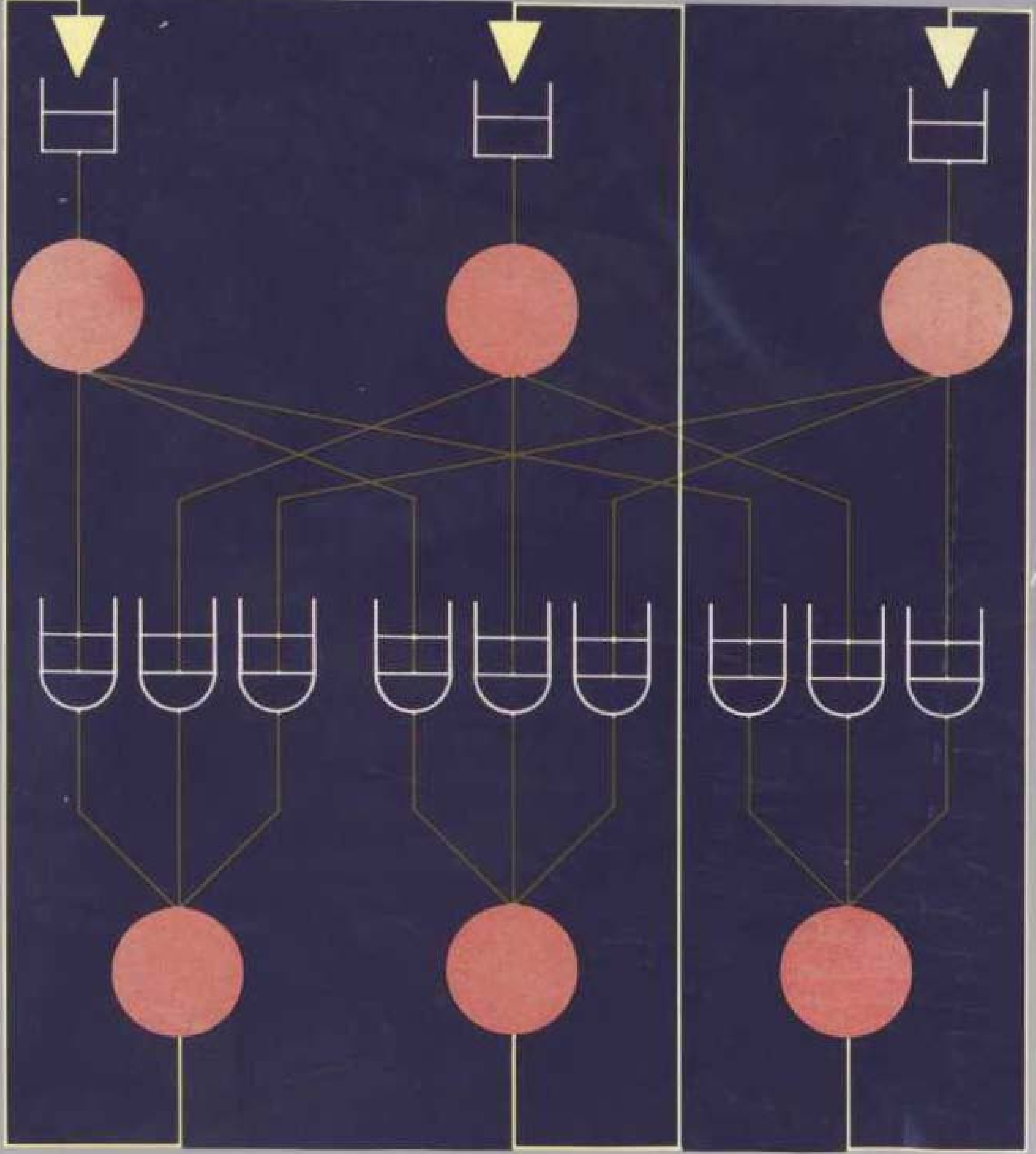
Parallel Processing

The Cm* Experience

Edward F. Gehringer
Daniel P. Siewiorek
Zary Segall



Parallel Processing

The Cm* Experience

Parallel Processing
The Cm* Experience

Edward F. Gehringer
North Carolina State University

Daniel P. Siewiorek
Carnegie-Mellon University

Zary Segall
Carnegie-Mellon University

digital

Digital Press

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

Order number EY-6706E-DP

This book was formatted by its authors using the Scribe Document Production System (Unilogic Ltd.). The electronic files have been converted to type by E/F Typographic Laboratories, Pittsburgh. The illustrations were prepared under the authors' direction with the DP program on a PERQ system. Editorial Inc., Rockport, Massachusetts, handled design and production.

