the one-link update time per node (cell) on the AP1000 is 1.8 times faster than on the QCDPAX. Also, the AP1000 has a Fortran and C programming environment which reduces the development time and increases program reliability. The AP1000 therefore provides a high performance environment for QCD simulation that rivals a dedicated QCD machine.

4.3 Blocking program

4.3.1 Basic equation

The real space renormalization method integrates the higher frequency parts in the path integral and defines a new action, $S^{(i+1)}$, from the old one, $S^{(i)}$, as follows:

$$e^{-S^{(i+1)}[U_l^{(i+1)}]} = \int \delta[U_l^{(i+1)} - B(U_l^{(i)})] e^{-S^{(i)}[U_l^{(i)}]} \Pi\, dU_l^{(i)}. \quad \cdots\cdots(8)$$

In this formula, $U^{(i+1)}$ is derived from $U^{(i)}$, i.e. $U^{(i+1)} = B(U^{(i)})$, and $U^{(i)}$ is integrated. B is the blocking operation and is defined as the following two steps:

1) X is the sum of products along a set of paths connecting two block sites:

Table 2. Mapping for blocking in two dimensions

Lattice size	32^4	16^4	8^4
Cells used	512 (16×32)	256 (16×16)	64 (8×8)
Lattice size/cell	$2 \times 1 \times 32 \times 32$	$1 \times 1 \times 16 \times 16$	$1 \times 1 \times 16 \times 16$

When the lattice is smaller than 32^4, efficiency is very low because some of the 512 cell processors are not working.

$$X = U_\mu^{(i)}(x)U_\mu^{(i)}(x+\mu) + \frac{1}{2}\sum_{\nu\neq\mu} U_\nu^{(i)}(x)U_\mu^{(i)}(x+\nu)U_\mu^{(i)}(x+\mu+\nu) U_\nu^{(i)\dagger}(x+2\nu), \quad \cdots\cdots(9)$$

2) x is then mapped onto an SU(3) element, $U_\nu^{(i+1)}(x)$.

Blocking starts from a 32^4 lattice. After each blocking step, the lattice size is reduced by a facter of 2 (i.e. from 32^4 to 16^4 to 8^4 to 4^4 to 2^4).

4.3.2. Parallelization

This subsection discusses the mapping of a lattice onto cells. At first, it seemed impossible to parallelize the blocking. If blocking uses the xy-partition employed in the update program, the blocked lattices (16^4, 8^4, 4^4, and 2^4) cannot be mapped onto the cells (see Table 2). This is why blocking was previously done only on vector machines and parallel machines were only used for updating. Blocking does however use enormous amounts of memory and CPU time, so only small scale studies have been done on vector machines. In the QCD－TARO project, a parallel machine is being used to successfully handle the large lattice size of 32^4.

Updating and blocking for a renormalization study using the AP1000 was achieved by ensuring that the programs fully exploit the AP1000's flexibility. Two types of blocking, 32^4 to 8^4 and 8^4 to 2^4, were considered.

Figure 6 shows the general-flow of blocking. The first type of blocking reduces the lattice size from 32^4 to 8^4. First, all cells are divided into eight groups of 64 cells (8×8); each group is called a super cell (see Fig. 7). The whole lattice is divided, perpendicular to the z-axis, into eight pieces, each of which is mapped onto a super cell. These pieces are then partitioned in the xy -plane, and each partition is mapped onto cells in the super cell. This method is called xyz -partition. As discussed in Chap. 2, the T-net's communication performance is only slightly influenced by distance.

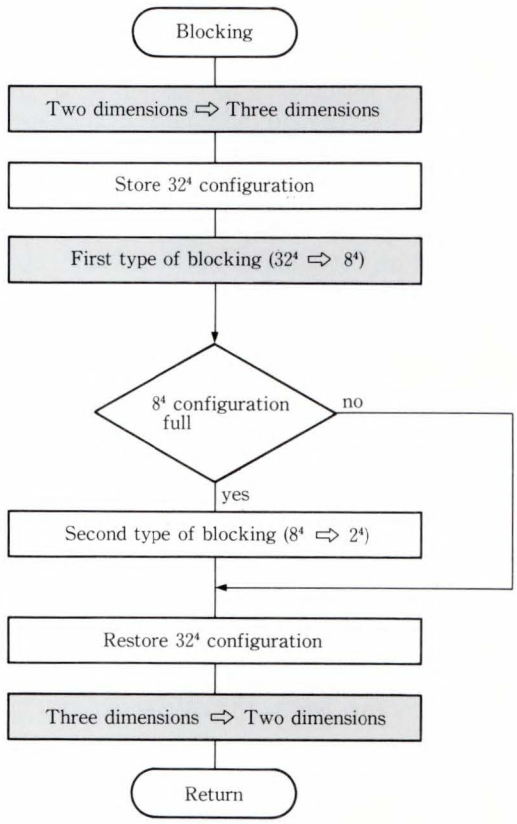


Fig. 6－General flow of blocking.

Each cell has a piece of the 32^4 configuration. Forty-eight 8^4 configurations are produced by a blocking procedure from 32^4 to 8^4. After the first type of blocking has been repeated ten times (from 32^4 to 8^4), the second type of blocking (from 8^4 to 2^4) starts on 480 processors.

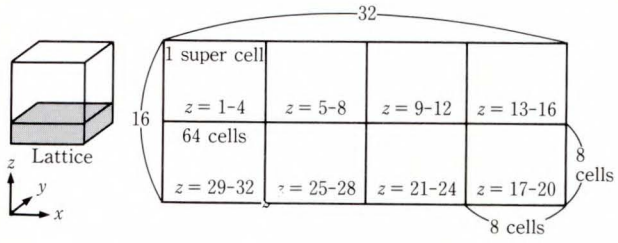


Fig. 7 – Mapping of lattice onto super cells.

All cells are divided into eight super cells. The whole lattice is sliced perpendicular to the z-axis into eight pieces, and each piece is mapped ontoa super cell. Each piece is cut in the super cell in the xy-plane.

This super cell arrangement does not work when the lattice is smaller than 8^4. After blocking from 32^4 to 8^4, which is scattered over all 512 cells, the 8^4 configuration is compressed into one cell. The original 32^4 configuration becomes 48 configurations of 8^4. The blocking process from the 32^4 to 8^4 lattice is repeated ten times, giving 480 cells with an 8^4 configuration.

The second type of blocking starts to block from 8^4 to 2^4 on each cell. Because the AP1000 has about 14.5 Mbytes of user memory for each cell, the 32^4 configuration data can be kept in memory to continue the update program. When blocking is complete, the 32^4 configuration data is restored, mapping is changed from three-dimensional mapping (xyz-partition) back to two-dimensional mapping, and updating starts again. In this way, full blocking from 32^4 to 2^4 can be performed. Communication between cells in three-dimensional mapping is achieved using a subroutine called XGATHER. Each cell constructs a look-up table of all indices of link variables, and uses it to find a partner for communication.

4.3.3 Performance

Table 3 shows the performance of the Blocking program for one sweep. The first type of blocking, from 32^4 to 8^4, which calculates one configuration for the 512 cells, takes 68.5 s. The second type of blocking, from 8^4 to 2^4, which processes 480 configurations using 480 cells, takes 147.9 s. The performance of the communication subroutine, XGATHER, was measured during the first type of blocking. In the second type of blocking, XGATHER is not called since each cell calculates only one configuration. In one sweep, XGATHER is called 8 848 times and runs for a total of 44.3 s (buffering time: 17.97 s, synchronization and idle time: 26.28 s, and communication time: 0.05 s). The synchronization and idle times are very large because cells send the results to the host before XGATHER is called, which delays the start of the XGATHER subroutine.

Table 3. Performance of blocking program for one sweep

Lattice size	$32^4 \rightarrow 16^4 \rightarrow 8^4$	$8^4 \rightarrow 2^4$
Total time (s)	68.5	147.9
XGATHER		
(8 848 repititions)	44.3	
(Buffering)	(17.97)	
(Synchronization and idle)	(26.28)	
(Communication)	(0.05)	

The first type of blocking (from 32^4 to 8^4) calculates one configuration for each lattice size using 512 cell processors. The second type of blocking (from 8^4 to 2^4) processes 480 configurations for each lattice size using 480 cell processors. The communication routine, XGATHER, is only called by the first type of blocking.

4.4 Hadron program

The matter fields in the QCD system are degrees of freedom of quarks. The quarks live on sites and have 12 components (three color components × four Dirac components) on each site. The propagator of hadrons is a two point correlation function between hadronic operators composed of quark fields and quantum numbers corresponding to the hadrons. For example, the meson propagator has the following form:

$$m(z) = \Psi^{\dagger}_{\mu,a}(z)\Gamma\Psi_{\mu,a}(z), \qquad \cdots\cdots(10)$$

where

$$\Gamma = 1, \gamma_\mu, \gamma_\mu\gamma_\nu, \gamma_5, \gamma_\mu\gamma_5. \qquad \cdots\cdots(11)$$

The values of Γ are scalar, vector, tensor, axial vector, and pseudo scalar, respectively. The propagator can be expressed in the path integral representation as follows:

$$<h(x)h(y)> = \int h(x)h(y)e^{-S[U_\mu(z)]-S_{matter}[\Psi^\dagger_{\mu,a}(z),\Psi_{\mu,a}(z)]} \times \prod_z \prod_\mu dU_\mu(z) \prod_a d\Psi^\dagger_{\mu,a}(z) d\Psi_{\mu,a}(z), \quad \cdots\cdots(12)$$

where S_{matter} is a Euclidean action of quark fields,

$$S_{matter} = \sum_{\mu,\nu,a,b} \Psi^\dagger_{\mu,a}(x) W_{\mu\nu,ab}(x,y) \Psi_{\nu,b}(y). \quad \cdots(13)$$

The Wilson fermion is taken as follows:

$$W(i,i) = I, \quad \cdots\cdots(14)$$

$$W(i,j) = -\kappa(1-\gamma_\mu)U(i,j) \qquad (j = i+\mu), \quad \cdots\cdots(15)$$

$$W(i,j) = -\kappa(1+\gamma_\mu)U^\dagger(j,i) \qquad (j = i-\mu), \quad \cdots\cdots(16)$$

$$\text{others} = 0. \quad \cdots\cdots(17)$$

The path integration of the fermionic part can be performed analytically. For example, a meson propagator can be written as follows:

$$<h(x)h(y)> = Tr<W^{-1}(y,x)\Gamma W^{-1}(x,y)\Gamma detW>. \quad \cdots\cdots(18)$$

From here on, $detW$ is set to unity, i.e. it becomes a quenched approximation. The same type of expression can be obtained for baryons. The sum over all spatial sites is taken to extract zero spatial momenta and measure the propagator along the Euclidean temporal direction as follows:

$$<h(t)h(t')> = \sum_{\boldsymbol{y}} <h(t,\boldsymbol{x})h(t',\boldsymbol{y})>. \quad \cdots\cdots(19)$$

From these expressions, it can be said that the propagators of hadrons are constructed by a set of rows of the inverse matrix of the Wilson fermion corresponding to the origin, $\boldsymbol{x}$. The propagators of hadrons are obtained by solving $W\boldsymbol{x} = b$ 12 times with b's of unity in one column and zeros for all other elements. The Wilson fermion is a function of link variables produced by the Update program described earlier. The hadron propagators are measured on each configuration to provide a statistical analysis. The hadron program consists of two parts. One part solves the linear equation and the other calculates hadron propagators from the solutions. The most time consuming part is the conjugate residual (CR) routine used when solving the linear equation. Because the QCD contains only local interactions, the Wilson fermion has only eight non-zero off-diagonal elements in each column from a kinetic term. The matrix is however very large, i.e. (number of lattice points)×(number of Dirac components)×(number of color components), and obtaining the product of this large, sparse matrix and a vector comprises the main part of the calculation. Moreover in the CR method, we must frequently calculate $\boldsymbol{y} = W \cdot \boldsymbol{x}$ type matrix vector multiplications.

4.4.1 Parallelization

From the concrete form of the Wilson fermion matrix, it can be seen that the product of the matrix and a vector $\boldsymbol{x}$ can be divided into eight parts. y_i (y at site i) is constructed in the CR method from the sum of x_i and $W_{i,i\pm\mu} \cdot x_{i\pm\mu}$ where $W_{i,i\pm\mu}$ is a 12 × 12 block matrix that connects sites i and $i \pm \mu$, and $\mu = x, y, z$ and τ. This calculation process can be implemented on the AP1000 in the following two steps:

i) $y_{i\pm\mu}$ are constructed from the sum of $x_{i\pm\mu}$ and $W_{i\pm\mu,i} \cdot x_i$.

ii) For calculating the boundary data y_i on each cell, the data of $x_{i\pm\mu}$ and $W_{i\pm\mu,i} \cdot x_i$ are transfered from the nearest cells using T-net.