Library of Congress Cataloging-in-Publication Data

Gehringer, Edward F. (Edward Francis)
Parallel processing.

Bibliography: p.
1. Parallel processing (Electronic computers)
2. Cm* (Computer system) I. Siewiorek, Daniel P.
II. Segall, Zary. III. Title.
QA76.5.G365 1987 004'.35 86-29119
ISBN 0-932376-91-6

To the Cm* Family,
and our newest members,
Gail and Nathaniel.

Contents

III. Programming Environments

IV. Experiments

Foreword

This book is more than just a report of an interesting system. It includes the rationale for and description of a large multiprocessor utilizing two operating system approaches, a comparison of the two operating systems' intercommunication methods, and methodology and measurement techniques. The book also deals with understanding and predicting speedups for parallelism, comparing these experimentally on 30 different problems, and positing a general approach for parallel decomposition.

The Cm* architecture was described in the fall of 1973. It was designed to explore how microcomputers could be combined to form large digital systems. As the sixteen-processor multiprocessor C.mmp became operational at Carnegie-Mellon University in the early 1970s and Digital Equipment Corporation introduced the LSI-11 (January 1975), the project became focused on building a modularly expandable multiprocessor for parallel processing research. The architecture was specified by March 1975, and the design was completed by the fall of 1975. A single-cluster system was operational by July 1976. A ten-processor, three-cluster system and operating system were demonstrated in June 1978. All fifty processors in five clusters were operational by September 1979. The machine was used experimentally for applications until January 1986.

The project output took many forms:

Twenty-seven papers were critically reviewed.

Twenty-four Ph.D. dissertations and fourteen master's theses (five of whose authors completed Ph.D. degrees on other topics) were finished.

Eight faculty (three disciplines outside computing), ten research faculty, and ten staff members participated over the years.

Cm* influenced at least four subsequent commercial or research multiprocessors.

At least four of the Cm* alumni went on to design and produce major innovative and useful computer systems (not assemblages of mundane boards with microprocessors that characterize too much of today's information processing industry). I don't think these people would have built exciting systems without the Cm* experience.

Dan Siewiorek estimates that more than 100 man-years went into the project, costing around $5 million (or about $500,000 per year), including the $500,000 from Digital Equipment Corporation for the LSI-11s and another $500,000 for special hardware. A 1985 report estimated that the federal government spent $100 million during the previous year on experimental computer research; thus we should be seeing twenty books like this per year. (Large companies spend an equal amount on open systems research.) Since I know of only a few works of this high a quality over the past decade, I think the book is required study for the computer and computing research communities.

The Cm* programming environment was a critical part of getting the work done. In general, multiprogramming use probably is required to involve enough computer users in applications search. What were the weaknesses of this experiment? Since most applications that occur within the engineering and scientific domains require floating-point arithmetic, an experimental machine must be able to deliver a substantial amount of floating-point computer power to attract real users. Since the floating-point times for the LSI-11 were about 100 microseconds, even fifty computers would not be especially useful in obtaining real results. Experiments had to be run with either integers or "simulated" fast floating-point times to understand the speedup on what a "real" machine would deliver. Thus, while the slow floating-point times made the machine less useful and attractive to a large user base, it did not detract from the experiments because the researchers were dedicated to exploring parallelism. A machine with faster floating-point operation, however, would have attracted more users.