In this procedure, to reduce data communications and the required amount of memory, the products $W_{i\pm\mu,i} \cdot x_i$ are calculated on the nearest cells before communications. This method is generally applicable to matrices of the same type. The conjugate residual routine needs global sums to calculate the inner products of vectors whose components are scattered among the cells. An efficient system library routine, C2DSUM, calculates the total sum over all cells. The solver part of this routine is executed only on cells and does not perform communications between cells and host.

Figure 8 shows the flow of the conjugate residual method in the Hadron program.

4.4.2 Performance

The kernel of the product of the Wilson

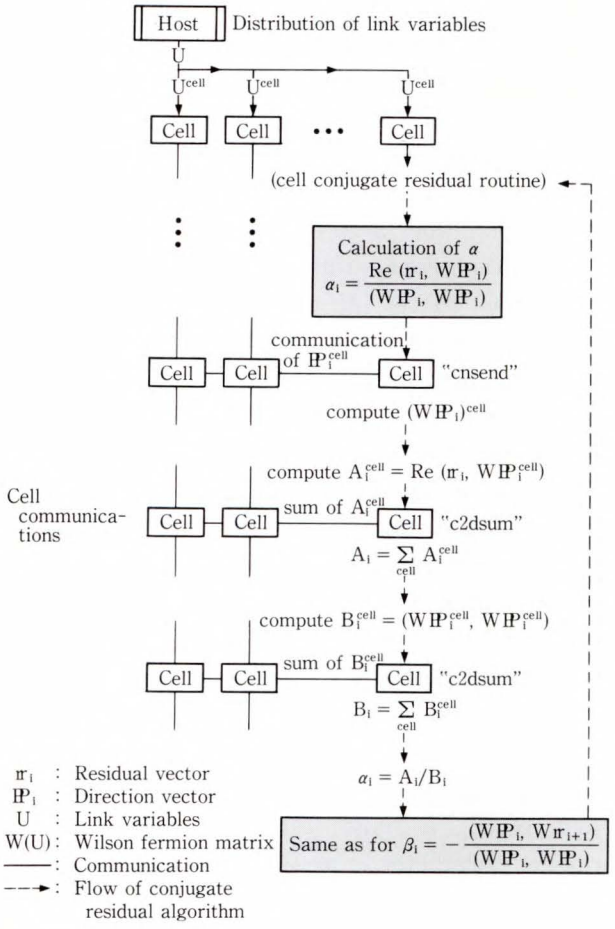


Fig. 8 – Flow of conjugate residual method in Hadron program.

Only C2DSUM and CNSEND are necessary to transfer data when the quark propagators are constructed using the conjugate residual algorithm. C2DSUM is a fast routine which sums a variable on each cell and distributes the result to all cells. This routine avoids any cell-host communication in the solver. CNSEND handles the communication with the nearest cell.

Table 4. Performance for one iteration of conjugate residual (CR) method

Process	Time (s)
Matrix solver	19.4
Sum over all cells	0.012
Others	4.5
Total time	23.912

The simulation is for a $32^3 \times 48$ lattice on 512 processors. The sum for all cells is done by the C2DSUM routine, which is called 24 times per iteration.

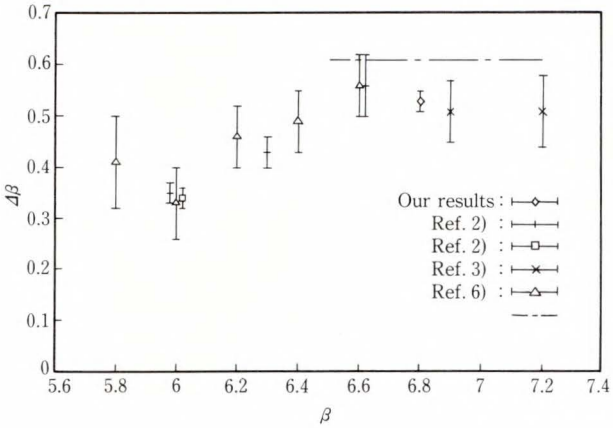


Fig. 9 – $\Delta\beta$ as a function of β.

The lattice distance is halved when the coupling parameter changes from $\beta - \Delta\beta$ to β. The dotted line is a prediction by the continuum theory. Our Monte Carlo data are plotted together with the old data. A point at $\beta = 6.8$ is calculated on a 32^4 configuration. All other points are calculated on a 16^4 configuration. A new point at $\beta = 6.8$ is stable and indicates a slow approach to the continuum limit.

fermion matrix and a vector is coded in assembler. Using the CR method, the speed of the solver is 1.78 μs/site for each iteration. Table 4 breaks down the calculation time for one CR iteration for a $32^3 \times 48$ lattice configuration of 512 cells. The C2DSUM subroutine is called 24 times for one iteration and uses only 12 ms of the total time. C2DSUM is very efficient, and the global sum takes only a small proportion of the CR calculation time.

5. Physical results

5.1 Monte Carlo study of renormalization group

The lattice simulation attempts to approximate the continuum theory. The best way to see whether a lattice theory is reaching the correct continuum limit is to perform a real space renormalization group study. Repeating the renormalization group transformation changes the action, S, as follows:

$$S = S^{(1)} \rightarrow S^{(2)} \rightarrow S^{(3)} \rightarrow, \quad \cdots\cdots(20)$$

and one expects to reach the renormalization trajectory originating from the correct fixed point corresponding to continuous QCD.

The real space renormalization group transformation changes the cut-off from $\Lambda = \pi/a$ to π/ba, where $b = 2$. The corresponding change in the coupling, g, is governed by the β-function,

$$\beta_{func} = \Lambda dg/d\Lambda = -adg/da. \quad \cdots\cdots(21)$$

In the two loop calculation, the β function is given as:

$$\beta_{func} = C_1 g^3 + C_2 g^5. \quad \cdots\cdots(22)$$

From these equations, one can calculate the change in $\beta = 6/g^2$ after one renormalization group transformation. The two-loop prediction of $\Delta\beta$ is shown in Fig. 9 as a dotted line.

To evaluate $\Delta\beta$ in the numerical simulation, six operators at β on a 32 lattice and at β' on a 16^4 lattice at each blocking level are calculated. The expectation value at level l as calculated using this method is merely the expectation value in Equation (1) with $S^{(l)}$ instead of S. Therefore, if we are near the renormalized trajectory and $\beta' = \beta - \Delta\beta$, all expectation values at level l on a 16^4 lattice match those at level $l + 1$ on a 32^4 lattice. To determine $\Delta\beta$, blocking at $\beta = 6.8$ and 7.0 on the 32^4 lattice and at many values of β' on the 16^4 lattice are performed. Figure 9 shows our results and those of the CERN-DESY-Edinburgh collaboration[12),13)], which were obtained on 16^4 and 8^4 lattices. Note that above $\beta = 6.5$, the configurations generated on 16^4 are in the deconfinement phase.

5.2 Hadron spectroscopy

Meson and baryon spectroscopy with Wilson fermions in the quenched approximation were studied. In this equation, unnecessary quark doublers have higher masses (of the order of $1/a$) and hadron wave functions can be constructed from point like Dirac fermions in a normal way. The Wilson fermion action has a troublesome chiral breaking term that is proportional to the higher order of a. However, this term is expected to become negligible at the continuum limit, and the pion is expected to behave as a proper Goldstone boson corresponding to the chiral symmetry. It is therefore quite important to study hadron spectroscopy at smaller lattice distances, a. This means, as stressed in the introduction, that large lattices must be handled.

Hadron propagators along the τ-direction, which has more lattice points than spatial directions, were measured. All other spatial directions are integrated, which sets a zero spatial momentum, and point like operators were chosen as sources. Boundary conditions are periodic in all directions. With just a single pole in the spectral function, the propagator behaves like cosh ($m\tau$). If the lattice is large enough, and excited states are well separated, this type of behavior like cosh ($m\tau$) is seen. Using the 21 most distant points, an χ^2-fit of the cosh type is performed, and the data fits well with that of a single mass.

Although $\kappa = 0.140$ is far from the critical value of κ, a rough estimate of the critical hopping parameter is around 0.151. The ρ mass in the chiral limit gives a lattice spacing of about $a = 0.071$ fm, so our lattice covers 2.3 fm in the spatial direction and 3.4 fm in the temporal direction.

The plot of m_p/m_ρ versus m_π/m_ρ is often used to present results. Figure 10 shows a curve estimated by Ono[14)]. Clearly m_p/m_ρ still seems large and $\kappa = 0.150$ is not close enough to the chiral limit. The values show good agreement with the curve within a single pole fit.

6. Conclusion

This paper reported on a parallelization study of QCD simulation that was done as part of the QCD -TARO project. This study was done using the AP1000 highly parallel, distributed memory computer.

A new algorithm and improved method for a Monte Carlo renormalization study, and hadron spectroscopy in the quench approximation were discussed. The AP1000's low-latency, high -throughput network and flexible system configuration were fully exploited. The quench QCD calculation with the largest lattice size was

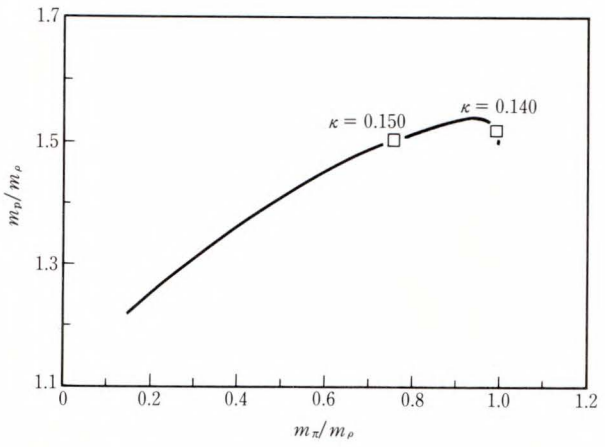


Fig.10 – Edinburgh plot for hadron masses.

(Nucleon mass/ ρ meson mass) versus (π mass/ ρ mass) (Edinburgh plot) with an empirically estimated curve by Ono. The ρ meson – π meson mass ratio is still higher than the experimental value. The data agrees well with the curve, which suggests that a non-relativistic quark model works well.

performed by using the large main memory of the AP1000. The combination of our new algorithm and a super cell technique makes it possible to perform a renormalization study on a highly parallel computer.

The highly parallel computer has proved to be a viable alternative to the conventional vector supercomputer, despite the complicated programs that are usually required to investigate interesting observables. Improvements in the update and hadron spectroscopy programs, such as our data mapping strategy, enable us to perform one of the fastest and largest calculations in this field on a highly parallel computer. Also, because of the excellent program development support environment and high reliability of the AP1000, the first stage of the QCD-TARO project has already produced useful results with a lattice close to the continuum limit.

7. Acknowledgement

The authors would like to thank the members of the QCD – TARO project, especially Assoc. Prof. A. Nakamura, Assoc. Prof. O. Miyamura, Dr. S. Hioki, Mr. T. Takaishi, and Dr. T. Hashimoto, for their valuable advice and technical assistance.

Reference

1) Wilson, K.: Confinement of Quarks. *Phys. Rev.*, **D10**, pp.2445-2459 (1974).
2) Christ, N. H.: QCD machines-present and future. *Nucl. Phys.*, **B20**, pp.129-137 (1991).
3) Nakamura, A., Feuer, G., Hege, H. C., and Linke, V.: Fast Algorithm for an exact Treatment of Lattice Quantum Chromodynamics by Monte Carlo Simulation on Vector Processors. *Comput. Phys.*, **51** pp.301-315 (1988).
4) DeGrand, T. A.: Lattice QCD Spectroscopy Progress and Prospects. *Nucl. Phys.*, **B20**, pp.353-361 (1991).
5) Iwasaki, Y., Kanaya, K., Yoshie, T., Hoshino, T., Shirakawa, T., Oyanagi, Y.,Ichii, S., and Kawai, T.: Deconfining transition of SU(3) gauge theory on $N_t = 4$ and 6 lattices. *Phys. Rev. Lett.*, **67**, 24 pp.3343-3346 (1991).
6) Ishihata, H., Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Processing., pp.13-16 (1991).
7) Akemi, K., deForcrand, Ph., Fujisaki, M., Hashimoto, T., Hege, H. C., Hioki, S., Makino, J., Miyamura, O., Nakamura, A., Okuda, M., Stamatescu, I. O., Tago, Y., and Takaishi, T.: SU(3) Renormalization group study on parallel computer AP1000. *Nucl. Phys.*, **B26**, pp.420-422 (1992).
8) Akemi, K., deForcrand, Ph., Fujisaki, M., Hashimoto, T., Hege, H. C., Hioki, S., Makino, J., Miyamura, O., Nakamura, A., Okuda, M., Stamatescu, I. O., Tago, Y., and Takaishi, T.: Hadronic spectroscopy on a $32^3 \times 48$ lattice. *Nucl. Phys.*, **B26**, pp.293-295 (1992).
9) Cabibbo, N., and Marinari, E.: A new method for updation SU(N) matrices in computer simulations of gauge theories. *Phys. Lett.*, **119B** pp.387-390 (1982).
10) Kennedy, A. D., and Pendleton B. J.: Improved heatbath method for Monte Carlo calculations in lattice gauge theories. *Phys.*

Rev. Lett., **156B**, pp. 393-399 (1985).

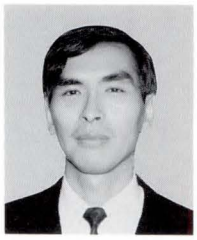
11) Brown, F. R. and Woch, T. J.: Overrelaxed heatbath and metropolis algorithms for accelerating pure gauge Monte Carlo calculations. *Phys. Rev. Lett.*, **58** pp.2394-2396 (1987).

12) Bowler, K. C., Hasenfratz, A., Hasenfatz, P., Heller, U., Karsch., F., Kenway, R. D., Meyer-Ortmanns, H., Montvay, I., and Pawley, G. S.: Monte Carlo renormalization group studies of SU(3) lattice gauge theory. *Nucl. Phys.*, **B257**[FS14] pp.155-172 (1985).

13) Bowler, K. C., Hasenfratz, A., Hasenfratz, P., Heller, U., Karsch, F., Kenway, R. D., Pawley, G. S.,and Wallace, D. J.: The SU(3) beta-function at large beta. *Phys. Lett.*, **179B**, pp375-378 (1986).

14) Ono, S.: States of hadrons in a five-quark model. *Phys. Rev.*, **D17**, pp.888-891 (1978).

15) Bernard, C., DeGrand A., T., DeTar, C., Gottlieb, S., Krasnitz, A., Ogilvie, M. C., Sugar, R. L., and Toussant, D.: "QCD on the iPSC/860". *Fermion Algorithm*, Herrman, H. J. and Karsch, F. ed., World Scientific, 1991, pp27-42.



Motoi Okuda

Computer Science Research Laboratory
System Engineering Group
FUJITSU LIMITED
Bachelor of Nuclear Eng.
Nagoya University 1974
Master of Nuclear Eng.
Nagoya University 1976
Specializing in Scientific Computing and Parallel Computing

Akemi Kawazoe

Computer Science Research Laboratory
System Engineering Group
FUJITSU LIMITED
Bachelor of Physics Eng.
Science University of Okayama 1984
Specializing in Scientific Computing and Parallel Computing

Masahide Fujisaki

Computer Science Research Laboratory
System Engineering Group
FUJITSU LIMITED
Bachelor of Science
Science University of Tokyo 1984
Specializing in Scientific Computing and Parallel Computing

UDC 539.12:681.322

Connecting the AP1000 with a Mainframe for Computations of the Experimental High Energy Physics

● Sin-ichi Ichikawa ● Atsushi Manabe ● Toshihiko Matsuura

(Manuscript received August 31, 1992)

A scalable, high-bandwidth connection mechanism between FUJITSU's AP1000 highly parallel computer and the M-series mainframe has been designed to provide the AP1000 with an input/output system that matches its computing capability.
A prototype input/output system using Fiber Distributed Data Interface (FDDI) network with a TCP/IP interface, and a dedicated hardware interface board which interfaces the local bus of the processor nodes with the VME bus have been developed.
A software system for this connection has also been developed to facilitate porting the input/output procedures of the mainframe-originated application programs to the processor-nodes. This system greatly reduces the parallelization programming workload.

1. Introduction

Recently, many attempts have successfully proven that the microprocessor-based highly parallel processors are cost-effective for providing high and scalable CPU power to the parallel computations of various areas. However, the effective performance of parallel processors depends much on the degree of parallelism, granularity, load-balancing, and the way of using the interprocessor network by each application program. In addition, to extract the effective computational scalability from highly parallel processors, a well-balanced input/output bandwidth has to be attached to highly parallel processors so as to meet their high CPU performance.

A highly parallel processor without sufficient input/output bandwidth will show effectively low CPU performance or poor scalability to the number of processors even for the highly parallelizable applications such as the high energy physics experimental data analysis, the large-scale gene analysis, and signal processings of the remote sensing.

The background of this work is in the application of parallel processors to the high energy physics experimental data analysis. Computations for current high energy physics experiments have already been heavily CPU demanding. For example, the experiments of TRISTAN accelerator held at the National Laboratory for High Energy Physics requires more than 100 MIPS of computing power for the offline data analysis. Furthermore it is planned to boost the accelerator experiment by transferring to higher beam intensity and energy as shown in the Appendix. As the necessary amount of computations will explode drastically far beyond the present level, it is obvious that only the parallel processing approach has a possibility to provide the required amount of computer power for the coming experiments.

According to a preliminary study, experimental data analysis programs used heavily in the high energy physics computation can be perfectly parallelized thanks to the independent nature of each particle collision event (the event parallelism)[1]. However, the application requires

large amount of data-access to external file systems during the course of event-parallel processing. This will result in the I/O-bounded processing of highly parallel processors.

The AP1000 is a highly parallel processor system developed by Fujitsu Laboratories Ltd.. Because this system has only a single host processor, i.e., the SUN4 workstation which has to handle all the input/output requests from the cells, the total I/O bandwidth is determined by that of the host processor. For example, given the average size of experimental data for one event and the average computational time for the analysis of it, one experimental data analysis program requires approximately 530 Kbytes/s and 2.2 Mbytes/s of the effective I/O throughput so as to balance the CPU capability of the AP1000 with the configuration of 128 nodes and 512 nodes, respectively. Therefore, because input-output bandwidth of the host processor is about 500 Kbytes/s, no more performance increase can be expected in this particular case beyond a configuration of approximately 120 nodes[1]. Relatively five times heavier I/O bandwidth requirement to this has been observed in another experimental data analysis program.

The amount of the experimental data involved in high energy physics computations is enormous, and frequent access to this data by CPU demanding jobs and various other analyses by using an archive facility must be possible. Therefore, experimental data analysis requires large capacity DASDs and high speed channels such as are found in a mainframe. It seems that the only choice for the major component of an AP1000 external data-file system for experimental data analysis computations is a mainframe. Therefore, the AP1000/mainframe connection will be a key issue in the practical application of the AP1000 in experimental data analysis computations.

This paper describes a study on a scalable, high-bandwidth connection between the AP1000 and the Fujitsu M-series mainframe for high energy physics computations. Chapter 2 discusses the connection in terms of the input/output bandwidth and exploitation of hardware resources. Chapter 2 also describes in detail the implementation of the prototype hardware. Chapter 3 describes the programming issues regarding reuse of mainframe-originated experimental data analysis programs. Chapter 3 also describes the design of the data transfer software and its implementation. Chapter 4 discusses the effectiveness of the connection. Finally chapter 5 looks at the future plans for work in this area.

2. Hardware design of the mainframe/AP1000 connection

2.1 AP1000 architecture

The AP1000 highly parallel computer is a multiple-instruction multiple-data (MIMD) computer with distributed memories. A node processor of AP1000, called the cell, consists of a SPARC microprocessor and a specially designed communication interface. Program loading, input/output processing, and control of cell program is done using a workstation called the host. The AP1000 has three communication networks: the synchronization network (S-net), which is used for inter-cell and host-cell synchronization, the broadcast network (B-net), which is used for host-cell and inter-cell broadcasting, and a two-dimensional torus network (T-net), which is used for inter-cell communication[2),3)].

Each network uses a different structure and mechanism to enable efficient data transfer. Communication on the T-net has a low latency and is virtually deadlock free due to a unique message routing technique which consists of the wormhole routing and the structured buffer pool algorithm[2),3)]. Because of this technique, the communication throughput between a pair of distant cells is not so different from that between an adjoining cell-pair. The peak communication throughput of the hardware is 25 Mbytes/s.

The B-net is an exclusive-access bus which allows only one sender to start communication at a time. The peak hardware throughput for inter-cell communication is 50 Mbytes/s. However, the host-cell data transfer rate is restricted to slightly below 2 Mbytes/s because of the VME bus data transfer speed of the host

workstation.

The programming model of the AP1000 consists of message passing and the synchronization. The communication library routines for inter-cell and intra-cell communication, host-cell communication, synchronization, and initialization can be used in C or Fortran programs.

2.2 Characteristics of mainframe input/output

The mainframe has high-speed channels for external input/output connections. By using multiple channels, the total input and output throughput currently reaches 1 Gbytes/s. The mainframe channel supports a variety of large-scale, high-speed, external storage systems, for example, tape-robot systems, array disks, and large-storage optical disks. These features make the mainframe a suitable data-file storage for experimental data analysis computations.

2.3 Candidates for hardware configuration of AP1000/mainframe connection

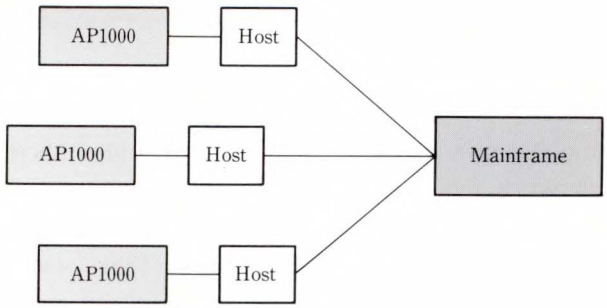
The purpose of the AP1000/mainframe connection is to provide the AP1000 with a scalable high bandwidth and a large-capacity data-file storage by using a mainframe as an external file system. The practical approach for this purpose is to make the connection between the AP1000 and a mainframe multiple.

The candidates for the connection configuration with respect to the AP1000 network organization are as follows:

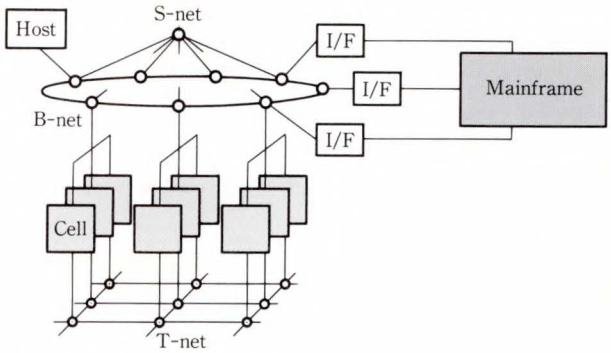
1) Multiple full sets of small-scale AP1000 systems connected to the mainframe via hosts {see Fig. 1a)},
2) a single AP1000 system multiply connected to the mainframe via B-net and its network interface {see Fig. 1b)}, and
3) a single AP1000 system whose multiple cells are connected directly to the mainframe {see Fig. 1c)}.

In the second and the last configuration, it is appropriate for each data path to supply data to one cell-cluster which contains certain number of cells.

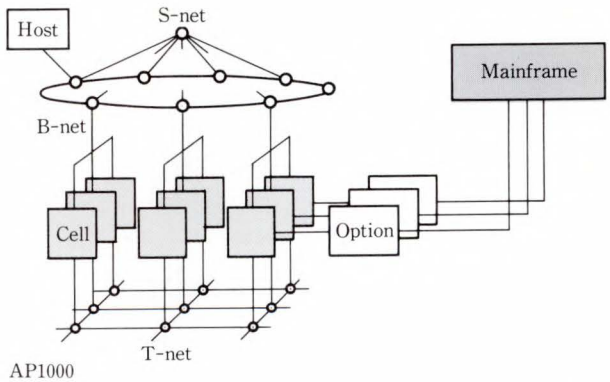
The first connection configuration relieves communication traffic between the cells and mainframe. However, this configuration is the least flexible because virtually no cooperation is possible among individual AP1000s with their hosts. This feature might be disadvantageous in certain computations of the experimental data analysis, for example, statistical analysis over all events. Also, there is a hardware redundancy



a) Multiple sets of AP1000 connected via their hosts



b) B-net connected to mainframe via multiple interfaces



c) Cells connected to mainframe via multiple options

Fig. 1 – AP1000/mainframe connection configurations.

if multiple full-sets of AP1000 systems are to be used to run a single job.

The second configuration relieves only the traffic between the network interface and the mainframe. This configuration will complicate network control of access to the B-net by multiple cells and multiple network interfaces. As in the first configuration, the second configuration does not use the T-net to reduce input/output communication traffic.

The third configuration can disperse input/output communication traffic by using the T-net and by connecting multiple cells to the mainframe. With this configuration, communication without inter-cluster interference is possible because each cluster has its own mainframe connection. Therefore, the third configuration enables an input/output system whose data transfer throughput is scalable with respect to the number of cells.