The book should be useful for people who are trying to use multicomputers (such as hypercube, grid, and tree-connected computers, along with workstation and PC clusters) for production or for research on parallelism because Cm* emulates these structures. I believe that none of the message-passing multicomputers can achieve the speedups experienced on Cm* unless a new methodology is found, simply because the message-passing mechanisms used for data access and program synchronization are relatively slow. Similarly, unless these machines (like Cm*) have the proper arithmetic and memory capability, they will be only scale models, lacking real utility and user pull. Therefore, if multicomputers are to do useful parallel processing (as opposed to attracting money for research on parallelism), only very coarse grain parallelism problems are candidates for use. These problems include design-rule checking, monte carlo simulation, and ray tracing (Apollo made a movie, *A Long Ray's Journey Into Light,* using 200 workstations to generate the images). They can all be partitioned into *N* independent cases and run with a single program and common data on all the computers, where the results are combined at the end of the calculation when all cases have been finished. An idle collection of high-performance workstations is an ideal structure to use for coarse grain parallelism. Even though Cm* was built as a "scale model" computer, the results indicate that shared-memory multiprocessors (with at least ten processors) will work efficiently for problems of medium-grain parallelism (such as linear algebra, finite-element analysis, and simulation), provided the programming tools and environment are good.

The reader should understand my biases and involvement: The original work on computer modules came out of my research at Carnegie-Mellon University prior to 1972; I helped on the Cm* architecture, critiqued the design, and generally have encouraged the direction the work has taken (including this book). Furthermore, I have been involved in building at least ten multiprocessors over the past two decades and believe that "production" parallelism can be achieved within the decade using multiprocessors. I also believe computers for parallelism are nearly as important as the development of the computer (circa 1950). A complete change in theory, algorithms, programming languages and environments, problem decomposition, and user training is likely to be needed to achieve "production" parallelism. Now is the time to start working on parallelism by understanding and building on the solid results of Cm*.

Gordon Bell 16 September 1986

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency under order no. 3597, monitored by the Air Force Avionics Laboratory under Contracts F33615-78-C-1551, F33615-81-K-1539, and F33615-84-K-1520; by the National Science Foundation under grants SPI-7821198, MCS-81-20270, MCS-81-03044, MCS-79-20698, and DCI-86-02143; by the Ballistic Missile Defense Technological Center under contracts DASG-60-80-C-0057, DASG60-81-0077, and DASG60-86-C-0015; by NASA Langley Research Center under contract number NAG-1-190; by RCA under contract 904822-0005-21; by Office of Naval Research contract number N00014-85-K-0008; and by the Digital Equipment Corporation. Ivor Durham received support from an IBM fellowship, Roger Dannenberg and Steven Vegdahl were supported by the John and Fannie Hertz Foundation, and John Ousterhout and Richard Snodgrass held National Science Foundation fellowships.

The Carnegie-Mellon University Design Research Center and the Electric Power Research Institute provided support for some of the experiments carried out on Cm*.

Parallel Processing

The Cm* Experience

1. Why Multiprocessors?

In the past two decades, advances in integrated-circuit (IC) technology have brought about an exponential increase in logic functions per unit cost. The power of the largest computers of the mid-sixties can today be packed onto a single silicon chip called a microprocessor. On the horizon, fundamental physical limits (such as the speed of light and atomic dimensions) loom as barriers that circuit designers will reach by the year 2000, capping the performance attainable by a uniprocessor. Yet the demand for computational cycles continues to grow unabated as large scientific problems and commercial services push back the frontiers of computer applications. The synergy of high-performance microprocessors, the approaching limit on uniprocessor performance, and the growing demand for computation has stimulated research into computer organizations utilizing a large number of processors to achieve either higher absolute performance or high performance at a lower cost than was previously possible.

There are many reasons for choosing multiprocessor structures. A partial list might include [Siewiorek *et al.* 82]:

Peak computing power. The entire system can be devoted to a single problem. A multiprocessor system can solve problems with higher or more frequent interprocessor communication than a network because the interprocessor-communication bandwidth is higher.

Performance / cost. Advanced technology has produced low-cost processors whose instruction / second / dollar ratio is 10 to 100 times better than that of large, high-speed processors. Even though these low-cost processors have minimal functionality (e.g., simple instruction set, limited data types), there are special applications for which they are adequate.

Availability and graceful degradation. Multiprocessor systems can be designed with no central, critical component. Thus failures can be configured out of a system for only an incremental loss in computing power. Multiprocessors are more cost-effective than uniprocessors with respect to the relative cost of redundancy. A uniprocessor system requires redundant hardware for failure detection, diagnosis, and recovery. A multiprocessor need only have hardware for failure detection, relying on the unaffected processors to perform the diagnosis and recovery in software.

Modular growth. Systems can be designed to allow processors, memories, and input / output subsystems to be added incrementally.Thus multiprocessors can be tailored to individual applications or grow incrementally to meet demand.