It is therefore decided to use the third configuration.

2.4 Design of connection hardware

1) Prototype configuration of cell I/O system

The communication network chosen for the prototype AP1000/mainframe connection is an

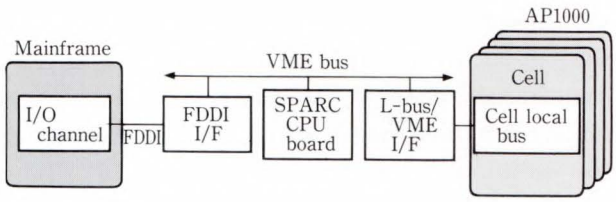


Fig. 2 – Prototype hardware configuration.

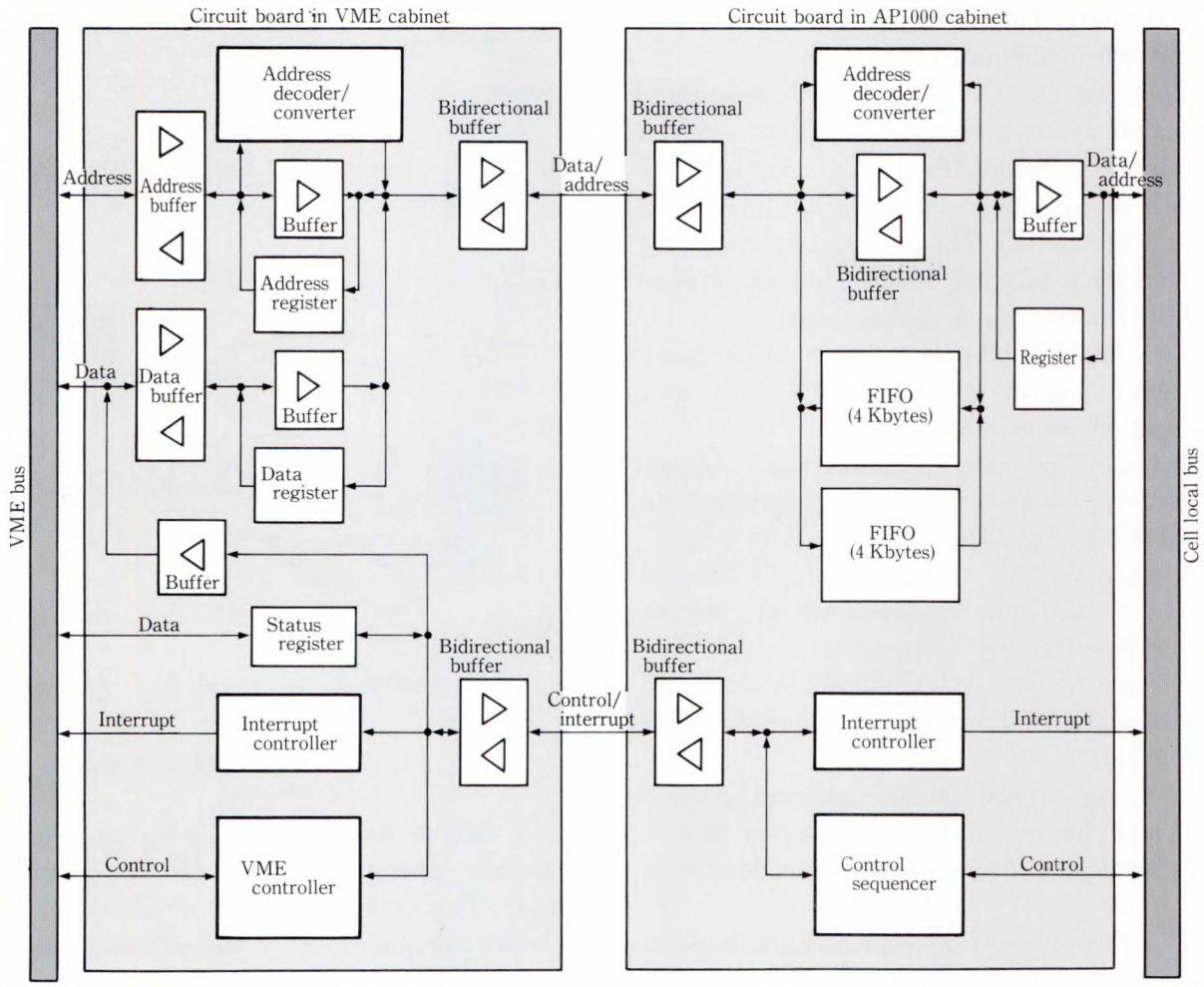


Fig. 3 – L-bus/VME interface circuit configuration.

optical fiber link network (FDDI: Fiber Distributed Data Interface of the American National Standards Institute) which provides a comparatively high communication throughput and is easily connected to a mainframe. The hardware functions of the AP1000 can be extended by adding optional hardware to cell's local bus (L-bus). This extendable feature is used to connect the FDDI interface to the cells.

The cell input/output system has to provide the following two functions. One function distributes and gathers data to and from multiple cells for the input/output requests of an application program. The other function communicates with mainframe, which performs the actual data-file input/output operations.

To achieve high-performance input/output system, communication with the mainframe should be done simultaneously with the processing for the distribution and gathering of data to and from the computing cells. Therefore, a SPARC CPU board is used to control FDDI interface. The SPARC CPU board, which can be operated by the SUN OS, also makes it easier to develop software for network communication using TCP/IP (Transmission Control Protocol /Internet Protocol).

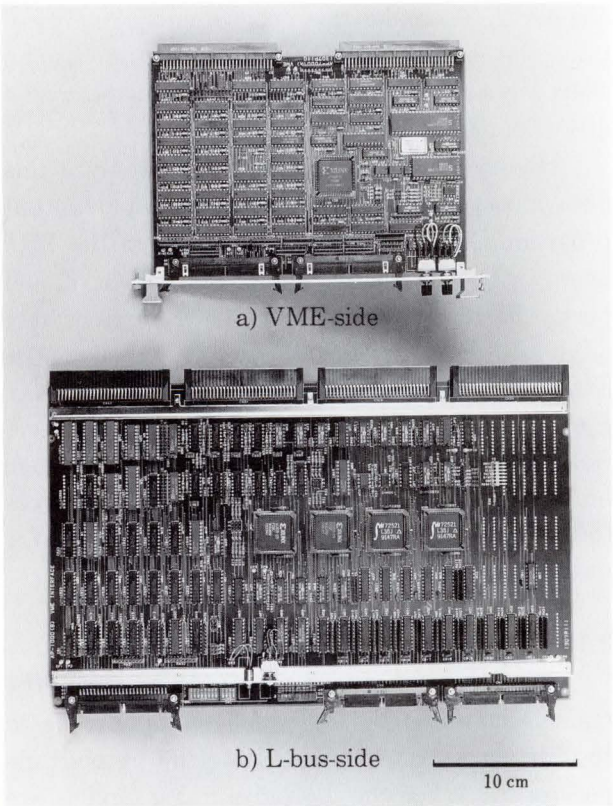


Fig. 4 – Option board.

The VME bus is a good choice for the common bus of cell input/output system because the available FDDI network interfaces and SPARC CPU boards use a VME bus for their external interface. However, to connect the FDDI network interface to a cell, an interface to convert bus communication protocol between L-bus and VME bus is necessary.

The cell input/output system consists of the cell, an L-bus/VME interface, VME bus, SPARC CPU board, FDDI interface, FDDI optical fiber link network, and mainframe (see Fig. 2).

2) Design of L-bus/VME interface hardware

A dedicated interface circuit board is designed for the connection to AP1000 cells (see Fig. 3). This interface makes VME peripherals act as cell slaves. Because the L-bus is a synchronous data-address multiplexed bus while the VME bus is an asynchronous data-address separated bus, bus-timing adjustments and protocol conversions are indispensable.

The interface circuit consists of two parts connected via bus buffers that compensate for differences in bus timings. The VME-side circuit controls VME bus protocol and VME interruption, and the L-bus-side circuit controls L-bus protocol and L-bus interruption. This latter circuit contains an 8 Kbytes first-in first-out (FIFO) memory to compensate for the differences in the data transfer speeds of two buses (see Figs. 3 and 4).

3) Function of L-bus/VME interface hardware

The L-bus/VME interface is designed to enable transparent mutual access to the address spaces of the VME bus and L-bus. If accesses from the VME bus and L-bus occurs simultaneously, the interface inhibits the access from the VME bus. Because of the mutual address mapping function of the interface, the direct data transfer between L-bus and the VME bus will be inefficient; therefore, a design to enable a data transfer via the FIFO memory is employed. The VME bus and L-bus can access the FIFO memory independently without disturbing each other. A data transfer from the L-bus to

the FIFO memory can be performed by a direct memory access (DMA) to cell memory. This DMA transfer is controlled by the empty/full status of the FIFO. For mutual notification of events on the busses, the master of each bus can interrupt the other bus.

3. Software design of the mainframe/AP1000 connection

3.1 Input/output processing structure of experimental high energy physics data analysis

High energy physics experimental data can be analyzed independently in an event-by-event manner by analysis codes. This parallelism has high potential for highly efficient parallel processing. Meanwhile, the experimental data is stored in a single data-file to be analyzed in a single batch. Therefore, even if the data is to be concurrently processed, it is unavoidable to distribute data to or gather data from cells both in an event-by-event manner.

Here, data division and combination must be done in an event-by-event manner. This type of data division requires deciphering of the data-file contents because of the complicated data organization, the structure of which is written in the data-file. The complicated data organization is unavoidable because of the different data types, namely, character, integer, and floating point, and the great differences in the amount of data among events.

3.2 Problems related to message passing programming

Existing programs for the current experimental data analysis are based on a procedure-call programming model of mainframe's single-addressing programming view. This means that all process communications and data transfers involved in procedure-calling are done via shared memory, e.g., common blocks, or via shared resources that can be accessed by using ordinary read and write instructions.

On the other hand, the programming model of the AP1000 distributed memory machine is message passing with synchronization. This means that all process communication, host-cell data transfer, and control of host and cell

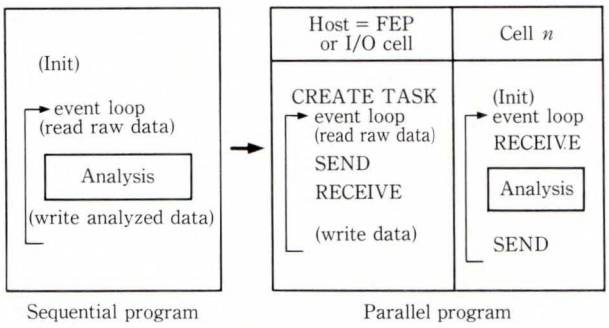


Fig. 5 – Parallelization on distributed-memory machine AP1000.

processes are to be performed via explicit description of message passing primitives. This difference in programming model causes the machine porting problem described below.

To run a data analysis program on computation cells which have no peripheral for input/output, the input/output procedures for serial programs ported from the mainframe must be separated as shown in Fig. 5. Figure 5 also shows how these procedures should be distributed among the input/output cells described in chapter 2. To make this programming modification, the structure of data allocation among shared resources, namely the main memory and disk file, must be clearly defined.

However, it is very difficult to analyze this data structure, partly because the input/output programming of our applications have the high energy physics-specific file-access procedure, and partly because common blocks are extensively used as a data interface between the data-file read/write subroutine and the other subroutines. For our applications, the programming modification is not an easy task because it requires huge amounts of replacements of the data interfaces for shared memory by the data interfaces for message communications.

3.3 Programming with server-client model

There are two programming models, the master-slave and the server-client, applicable to the file-accesses to a single data-file by multiple processors. When these programming models are applied to the event-independent parallel proces-

sing of the experimental data analysis, concurrent computations have the following characteristics.

1) Data division:

In the case of the master-slave model, the decipherment for division of the input file into individual event data must be done by the master. In the case of the server-client model, this decipherment can be done both by the server and the clients.

2) Balancing computational loads among cells:

Usually, to realize highly efficient parallel processing, some management is required to achieve a uniform computational load among cells. In the case of the server-client model, no explicit control is necessary to balance the load, while monistic control by the master is required in the case of master-slave model.

Because of these characteristics, the server function for the server-client model has a generality that makes application-specific server program unnecessary; whereas, for the master-slave model, a master program must be developed individually for each application.

In addition, from the viewpoint of parallel programming workload, the two programming models can be compared as follows:

3) Separation of input/output procedures from the analysis programs:

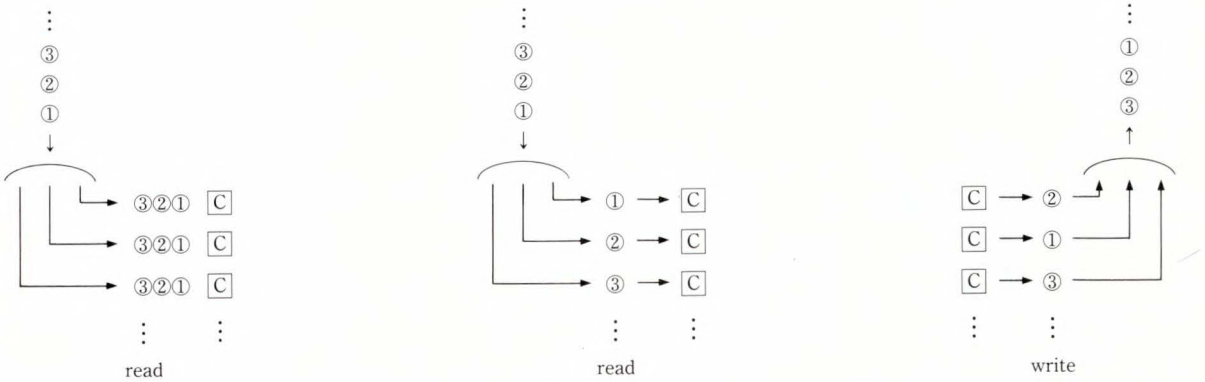
Because actual input/output must be performed on remote processors other than computational cells, the input/output procedure must be separated from the program. If this separation is to be done at the Fortran syntax-level, the programming workload, as mentioned above, is very large.

The only practical and effective way to reduce the programming workload for separating the input/output procedures from the analysis program is to do it not at the Fortran syntax-level but at the level of input/output mechanism servicing to Fortran programs. This approach makes it practical to perform the decipherment for the above mentioned data division on the computation cells, as is the case for the server-client model. However, in the case of the master-slave model, a data division decipherment by the slaves on the computation cells has no use on the actual data division. Hence, separating the input/output procedure at the input/output mechanism-level is feasible for the server-client model, but not for the master-slave model.

From the viewpoint of reducing user's workload, above characteristics suggest that input/output processing should be based on server-client model.

3.4 Mechanism to realize server-client model in input/output processes

Based on the server-client model, the input/output functions of analysis programs are replaced by cooperative functions of the server and the client. This server-client-based file-access can be realized by the distributed file-access mechanism[4], namely by distributing



a) Class-1: Reading in batches b) Class-2: Reading event-by-event c) Class-3: Writing event-by-event

Fig. 6 – Patterns of file-access from the experimental data analysis programs.

input/output processes between computation cells and input/output cells. In this mechanism, the client makes input/output requests to the server which resides in the remote input/output cell, and the server executes the actual disk input/output processes. In the actual implementation of this mechanism, a file-access management by using a file buffer and a buffer pointer is required.

For multiple clients to be able to request the server to read/write data from/to a single data-file, a control over concurrent requests is necessary. The type of control to be used depends on the file-access pattern. In the experimental data analysis, the file-access patterns are classified as follows:

Class-1: In a "batched manner" when the same data stream is read by multiple clients from a single data-file {see Fig. 6a)}

Class-2: In an "event-by-event manner" when data different from client to client is read from a single data-file {see Fig. 6b)}

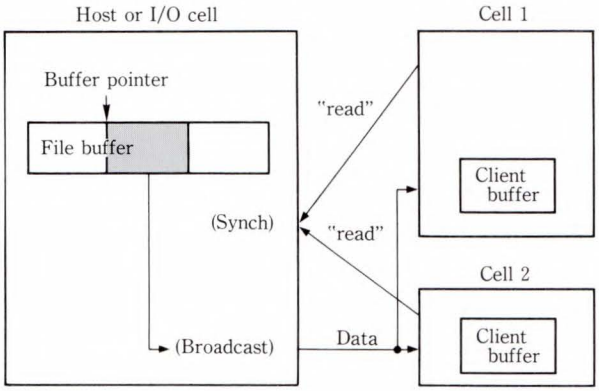
Class-3: In an "event-by-event manner" when data different from client to client is written to a single data-file {see Fig. 6c)}

An example of a class-1 file-access is the input of analysis parameters which have to be read at the beginning of the program. An example of a class-2 file-access is the input of experimental raw data. Examples of a class-3 file-access are outputs of analysis results and messages.

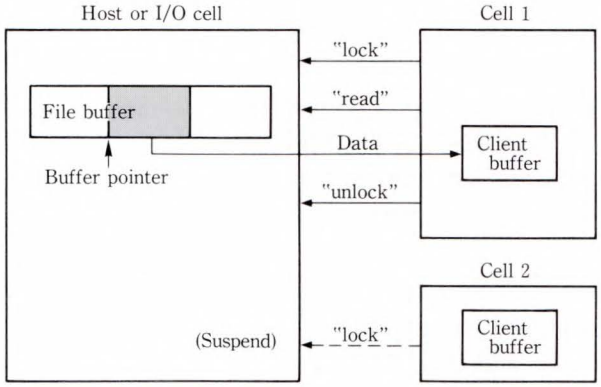
To replace the processing of these three file-accesses classes with concurrent processing among multiple clients and a single file server, input and output system must have the following mechanisms:

Mechanism-1: For a class-1 file-access, the server waits until all clients have requested data input. It then broadcasts the requested data and updates the buffer pointer {see Fig. 7a)}.

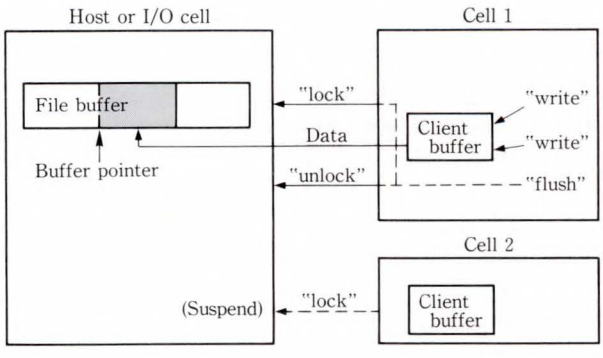
Mechanism-2: For a class-2 file-access, when the server receives an access request from a client, the server locks the buffer so that other clients can not access it {see Fig. 7b)}. After the accessing client has finished the access, the server unlocks



a) Mechanism when single file is read synchronously



b) Mechanism when single file is read event-by-event



c) Mechanism when single file is written event-by-event

Fig. 7 – Access mechanisms for sharing of single file by multiple clients.

the buffer upon a request from the client so that another client can access it.

Mechanism-3: For a class-3 file-access, outputs from the user program are stored in the buffer on the side of client. After all outputs for the analysis of one event data have been written in the client buffer, the server locks the server buffer on receiving a request from the client. Then, the client flushes the contents of the client buffer to the server buffer followed by unlocking of the server buffer {see Fig. 7c)}.

If the file-access is uniformly distributed over the analysis of an event, as is the case for the message output, the exclusive file-access mechanism serializes not only the file-access but also the analysis procedures. Fortunately, because a class-3 file-access is a write access, the buffering-and-flush mechanism described above can minimize the ill effect caused by serialization.

3.5 Clusterization of input/output function

1) Serialization and the distributed file-access mechanism

As mentioned in the previous section, the input/output processes must be serialized for experimental raw data and analyzed data. The control for exclusive file-access by the lock/unlock mechanism is to guarantee this serialization. This means that input/output procedures between lock and unlock operations must be serialized from client to client. Therefore, the maximum number of clients that can be supplied with data without a waiting time in average, is limited not only by the data transfer-speed but also by the time taken by input/output procedures.

$$N_c = T_c / (T_d + T_p)$$

N_c : Maximum effective number of clients

T_c : Time for computation, excluding input/output procedures

T_d : Time for data transfer

T_p : CPU-time for input/output procedures

Therefore, the maximum number of clients effective to derive performance from cells is limited by the ratio T_c over T_p, which is mostly independent of cell and input/output performance. Actually, in one experimental data analysis program, the ratio T_c over T_p is about 110. This means that an effective CPU power of a cell configuration having more than 110 cells will not exceed that of 110 cells.

When server-client-based file-access is applied to an application program which requires input/output serialization, to achieve scalable CPU power, multiplication of the server is necessary.

2) Hierarchical server function

In general, the mainframe to be used as an external file system of the AP1000 is run in a multi-programming mode. In this mode, the execution right is transferred from one job to another at every file-access. If the server-client communication with short data length and congested traffic is directly extended to the mainframe, the communication will not be efficient. Therefore, we employed a decomposition of the server into two parts to improve communication efficiency. One server, called the cell-cluster server, resides in the input/output cell of the cell-cluster described in Chap. 2, which directly processes requests from clients (see Fig. 8). The other server, called the mainframe server, resides in the mainframe, which processes requests for file-access and data transfer from the cell-cluster server (see Fig. 8). The configuration of cell-cluster server is

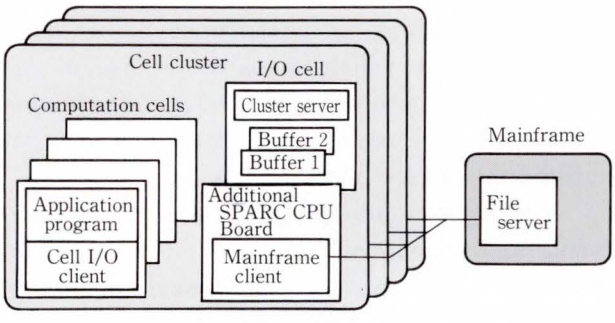


Fig. 8 – Task assignment to hardware.

described below.

To achieve a high processing performance, the sending of requests to the mainframe and their incidental processing, and the reception of requests from clients on computation cells and their incidental processing should be parallelized as much as possible. Therefore, the server task that serves clients in cells and the client task that makes requests to the mainframe file server (called the mainframe client) are made to act asynchronously. This is done by making them independent processes and by using double buffer that is accessible both from the client and server processes. The cell-cluster server process resides in the input/output cell, and the mainframe client process resides in the SPARC CPU board (see Fig. 8). To assist the parallel action of the two processes, the client process makes request to mainframe before requests are issued by the cell clients in input mode.

3.6 Implementations

The software components that organize the mechanisms described in the previous sections are the cell clients, the cell-cluster server, the mainframe client, and the mainframe file server.

The cell clients consist of replacements for system-call and user-callable modules. The replacements of system-call that are called from Fortran object code issue requests for data transfer using the Fortran input/output buffer as a transfer unit. These requests are made to the cell-cluster server by sending messages using communication library routines. For an exclusive file-access in mechanism-2 (described in the section 3.4), user callable modules to run on computation cells which lock or unlock the server buffer are prepared. For the buffering and flushing functions in mechanism-3, a user-callable module which changes the size of the Fortran input/output buffer, and a user-callable module which flushes the Fortran input/output buffer contents are prepared.

These user-callable modules and replacements of system-call manage all of Fortran input/output operations in the application program by referencing and updating the file-access information table. This table contains information to identify the file in the server and information to count the length of the already-accessed part of the buffer. In the read mode, the replacements of input/output system call not only receive input data from the server, but also notify the server of the length of data that has been read from the Fortran input/output buffer by the application program.

On the cell-cluster server side, the server manages the switching of buffers from file to file or from one side of a doubled server buffer to the other side. Also, the server processes requests from clients based on the file-access information table on the cluster-server side. This table contains the following: information that identifies files in the server or the mainframe, information about the file-access pattern, information about file organization in the server and the mainframe, and the buffer pointers.

Control of exclusive access to a double buffer is done between the cell-cluster server process and the mainframe client process. On the mainframe client-process side, request commands are sent to the mainframe file server.

On the mainframe file server side, file opening, reading, writing, and closing are performed on request from the mainframe client. Also, conversion between the mainframe OS and UNIX[Note)] of the file organization and the data format, for example, character or floating point, is performed by the mainframe file server.

The communication between cell-cluster server and mainframe client is done on a message passing-basis by using dedicated device driver routines. The communication between the mainframe file server and the mainframe client is performed by using the socket of the connection-type which is appropriate for communication of large amount of data.

Additionally, a function that reads the definition file which contains the file-access information and a function that sends the file-access information to the cell input/output clients and cell-cluster server are required. A

Note: The UNIX operating system was developed and is licensed by UNIX SystemLaboratories, Inc.

module which is callable from the host program is prepared for these functions.

4. Effectiveness of the system

1) Use of mainframe as an external file system of the AP1000

The hardware and software we developed enable us to use a mainframe as an external file system of the AP1000.

The input/output performance of the system can be greatly upgraded by the above-mentioned configuration because the distribution of input/output processes resolves data-transfer bottlenecks on the AP1000 side. However, bottlenecks on the mainframe side, for example, bottlenecks in the mainframe input/output and FDDI optical fiber link network, may become a problem depending on the required CPU power and the input/output throughput for the data analysis.