Functional specialization. Functionally specialized processors can be added to improve performance for particular applications.

If parallel processing is so attractive, we may then ask why it is still considered an "exotic" means of computation. The answer to this question is complex but depends basically on how well we know how to apply parallel processing to practical problems. The computing profession must learn how to develop parallel algorithms, how to program a parallel machine, how to decompose a problem systematically into parallel parts, how to determine the optimal size of these parts, and how to specify the right architecture and operating system to support a given grain of parallelism.

The complex space of parallel processing poses a set of heavy scientific challenges. One of the main challenges is performance—realizing concrete benefits out of the potential performance of a parallel processor. Although the machine boasts a certain amount of raw processing power (measured in MIPS or MFLOPS,* for example), the application running on the machine may be able to extract only a part of it. A number of factors may contribute to suboptimal performance: the algorithm and its implementation, predictable hardware operating-system overhead, and the added complexity of parallel programming. Given the multiple facets of this problem, the need for an experimental parallel machine becomes obvious.

This book describes the scientific insights gained in building, programming, and evaluating the Cm*, an experimental 50-processor multiprocessor system. Conceived in the early 1970s at the beginning of the microprocessor revolution, Cm* became operational in 1977. It was one of the first large-scale general-purpose multiple-instruction / multiple-data (MIMD) processors. Two complete operating systems—StarOS and Medusa—were developed along with a host of applications.

This book reports on a decade of experience representing more than 100 man-years of effort in the newly emerging area of parallel processing. It constitutes a concise description of perhaps the most comprehensive parallel-processing research devoted to a single architecture. Rather than emphasize the particular hardware / software structure of the machine, we have chosen to concentrate on the wealth of issues explored via Cm* experiments. These issues range from programming parallel applications, to investigating interactions between the operating system and architecture, to developing an automated experimentation environment that facilitates the construction of prototype applications.

One can easily argue that most of today's large parallel processors serve primarily as research vehicles for the study of parallel processing. Cm* is undoubtedly the most mature parallel system in terms of experimental results and currently is the best equipped for performing experiments. We hope that this book will make a significant contribution to the understanding of parallel processing and will inspire both designers of new parallel systems and students of parallel processing.

1.1. What Is a Multiprocessor?

It is suitable at this point to place Cm* in the space of parallel computer architectures. To do so, we will briefly explore, without claims to completeness, several dimensions related to the classification of distributed systems. Six main classes of parallel processors can be identified as follows:

* Millions of Instructions per Second; Millions of Floating Point Operations per Second.

Simple uniprocessor computer: Single-instruction, single-data. The processor interprets a single instruction stream to operate on data stored in a single memory. Some parallelism can be achieved through the use of a fetch / decode / execute pipeline.

Pipelined multiple-execution-unit uniprocessor for scalar and vector processing: Single- and multiple-statement execution architectures. The processor interprets a single machine instruction to operate, in parallel, on multiple data (e.g., several numbers or a vector) stored in a single memory. In addition, several instructions from the stream also may execute at one time.

Lockstep processor: One instruction-execution unit, many processing-element / data-memory pairs. A single machine instruction controls the simultaneous execution in a large number (greater than 100) of processing elements on a lockstep basis. Each processing element has an associated data memory. For this reason, the organization is often called a single-instruction / multiple-data (SIMD) architecture. Intercommunication is via fixed paths among the processing elements.

Multiprocessor: Many instruction-execution units operating on many data memories. Each simple uniprocessor accesses programs and data stored in shared memory. Cm* is an example of a multiprocessor.

Multicomputer: Many instruction-execution units, each with a dedicated data memory. Each primitive element is a computer (processor-memory pair). Communication among computers is either via fixed paths or via some message-switching mechanism.

Dataflow architecture: Many instruction-execution units, which are activated upon receipt of data. The order of execution depends on when data is received by the execution units, not on the order in which instructions appear in the source program.

A pipelined uniprocessor can overlap processing of several instructions, attaining a degree of parallelism of from two to four. The multiple-execution-unit uniprocessor increases parallelism to the order of four to ten by simultaneously operating on several instructions from the instruction stream with its different execution units. The lockstep processor uses a simple instruction decoder to process many data items in parallel. Typically, the processing elements are 1 bit wide and together can operate on bits across 100 or more words at the same time.

Multiprocessors consist of many autonomous processors that address the same primary memory. In contrast, multicomputers are made up of autonomous computers communicating by means of messages through static or dynamic communication links. Since a multiprocessor can emulate a multicomputer with a message system implemented in shared memory, the multiprocessor is a more general structure for a distributed computer research laboratory. In particular, the Cm* architecture provides for a continuum of memory sharing between processors ranging from no shared memory (multicomputer) to fully shared memory (multiprocessor).

Another way to define the place of multiprocessor systems in the world of distributed computers is to consider the synchronization *granularity* [Mohan *et al.* 85], or frequency of synchronization between tasks in a distributed system. Table 1-1 shows, for a variety of synchronization granularities, the best-suited distributed computer organization. Multiprocessor systems such as Cm* are most suitable for a medium grain of synchronization.

Multiprocessors and networks employ a variety of interconnection structures to couple processors with memory units. Among these are full interconnections, shared buses, multiple shared buses, crossbar switches, and multistage networks. See [Siegel 85] for a complete discussion of this topic. The fully connected network couples each processor with each memory through a dedicated link. A shared bus is a single communication path to which both processors and memory are connected. Bus arbitration may either be done in a central arbiter or be distributed among the units.

A multiple shared bus is a set of shared buses connected by gateways or switches. The switches may be organized in many ways. The best-known topologies are hierarchical switches and crossbar switches. With a hierarchical switch, a group of functional units is clustered around a single shared bus, and the clusters are interconnected with another shared bus. By repeating the process, a hierarchically organized multiple-shared-bus network is obtained. If crossbar switches are used instead, they are arranged into a regular pattern. Each processor and each memory unit is connected to a dedicated shared bus. Switches are then used to connect the processor buses and memory buses into the form of a crossbar. Multistage interconnection networks usually require a logarithmic number of levels of switches to connect the processor units with the memory units, while the number of switches in a crossbar is proportional to the product of the number of processors and the number of memory units. Cm* uses a hierarchically organized multiple shared bus.

1.2. Why Experimental Multiprocessors?

At the foundation of the Cm* project stands a set of basic scientific questions. In our case, the questions are related to the nature of parallel computation and its application to real problems. The Cm* multiprocessor was perceived as being the catalyst for bringing together a set of researchers to focus attention on these questions. Hence the Cm* project not only consists of hardware and software but includes the combined effort of these researchers to gain new insights into parallel processing.

Cm*'s hardware and software, as a complex computing system, is structured in terms of layers of abstraction. To run effectively, a parallel application needs both an efficient implementation of each level of abstraction and a well-designed methodology for mapping the application to the machine. Several levels of abstraction are shown in Table 1-2. Each level adds new facilities that hide its implementation details from the view of higher levels. The designer is faced with two questions: (a) how to choose between different approaches to the implementation of each layer, and (b) how to map a parallel application into the various layers, since special hardware or additional microcode may be the most effective way of improving performance.

Table 1-1 Synchronization Granularity and Distributed Computer Structures

Grain size	Synchronization interval (instructions)	Distributed computer structure	Communication overhead (instructions)
Fine	1	Vector / array processor	1
Medium	10–100	Multiprocessor	1–10
Coarse	100–10,000	Multicomputer	100–10,000
Large	10,000–10 million	Network	10,000–10 million

Table 1-2 Levels of Abstraction in Multiprocessor Systems

Level	Sublevel	Typical components
Parallel program	Application software	Processes, tasks, shared data structures
	Executive software	Message system, task scheduler, memory allocator
Multiprocessor	Configuration	Processor, memory interconnection network
	Instruction set	Memory state, processor state, effective address calculation, instruction execution
Hardware	Logic	Gates, flip-flops, registers, sequential machines

To explore such a large space properly, both theoretical and experimental research is critical. This need led to construction of the experimental research vehicle Cm*, with its modular and expandable architecture and a reconfigurable interconnection network.

Operating systems are a major thrust of Cm* research. Two conceptually different operating systems have been implemented and evaluated. Large parts of the operating-system kernels have been vertically migrated into firmware. One can solidly argue that a research laboratory is as good as the quality of its support environment. For its day, Cm* has had one of the most sophisticated and complete instrumentation systems, greatly facilitating complex experiment design, implementation, and measurement.