Moreover, for a configuration with multiple cell-cluster servers, a single data-file on the mainframe side must be divided corresponding to multiple cell-cluster servers. This data-file division can only be performed by a user-defined program because the organization of data-file records varies from user to user. If data-file division is to be performed by the mainframe file server, uniformity will be lost because the mainframe file server will contain a part which does not follow the server-client model. Therefore a dedicated mainframe file server for each program is necessary in this case.

2) Input/output performance

Because the input/output hardware option for AP1000 cells is still under development, the hardware performance and software overhead have not yet been measured.

Evaluation of the AP1000 input/output performance using the prototype option connected to the mainframe via the FDDI optical fiber link network are planned for the near future. This evaluation will include the measuring of the communication performance between the SPARC CPU board and a cell and the effect of the buffer size of the cell-cluster server on the mainframe SPARC CPU board communication overhead. The evaluation results will be used to improve the system by optimizing the software.

```
      parameter (maxd=25000)
      common a(maxd),b(maxd),nc(5),m,mb
      integer evnum
      equivalence ( b(1),evnum )
      read( 5, 100 ) ( nc(i),i=1,5 )
  100 format( 5i4 )
      open( unit=70,file='evdatain',
     & access='sequential',form='unformatted' )
      open( unit=80,file='evdataout',
     & access='sequential',form='unformatted' )
      do 10 i=1,100000
      read( 70, end=999 ) m,( a(j),j=1,m )
c----- event reconstruction subroutine
      call reconst
      write( 80 ) mb,( b(j),j=1,mb )
      write( 6, 200 ) evnum
  200 format( ' Event number ',i5,' processed.' )
   10 continue
  999 close ( 70 )
      close ( 80 )
      stop
      end
```

Fig. 9 – Simplified model program of experimental data analysis programs.

3) Parallel programming

Owing to the distributed file-access system, virtually no modification of the input/output processes is required for parallelizing the high energy physics experimental data analysis programs.

To demonstrate the effectiveness of the new environment, a model program of the high energy physics experimental data analysis that has been simplified by focusing on the input/output is used (see Fig. 9). If this program is parallelized by using the usual communication primitives, for example, send and receive, the number of source lines will increase considerably and extra complexity will be introduced as shown in Fig. 10. The parallelized version of this program by using the new file-access environment is shown in Fig. 11. As can be seen, no rewriting is necessary except for the addition of only five statements. The contents of the file used to notify the system of the file-access type and the file name is shown in Fig. 12. Simplicity in handling input/output processes of the parallelized program will be much helpful for large, real-world application programs.

4) Parallel processing efficiency

In concurrent input/output processes based on the server-client model, cell programs that

```
      SUBROUTINE CHMAIN
      parameter (maxd=25000)
      common a(maxd),b(maxd),nc(5),m,mb
      integer evnum
      equivalence ( b(1),evnum )
      CALL CCNFXY( 64,1 )
      CALL CCREAT( CC, 'CCMAIN.OUT', 30 )
       read( 5, 100 ) ( nc(i),i=1,5 )
  100 format( 5i4 )
      CALL CBROAD( 30, 1, NC, 4*5, CC )
      open(unit=70,file='evdatain',
     & access='sequential',form='unformatted' )
      open(unit=80,file='evdataout',
     & access='sequential',form='unformatted' )
      DO 5 I=0,63
      READ( 70, END=9999 ) M,(A(J),J=1,M)
      CALL CLASND(I, 30, 1, M, 4, CC )
      CALL CLASND(I, 30, 1, A, 4*M, CC )
    5 CONTINUE
      READ( 70, END=9999 ) M,(A(J),J=1,M)
       do 10 i=1,100000
      CALL CRECV( CID )
      CALL CRDMSG( CID, 4, NRSIZE )
      CALL CLARCV( CID, 30, 1, MB )
      CALL CRDMSG( MB, 4, NRSIZE )
      CALL CLARCV( CID, 30, 1, B )
      CALL CRDMSG( B, 4*MB, NRSIZE )
      CALL CLASND( CID, 30, 1, M, 4, CC )
      CALL CLASND( CID, 30, 1, A, 4*M, CC )
       read( 70, end=99 ) m,( a(j),j=1,m )
       write( 80 ) mb,( b(j),j=1,mb )
       write( 6, 200 ) evnum
  200 format( ' Event number ',i5,' processed.' )
   10 continue
   99 DO 20 I=0,63
      CALL CRECV( CID )
      CALL CRDMSG( CID, 4, NRSIZE )
      CALL CLARCV( CID, 30, 1, MB )
      CALL CRDMSG( MB, 4, NRSIZE )
      CALL CLARCV( CID, 30, 1, B )
      CALL CRDMSG( B, 4*MB, NRSIZE )
      WRITE( 80 ) MB,( B(J),J=1,MB )
      WRITE( 6, 200 ) EVNUM
   20 CONTINUE
  999 close ( 70 )
      close ( 80 )
      return
      end

      SUBROUTINE CCMAIN
      parameter (maxd=25000)
      common a(maxd),b(maxd),nc(5),m,mb
      integer evnum
      equivalence ( b(1),evnum )
      CALL CGTID( CID )
      CALL CHRECV( 1, NC )
      CALL CRDMSG( NC, 4*5, NRSIZE )
       do 10 i=1,100000
      CALL CHRECV( 1, M )
      CALL CRDMSG( M, 4, NRSIZE )
      CALL CHRECV( 1, A )
      CALL CRDMSG( A, 4*M, NRSIZE )
c----- event reconstruction subroutine
      call reconst
      CALL CHSEND( 1, CID, 4, CC )
      CALL CHSEND( 1, MB, 4, CC )
      CALL CHSEND( 1, B, 4*MB, CC )
   10 continue
      return
      end
```

Fig. 10 – Host and cell programs using message passing mechanism.

have finished analyzing an event receive new data in turns. As a result, the computational load, by itself, is evenly distributed among cells except during job start-up and termination. Therefore, for long jobs in which the job start-up time and termination time are negligible, a good load-balance and highly efficient parallel processing within the cell-cluster can be expected.

5. Future plans

5.1 Functions to be developed in the future

1) Multi-user function

In general, the abundant CPU resources of highly parallel computers must be shared effectively among multiple users. Our applications of highly parallel computers require a function that properly starts and terminates the user's application programs and allocates the file-access system to multiple cell-clusters in accordance with the number of cell-cluster servers.

In our applications, the appropriate number of cells for each cluster can be evaluated based on a property of the application program, namely the ratio of CPU-time required for input/output procedures to the time required for numerical computation. (This ratio is called the input/output procedure ratio.) The maximum number of effective cells in one cell-cluster that can be used without deteriorating parallel processing efficiency is determined by serialization of the input/output procedures among cells. This maximum number varies with the input/output procedure ratio of each program.

Operability will be improved very much by a multi-user function that enables configurations having differing numbers of cells and enables configuration changes without the need to physically move the cell option.

```
      SUBROUTINE CCMAIN
      parameter (maxd=25000)
      common a(maxd),b(maxd),nc(5),m,mb
      integer evnum
      equivalence ( b(1),evnum )
      read( 5, 100 ) ( nc(i),i=1,5 )
  100 format( 5i4 )
      open( unit=70,file='evdatain',
     & access='sequential',form='unformatted' )
      open( unit=80,file='evdataout',
     & access='sequential',form='unformatted')
      do 10 i=1,100000
      CALL CFPROT( 70, CC )
      read( 70, end=999 ) m,( a(j),j=1,m )
      CALL CFFREE( 70, CC )
      c----- event reconstruction subroutine
      call reconst
      write( 80 ) mb,( b(j),j=1,mb )
      write( 6, 200 ) evnum
  200 format( ' Event number ',i5,' processed.' )
      CALL CFLUSH(6,CC)
      CALL CFLUSH(80,CC)
   10 continue
  999 close ( 70 )
      close ( 80 )
      return
      end
```

Fig. 11 – Parallel program using distributed I/O mechanism.

```
unlocal = h
task = 30
stdin      :cardinp  ::unlocal,synch
evdatain   :rawdata  ::unlocal,read
evdataout  :dstdata  ::unlocal,write
stdout     :outlist  ::unlocal,write
```

Fig. 12 – Description of file acces information.

2) Operational function

At present, since most users are familiar with mainframe computers, they should be able to choose an operating environment which enables them to use a parallel processor as if it were a mainframe. Or, users should be able to choose an operating environment which enables batch-job management through a supervisor on a mainframe. To this end, we are now developing a prototype tool which enables execution of user programs on a parallel processor by submitting a batch job to a mainframe.

5.2 Expected uses in computations of experimental high energy physics

In Japan, our system will be used in experiments at the B-Factory project as extension of the TRISTAN accelerator and at the Japan Linear Collider project, which collides electrons and anti-electrons having energies of one terra-electron volts are planned[1]. In these experiments, the amount of experimental data and the required amount of computation for data analysis are expected to be 100 to 1 000 times greater than the level of previous experiments. The main part of the data analysis system for these experiments is expected to be built by a highly parallel processor with our system or its follow-on systems.

6. Conclusion

An input/output system that enables a mainframe to be used as an external file system of AP1000 has been designed for the high energy physics experimental data analysis. Prototype input/output hardware for the AP1000 cells is now being built. The input/output system software enables parallelization of the analysis programs with virtually no modification and with autonomous balancing of computational loads among cells. Also, the input/output system software enables the application programs on the AP1000 cells to access mainframe files via the prototype hardware. We expect to achieve scalability of input/output performance with respect to the number of computation cells, and therefore scalability of the effective computational power, by using multiple links between the cell-clusters and the mainframe channels via the input/output system. The system will be used in experiments in the near future.

7. Acknowledgement

The authors would like to express their sincere thanks to Dr. Yoshio Watase of the National Laboratory for High Energy Physics, Japan, for planning and conducting the research, and also for giving suggestion to the design of the prototype system.

8. Appendix

The computations for the current in high energy physics experiments have already been heavily CPU demanding. For example, the 30 GeV + 30 GeV e^- e^+ collision experiments of the TRISTAN accelerator at KEK (National Laboratory for High Energy Physics, Japan) require more than 100 MIPS of computing power for the offline data analysis. This means that typical event reconstruction from the raw data requires 0.5 seconds per event on a 50 MIPS class of computers. A typical detector simulation using the Monte Carlo method takes from 5 to 10 seconds per event on the same machine. Also, online data acquisition by the VAX-clustered DAQ system requires 0.2 seconds

KEK is now involved in the following accelerator activities:

1) TRISTAN data analysis for higher-intensity runs starting from 1991,
2) the B-Factory project at KEK, as an extension of TRISTAN activities,
3) participation in the world-wide collaboration for the SSC (Superconductive Super Collider) 20 TeV + 20 TeV proton-proton collision experiments that will start at Dallas in Texas in 1999,
4) the Japan Linear Collider at KEK (1 TeV + 1 TeV e^- e^+ collision). (This collider is currently under feasibility study.)

Since all of these experiments will treat higher energy phenomena involving larger numbers of events and particles with higher resolution detectors, the required amount of computations will be far beyond the present level. (For example, 100 to 1 000 times that required in offline processing of today's TRISTAN experiments, and considerably more than 100 000 times that for online processing.) It is therefore obvious that only the parallel processing approach has a possibility of providing the required computer power for these experiments. The physical and computational dimensions of two future accelerators, the B-Factory and the SSC, are compared with those of TRISTAN in Table 1.

Table 1. Experimental high energy physics computations

	TRISTAN (KEK 1986)	B-Factory (KEK 199X)	SSC (SSC Lab.1999)
BEAM energy	(e^-, e^+) 30×30 GeV	(e^-, e^+) 8×3.5 GeV	Proton 20×20 TeV
Circumference	3 km	1.2 km	87 km
Collision frequency	5 μs	10 ns	16 ns
No. signal channel	30 000	80 000	1 000 000
No. particles/event	10	5	300-500
Data size/event	20 Kbytes	30 Kbytes	1 000 Kbytes
Trigger rate	2 Hz	200 Hz	10-100 kHz (2nd Level)
Data rate			
Raw data	50 Kbytes/s 500 Gbytes/year	6 Mbytes/s 60 Gbytes/year	10-100 Mbytes/s 100-1 000 Tbytes/year
DST data	5 Kbytes/s 50 Gbytes/year	3 Mbytes/s 30 Tbytes/year	
Required computer power			
Online processing	1 MIPS	3 000 MIPS	100 000-1 000 000 MIPS (Event pre-processing)
Offline processing	100 MIPS	10 000 MIPS	10 000-100 000 MIPS

References

1) Matsuura, T., Ichikawa, S., Ishida, N., Shiraishi, H., Ikesaka, M., Takaiwa, Y., Kanzaki, J., Amako, K., Tsuboyama, T., Miyamoto, A., Nozaki, T., Ichii, S., Fujii, H., Manabe, A., Watase, Y.: Application of Fujitsu CAP-II to KEK Experimental HEP Computations[1], Proc. Conf. Computing in High Energy Physics '91, 1991, Tsukuba, Japan Universal Academy Press Inc., pp. 129-138.
2) Ishihata, H., Horie, T., Inano, S., Shimizu, T., and Kato, S.: An Architecture of Highly Parallel Computer AP1000. Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Processing, 1991, B. C., Canada, **1**, 1991, pp. 13-16.
3) Shimizu, T., Horie, T., and Ishihata, H.: Low-Latency Message Communication Support for the AP1000. 19th Int. Symp. Comput. Architecture, 1992, Gold Coast, Australia, *ACM SIGARCH Computer Architecture News*, **20**, 2, pp. 288-297 (1992).
4) Satyanarayanan, M.: Scalable, Secure and Highly Available Distributed File Access. *IEEE Computer*, 5, pp. 9-21 (1990).

Shin-ichi Ichikawa

Parallel Computing Research Center
FUJITSU LABORATORIES, KAWASAKI
Bachelor of Science
Tokyo University of Education 1977
Master of Science, Nuclear Physics
Tohoku University 1979
Specializing in Application of Vector and Parallel Processing to Scientific Researches

Atsushi Manabe

Computing Center
National Laboratory for High Energy Physics
Bachelor of Science
University of Tsukuba 1983
Dr. of Science, Nuclear Physics
University of Tsukuba 1988
Specializing in Nuclear Physics

Toshihiko Matsuura

Systems Engineering Dept., Tsukuba
FUJITSU LIMITED
Bachelor of Science
The University of Tokyo 1970
Ph. D., Theoretical Nuclear Physics
The University of Tokyo 1975
Specializing in Application of Vector and Parallel Processing to Scientific Researches

UDC 681.322:727.57

Activities in the Fujitsu Parallel Computing Research Facilities

● Takao Saito ● Koichi Inoue

(Manuscript received August 4, 1992)

The Fujitsu Parallel Computing Research Facilities (FPCRF) offers an optimal environment for researchers in parallel processing and its applications. Parallel processing is a promising technique for high-speed, large-scale computation that uses hundreds to thousands of interconnected processors. This paper gives an overview of FPCRF and the research into parallel software and algorithms being done there.

1. Introduction

There are growing demands for high-speed large-scale computation, especially in the scientific and engineering fields. For example, problems in quantum chromodynamics (QCD) require some 10^{17} floating point operations[1]. Even a 1-GFLOPS computer would require several years to solve such a problem, but the performance of single-processor computers is nearing a limit, that parallel processing is expected to overcome.

The performance of parallel processors does, in general, not increase linearly with the number of processors due to interprocessor communication overhead and load imbalance, i.e., the difference in the workload of different processors.

Before parallel processing can become a widely-used technology, more research is needed into the special algorithms and programs required to bring out the full potential of parallel processing.

Fujitsu Parallel Computing Research Facilities (FPCRF) was set up to provide an environment for research into tools for developing efficient programs and application domain specific algorithms.

2. Fujitsu Parallel Computing Research Facilities (FPCRF)

The FPCRF officially opened in June 1992 for the purpose of promoting parallel processing research and its applications, accumulating parallel processing techniques, and evaluating the architectural effectiveness of parallel computers. Highly parallel AP1000 computers[2] developed by Fujitsu Laboratories Ltd. have been installed as platforms for parallel computing research. The research environment is offered to researchers in universities and national laboratories.

FPCRF's activities are as follows:

1) Administration

Administration includes user support, resource scheduling, information logging, and user registration.

2) Technical information exchange

Annual meetings encourage FPCRF users to exchange ideas on the use of parallel computers and communicate results of their work in parallel computing techniques. User seminars are also held for discussing parallel processing techniques in specific user applications.

3) Research in parallel algorithms and software

Parallel processing techniques in various applications and evaluation of our parallel computers have been studied based on technical exchange and reports contributed by users. FPCRF also maintains a library for research results, including programs and documents, accessible to FPCRF users and Fujitsu researchers.

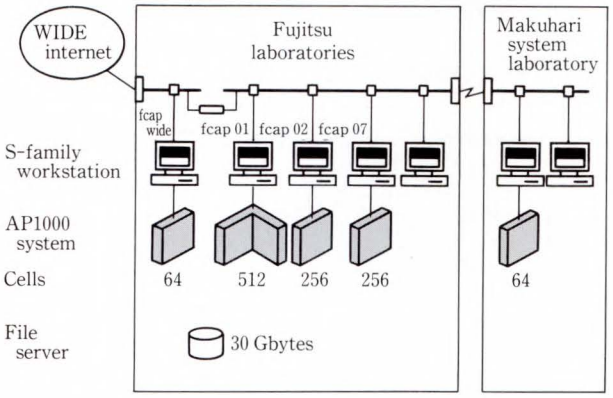


Fig. 1 – System configuration.

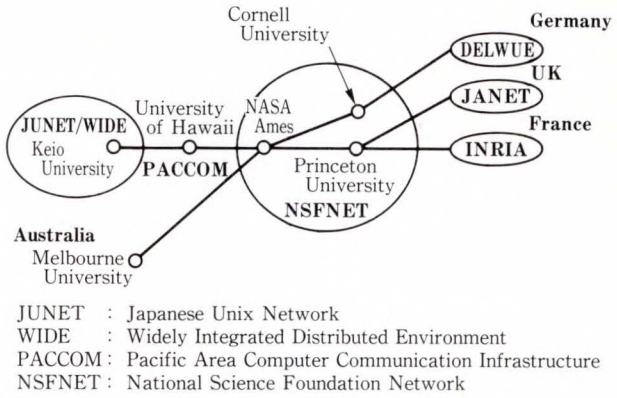


Fig. 2 – International academic network.

The first phase of this project will be completed in March 1995.

2.1 Equipment

The AP1000, an MIMD parallel computer with an architecture based on message passing, consists of up to 1 024 processor elements called cells. Each cell is connected to a broadcast network (B-net), which distributes programs and data from a host to the cells, and a torus network (T-net), for point-to-point communication between cells. FPCRF currently has five AP1000 systems – one with 512 cells, another with 128 cells and the other three with 64 cells – connected to a local area network (LAN) via S-family workstations (SUN4/330) (see Fig. 1). A 30-gigabyte file server on the LAN contains user programs and data. The LAN is connected via gateways to the academic networks in Japan and abroad.

2.2 Network

FPCRF is accessed using the Widely Integrated Distributed Environment (WIDE) internet, connected via a 64-kb/s line using the TCP/IP protocol. The internet was constructed under the direction of Associate Professor Jun Murai of Keio University. The WIDE internet connects universities and companies participating via operation centers in Sendai, Tokyo, Fujisawa, Kyoto, Osaka, and Fukuoka.

FPCRF is also accessed via JUNET, connected to the AP1000 systems via a 9.6-kb/s line using the unix-to-unix copy protocol

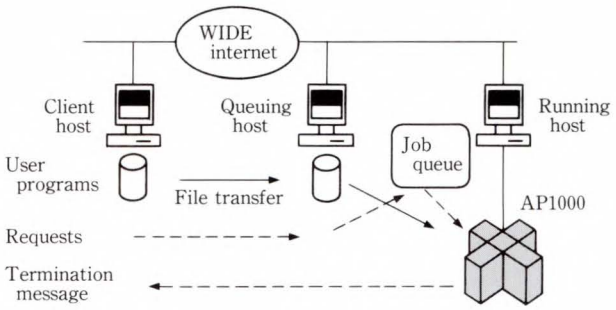


Fig. 3 – Batch operation.

(UUCP). Since communication data is relayed through nodes at fixed intervals in the same way as E-mail, this network is accessed in batches rather than online.

The WIDE operation center in Fujisawa is connected to the University of Hawaii via a 192-kb/s submarine cable. This network is part of the PACCOM project, setting up a linked computing environment for research activities along the Pacific Rim. Hawaii is connected to the US mainland, Australia, and New Zealand, and European countries can access FPCRF via the US (see Fig. 2).

2.3 Operation

The AP1000s at FPCRF run 24 hours a day. They are currently available for open use in the daytime and for batch operation at night. Open use means that users at FPCRF or via a network have exclusive use of the parallel computing systems for a fixed amount of time. This helps users who require lots of processing time and

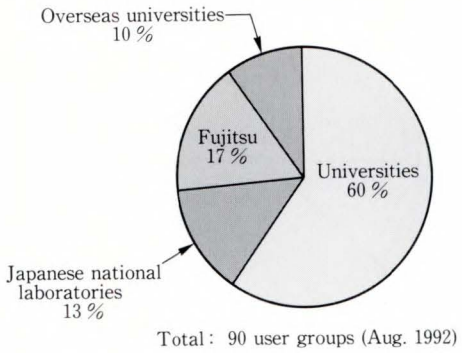


Fig. 4 – FPCRF users.

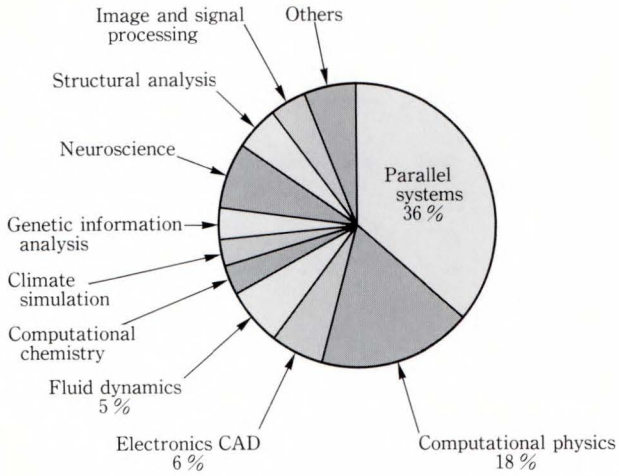


Fig. 5 – Applications studied at FPCRF.

have very large data files. A reservation is required and operations are scheduled monthly.

Requests to compile and run a program in batch mode are queued by the queuing host (see Fig. 3) and processed automatically by the running host during batch operation. Batch-mode users, who can request jobs at any time without a reservation, are notified automatically by E-mail when their programs have terminated.

2.4 FPCRF users

Among the organizations using FPCRF (see Fig. 4), there are eleven Japanese national and public research institutes, including the National Laboratory for High Energy Physics and the National Institute of Genetics, and fifty-four universities. There are also eight overseas organizations including the University of Heidelberg, the Federal Institute of Technology at Zurich, and the University of Manchester Institute of Science and Technology (UMIST).

3. Parallel computing applications

About one third of the research using FPCRF is in parallel systems such as parallelizing compilers and parallel languages, and about two thirds is in methods for solving parallel application problems (see Fig. 5).

3.1 Computational physics

Research in QCD[1)-3)] is one of the most active application fields at FPCRF. QCD is a quantum physics theory that describes the behavior of fundamental particles such as quarks and gluons. The QCD time-space field is represented by a four-dimensional lattice where sites (lattice points) and links connecting sites are associated with quark and gluon fields. QCD simulation is done using the Monte Carlo method. Statistical correlations between the links are then evaluated to obtain properties such as hadron masses. This method requires large amounts of calculation, and researchers have applied parallel processing by mapping the lattice onto the two-dimensional processor elements of the AP1000. Details are reported in this special issue[4)].

The Monte Carlo method has also been applied to radiative heat transfer analysis[5)], molecular gas dynamics[6)], and radiation shielding analysis[7)] with interesting results.

3.2 Computational chemistry

In molecular dynamics calculations, the macroscopic nature of molecules is studied by simulating particles at the molecular or atomic level. The force applied to an atom is calculated as the sum of forces resulting from other atoms. The velocity and position of an atom are then calculated from the force. In principle, each step of a molecular dynamics (MD) simulation requires $N(N-1)/2$ force calculations – an amount that increases enormously with the number of atoms.

In the parallel processing of AMBER[8)-9)], an MD simulation program, particle division is used to group particles equally and assign them

to processors to balance the load of calculations. During experiments using human lysosyme (20 thousand atoms) and hemoglobin (88 thousand atoms), the AP1000 achieved a performance 10 to 20 times faster than Fujitsu's M-780 mainframe.

Another parallelization method, domain decomposition, has been used for MD simulation. This splits the space occupied by the particles and assigns a processor to perform calculations for the particles in an individual area. This is effective for large-scale particle systems with short-range interaction[10].