1.3. Stages in the Life of a Multiprocessor Laboratory

Cm* research has progressed through three stages, reflecting the evolution of hardware, operating systems, and the experiment support environment. Each stage has been associated with a particular experimental methodology and a specific set of experiments. We believe that these stages are not particular to Cm* but that they track the evolution of a typical multiprocessing laboratory.

STAGE 1—STANDALONE. The system is completed through the instruction-set level of abstraction; that is, the instruction set has been defined, and the hardware has been built. There is virtually no software to support user applications. The only software utility is a loader, which allows programs compiled on another machine to be loaded into the experimental machine. Experiments are limited to simple, regular, compute-bound algorithms. Only a limited number of parameters may be varied, and only by editing the source code of the benchmark. The programmer must be a hardware expert because there is little software to provide a higher-level virtual machine. The program is tied closely to the hardware: The user must specify where code is to be placed, define the memory map, and write code to initialize memory, create processes, manage resources, and collect data.

For Cm*, the experiments in stage 1 included the following:

Hardware saturation. Programs consisted of a few instruction loops with varying placement of code and data. The capacity of various hardware resources was determined, along with the impact of contention for those resources.

Speed variations due to changes in algorithms or data. Experiments sought the impact of synchronization of access to data and of variation due to the amount of data and its placement.

Hardware diagnostics. Diagnostic programs were run for long periods of time. Errors were tabulated, and patterns of errors were studied.

STAGE 2—OPERATING SYSTEM. An operating system is available so that the user can take advantage of its abstractions. The system provides basic functions such as resource management and scheduling. The experimenter calls operating-system primitives in his code and thus needs substantial operating-system expertise. Also characteristic of this phase is the discrete incremental nature of the experimentation process. Experiments tend to be specific rather than comprehensive, owing to the difficulty of making and correlating observations from multiple runs.

In stage 2, the experiments were very regular, with limited variation of parameters; organized in terms of a "master" process that controlled a collection of "slave" processes, which performed the actual computation; heavy users of system calls (since many operating-system facilities were not gracefully integrated into language compilers and utilities).

Typical experiments included the following:

Measurements of the cost of various operating-system functions. Features were tested on a one-by-one basis. Examples include primitives for memory management, interprocess communication, synchronization, scheduling, and exception handling.

Measurements of different implementations of a parallel algorithm. The impact of using various strategies in parallel program organization, data structures, and resource allocation was studied.

STAGE 3—INTEGRATED INSTRUMENTATION ENVIRONMENT. At this staqe, hardware and software facilities have been provided for generating experiments, dynamically observing hardware and software activities, and analyzing results. With this enhanced support, the user can experiment at the application level of abstraction with the capability of varying many parameters. A major characteristic of this stage is the grouping of facilities for experiment generation, monitoring, data collection, and analysis under a single user interface. A richer collection of measurement tools makes it possible to observe the behavior of the operating system and support software with acceptable effort. The programmer may thus be a relative novice to the experimental system.

Stage-3 experiments had the following characteristics: They measured the dynamic behavior of the operating system and applications; measurements were continuous, and programs could be monitored on-line and sometimes in real time; different virtual machines could be studied, as could different logical interconnection structures.

Typical experiments at this stage included the following:

Interaction between algorithm and architecture. An application was run on different operating systems or different simulated architectures.

Designing application-oriented architectures. Based on the results of comparative algorithm / architecture experiments, a virtual machine could be tuned for the application. This virtual machine could be used to provide guidelines for designing an application-oriented architecture.

1.4. Plan of the Book

This book has been divided into four parts, reflecting the different levels of the Cm* hardware, firmware, and software. We have provided extensive cross-references between sections so that the reader can begin with the division that interests him or her most, consulting other chapters for background information where necessary. Here is a brief synopsis of each part.

The Cm hardware* is described in Part I. Chapter 2 recounts the developments that led to Cm* and describes its major components and communication links. Chapter 3 concentrates on the performance and reliability of the hardware.

The Cm* *operating systems* are the subject of Part II. Chapter 4 discusses the influence of the architecture on building operating systems and explains where the two operating systems for Cm* resemble each other and where they differ. MEDUSA, the operating system that most closely reflects the Cm* hardware, is described in Chapter 5. The following chapter focuses on STAROS, the operating system that endeavors to provide more general support for multiprocessing. Chapter 7 reports on measurements that compare the performance of various components of the two systems.