3.3 Fluid dynamics

Fluid dynamics, used in wide range of fields such as civil engineering and the design of airplanes and automobiles, analyse fluid motion based on calculation of Navier-Stokes equations at grid points in three-dimensional space. To increase the simulation precision and to approximate the results of flow phenomena require a fine grid space, e.g., 10^{11} to 10^{16} grid points for turbulence analysis[11]. Space division used for parallel processing maps divided grids onto the processors. In wind engineering, the highly parallel processing achieved by reducing the communication overhead and using SOR methods is very promising[12].

3.4 Electronics CAD

The following parallel methods[13] are known for generating test patterns for logic circuits:

1) Fault partitioning

Faults in a logic circuit's fault list are divided into groups. Each processor calculates circuit inputs that can detect the assigned fault group.

2) Search space partitioning

The several combinations of inputs to a logic element that could detect a fault form a search space tree. Each branch of the tree is assigned to a processor.

3) Topological partitioning

A logic circuit is divided into parts, each assigned to a processor.

A new method[14] under study, which deals with error propagation paths, generates fault lists without fault simulation. Other applications of parallel processing in electronics CAD, including circuit simulation and LSI mask pattern generation, are discussed in this special issue[15), 16].

3.5 Climate simulation

To predict long-term changes in the earth's climate, atmospheric models are created and studied. These models range from a simple radiative-convective model to the three-dimensional general circulation model (GCM), which is based on equations of fluid motion, the first law of thermodynamics, and the law of conservation of mass. To simulate the model, the earth's atmosphere region is divided longitudinally, latitudinally, and vertically into a lattice, and the equations are solved on these lattice points. For parallelizing climate model MRI-GCM, developed by the Meteorological Research Institute, Japan, two mapping methods have been studied to balance computational load: the block partitioning, the assignment of each cluster of lattice points to a processor, and the dot partitioning, the assignment of each point to a processor.

3.6 Genetic information analysis

A protein is a folded three-dimensional amino acid sequence created based on information in the DNA. The properties of a protein are predicted using information found in similar proteins. Sequences have been analyzed to better understand the traces of evolution, which the sequence reflects. Genetic information analysis[17] uses a homology search and multiple alignment to compare each sequence with tens of thousands of the sequences in a database. The computation can be suitably performed in parallel because the large number of combinations to be checked are independent. Execution on the AP1000 using 512 processors to analyze 15 233 sequences is estimated to be about 9 times faster than that on Fujitsu's VP2400 supercomputer[18]. A large cell memory and direct disk access from each cell are necessary

for analyzing long sequences and reducing the need for data transfer from a host.

4. Conclusion

The Fujitsu Parallel Computing Research Facilities promotes the study of parallel processing by providing a state-of-the-art computing environment for researchers. User softwares developed there are registered at FPCRF as freewares for all users. Such softwares, in particular parallel programming languages, parallelizing compilers, and numerical computation libraries, will enhance the research environment. Parallel processing techniques in different fields have been studied through discussion with users, and the results will be fed back to parallel computer R&D-one of FPCRF's major activities.

References

1) Ohta, S.: Toward Lattice QCD Simulation on AP1000. *Nucl. Phys.*, **26B**, pp. 647-649 (1992).
2) Ishihata, H. et al.: Third Generation Message Passing Computer AP1000. Proc. Int. Symp. Supercomputing, 1991, pp. 46 -55.
3) Kawazoe, A. et al.: QCD on the Highly Parallel Computer AP1000. *Nucl. Phys.*, **26B**, pp. 644-646 (1992).
4) Okuda, M., Kawazoe, A., and Fujisaki, M.: A Parallelization Study of Quantum Chromodynamics Simulations on the AP1000. *FUJITSU Sci. Tech. J.*, **29**, 1 (Special Issue on Cellular Array Processor AP1000), pp. 84-96 (1993).
5) Taniguchi, H. et al.: Monte Carlo Simulation of Non-Gray Radiation Heat Transfer on Highly Parallel Computer AP1000. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P1-D-1-P1-D-5.
6) Yoshimura, S., and Yagawa, G.: Probabilistic Fracture Mechanics Analysis Using Massively Parallel Computers. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P1-I-1-P1-I-3.
7) Masukawa, F. et al.: Parallelization of Monte Carlo Code MCACE for Shielding Analysis and Measurement of Parallel Efficiency on AP1000. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P1-A-1-P1-A-8.
8) Sato, H., Tanaka, Y., and Yao, T.: Molecular Dynamics Simluation on an AP1000 Distributed Memory Parallel Computer. *FUJITSU Sci. Tech. J.*, **28**, 1, pp. 98 -106 (1992).
9) Iwama, H. et al.: Application of Parallel Computer to Molecular Dynamics. Proc. First Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P1-C-1-P1-C-2.
10) Brown D., and Clarke, J. H. R.: Parallelization Strategies for MD Simulation on the AP1000. Proc. Second Fujitsu-ANU CAP Workshop, 1991.
11) Utumi, Y. et al.: Study on Parallel Processing of Computational Fluid Dynamics – The Reduction of Communication Time and the Implementation of Red/Black Partitioning. (in Japanese), Proc. 5th Numerical Fluid Dynamics Symposium, 1992, pp. L-1-L-10.
12) Kato, S. et al.: Application of Massively Parallel Computer to Computational Wind Engineering. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P2-C-1-P2-C-8.
13) Klenke, R.: Parallel Processing Techniques for Automatic Test Pattern Generation. *IEEE Computer*, pp. 71 -83 (1992).
14) Minohara, T. and Tohma, Y.: Test Pattern Generation on Highly Parallel Processors. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P2-F-1-P2-F-4.
15) Kage, T., Niitsuma, J., and Teramae, K.: Highly Parallel Circuit Simulator on the AP1000: PARACS. *FUJITSU Sci. Tech. J.*, **29**, 1 (Special Issue on Cellular Array Processor AP1000), pp. 41-49 (1993).
16) Tsujimura, R., Manabe, Y., and Morishita, K.: LSI Mask Data Processing System: PRANCER. *FUJITSU Sci. Tech. J.*, **29**, 1 (Special Issue on Cellular Array Processor AP1000), pp. 78-83 (1993).

17) Kawai, M., Kishino, A., and Naito, K.: Rapid Analysis Methodology for Gene Sequence Using a Parallel Processor. *FUJITSU Sci. Tech. J.*, **27**, 3, pp. 270-277 (1991).

18) Sakamoto, M., and Kawamura, H.: Parallel Computation of Cavity Flow Using HSMAC. Proc. First Annual Users' Meeting, Fujitsu Parallel Computing Research Facilities, 1992, pp. P2-E-1-P2-E-2.

Takao Saito

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electronic Eng.
Hokkaido University 1974
Master of Electronic Eng.
Hokkaido University 1976
Specializing in Parallel Processing and CAD

Koichi Inoue

Parallel Computing Research Center
FUJITSU LABORATORIES,
KAWASAKI
Bachelor of Electronics
Kobe University 1983
Specializing in Parallel Processing

International Network

Offices

Fujitsu Abu Dhabi
P.O. Box 47047 Suite 802.
A1 Masadood Tower,
Sheikh Hamdan Street,
Abu Dhabi. U.A.E.
Telephone : (971-2)-333440
FAX : (971-2)-333436

Amman Project Office
P.O. Box 5420, Ammán. Jordan
Telephone : (962)-6-662417
FAX : (962)-6-673275

Bangkok Office
3rd Floor, Dusit
Thani Bldg., 1-3, Rama IV,
Bangkok10500, Thailand
Telephone : (66-2)-236-7930
FAX : (66-2)-238-3666

Beijing Office
Room 2101, Fortune Building,
5 Dong San Huan Bei-Lu,
Chao Yang District, Beijing,
People's Republic of China
Telephone : (86-1)-501-3261
FAX : (86-1)-501-3260

Brussels Office
Avenue Louise 176, Bte 2
1050 Brussels, Belgium
Telephone : (32-2)-648-7622
FAX : (32-2)-648-6876

Harare Office
CABS House, Corner Central Avenue,
4th Street, Harare, Republic of Zimbabwe
Telephone : (263-4)-732-627
FAX : (263-4)-732-628

Hawaii Office
6660 Hawaii Kai Drive, Honolulu,
Hawaii 96825, U.S.A.
Telephone : (1-808)-395-2314
FAX : (1-808)-396-7111

Jakarta Representative Office
16th Floor, Skyline Bldg.,
Jalan M.H. Thamrin No.9,
Jakarta, Indonesia
Telephone : (62-21)-333245
FAX : (62-21)-327904

Kuala Lumpur Liaison Office
Letter Box No.47, 22nd Floor,
UBN Tower No.10, Jalan P.
Ramlee, 50250, Kuala Lumpur, Malaysia
Telephone:(60-3)-238-4870
FAX : (60-3)-238-4869

Munich Office
c/o Siemens Nixdorf Informationsysteme
AG, D8 SC,
Otto-Hahn Ring 6, D-8000,
München 83, Germany
Telephone:(49-89)-636-3244
FAX : (49-89)-636-45345

New Delhi Liaison Office
Mercantile Home
1st Floor, 15 Katsurba, Gandhi Marg
New Delhi-110001, India
Telephone : (91-11)-331-1311
FAX : (91-11)-332-1321

New York Office
680 Fifth Avenue, New York,
N.Y. 10019, U.S.A.
Telephone : (1-212)-265-5360
FAX : (1-212)-541-9071

Oficina Informativa en Mexico
Paseo de la Reforma No.255 Oficina 13-A
Col. Cuauhtemoc 06500. Mexico. D. F.
Telephone : (525)-592-3940
FAX : (525)-592-3985

Shanghai Office
Room 705, West Podium Office Building,
Shanghai Centre,
1376 West Nanjing Road,
Shanghai 200040, People's Republic of China
Telephone : (86-21)-279-8410
FAX : (86-21)-279-8411

Sucursal de Colombia
Cra. 9a. A No. 99-02 Ofc. 811
Edificio Seguros Del Comercio
Santa Fe De Bogotá, Colombia
Telephone : (57-1)-618-3147
FAX : (57-1)-618-3134

Taipei Office
Sunglow Bldg., 66, Sung Chiang
Road, Taipei, Taiwan
Telephone : (886-2)-551-0233
FAX : (886-2)-536-7454

Washington, D.C. Office
1776 Eye Street, N.W.,
Suite 880, Washington, D.C.,
20006, U.S.A.
Telephone : (1-202)-331-8750
FAX : (1-202)-331-8797

Subsidiaries

ASIA AND OCEANIA

Fujitsu Australia Ltd.
Fujitsu Australia Software Technology. Pty Ltd.
Fujitsu Component (Malaysia) Sdn. Bhd.
Fujitsu Electronics (Singapore) Pte. Ltd.
Fujitsu Hong Kong Ltd.
Fujitsu Korea Ltd.
Fujitsu Microelectronics Asia Pte. Ltd.
Fujitsu Microelectronics (Malaysia) Sdn. Bhd.
Fujitsu Microelectronics Pacific Asia Ltd.
Fujitsu New Zealand Ltd.
Fujitsu (Singapore) Pte. Ltd.
Fujitsu (Thailand) Co., Ltd.
Fujitsu Trading Ltd.

NORTH AMERICA

Fujitsu America, Inc.
Fujitsu Business Communication Systems, Inc.
Fujitsu Canada, Inc.
Fujitsu Computer Packaging Technologies, Inc.
Fujitsu Computer Products of America, Inc.
Fujitsu Microelectronics, Inc.
Fujitsu Network Switching of America, Inc.
Fujitsu Network Transmission Systems, Inc.
Fujitsu Systems Business of America, Inc.
Fujitsu Systems Business of Canada, Inc.
Open Systems Solutions, Inc.
Fujitsu Personal Systems, Inc.

EUROPE

Fujitsu Deutschland GmbH
Fujitsu España, S.A.
Fujitsu Europe Ltd.
Fujitsu Europe Telecom R&D Centre Ltd.
Fujitsu Finance (U.K.) PLC
Fujitsu France S.A.
Fujitsu International Finance (Netherlands) B.V.
Fujitsu Italia S.p.A.
Fujitsu Microelectronics Ireland Ltd.
Fujitsu Microelectronics Italia S.r.l.
Fujitsu Microelectronics Ltd.
Fujitsu Mikroelektronik GmbH
Fujitsu Nordic AB
Fujitsu Systems Business of Europe Ltd.
Fulcrum Communications Ltd.

LATIN AMERICA

Fujitsu do Brasil Ltda.
Fujitsu Vitória Computadores e Serviços Ltda.

FUJITSU LIMITED

6-1, Marunouchi 1-chome, Chiyoda-ku, Tokyo 100, Japan
Phone: National (03) 3216-3211 International (Int'l Prefix) 81-3-3216-3211 Telex: J22833 Cable: "FUJITSULIMITED TOKYO"
Fax: National (03) 3213-7174 International (Int'l Prefix) 81-3-3213-7174