Part III contains a discussion of the various *programming environments* implemented on top of the two operating systems. These can be classified into languages, run-time support, and experimentation environments. Parallel languages are the topic of Chapter 8. It describes three languages with very different implications for parallel-program development. Chapter 9 details three run-time environments that were designed to support languages and experimentation. Early experience revealed the need for a comprehensive way of generating and cataloging experiments, and hence an *integrated instrumentation environment* was developed. It is described in Chapter 10.

The *experiments* themselves are the focus of Part IV. These can be loosely divided into two categories: experiments with parallel algorithms and experiments with parallel architectures. The former are described in Chapter 11, which distills general lessons in parallel algorithm performance from the Cm* experience. Several experiments used Cm* to emulate a large class of multiprocessor structures. Their conclusions are presented in Chapter 12. An interesting adjunct to Part IV is Appendix A, which describes all the parallel algorithms implemented on Cm*.

As one of the first multiprocessors to support extensive research, Cm* has yielded results that will influence future developments in parallel processing in many ways. This book affords an opportunity to obtain a broad overview of a multiprocessor laboratory or to delve deeply into some of its aspects.

I. The Cm* Hardware

2. The Cm* Hardware Architecture

Historically, Cm* had its beginning in the *register-transfer module* (RTM) project [Bell *et al.* 72]. RTMs are a module set for the systematic construction of digital systems at the register-transfer level. The successor to the RTM project, and immediate forebear of Cm*, was the *computer module* (CM) project [Bell *et al.* 72, Fuller *et al.* 73], which also had the systematic construction of digital systems as its major objective.

The prime assumption of the CM project was that a simple computer, a processor-memory pair, is an appropriate module for building large digital systems. A computer is any general-purpose device that, within performance constraints, can perform any well-defined digital function. A computer also is well suited for interfacing devices such as sensors, actuators, storage media, and communication devices. Modular structures have several advantages, including reduced cost through faster system design, faster production, reduced inventories, and simplified maintenance.

A second basic assumption of the CM project was that communication between modules should be at the level of a single memory reference. Both lower and higher levels of communication were rejected. An example of a level of communication more fundamental than memory references would be a network of control lines that could be set and sensed by individual processors. Synchronization might be achieved by use of a central clock and data paths via communication registers. This very low level intermodule communication was rejected because it would too tightly constrain the use of the system. The need to synchronize modules to a centralized clock would limit the physical size of the structure and severely impair the ability to build fault-tolerant systems.

Alternatively, an example of a higher level of communication would be message passing between computers. This suggests a computer network. Even with very high bandwidth interconnections, the grain size of effective cooperation between computers is limited by the overhead of message preparation, reception, and activation of the designated recipient process. This makes it inefficient to run parallel algorithms such as global search or Ada rendezvous (see Section A.15), which require frequent references to nonlocal memory.

Contemporary with the RTM and CM projects was the multiprocessor system C.mmp [Wulf 72]. C.mmp sought large amounts of computing power at a lower cost through the use of minicomputers. Experience with the development and operation of C.mmp and Hydra [Wulf *et al.* 74], the C.mmp operating system, influenced both the architecture and the initial operating system of Cm*.

2.1. Evolution of the Cm* Design

The structure of Cm* has developed over a number of years into the "canonical" structure of Figure 2-1. This is a structure with a low concurrency switch (the network of buses) giving access to shared memory. The structure is built from processor-memory pairs called *computer modules* or *Cm's*.[1] The memory local to a processor is also the shared memory in the system. Inherent in this structure is the assumption of program locality. The efficient use of the system depends on ensuring that most of the code and data referenced by a processor will be held local to that processor. Early CM* measurements with various benchmark applications indicate that local-memory hit ratios of 0.8 to 0.9 (i.e., a fraction of total memory references directed to local rather than remote shared memory) were readily achieved (Sections A.1 through A.3).

A series of design studies was undertaken to explore this design space. Figures 2-1 to 2-3 depict the PMS (processor-memory-switch) structures studied, while Table 2-1 lists the estimated cost and complexity of each design [Fuller *et al.* 75]. Table 2-1 was created during a preliminary design exercise, so the numbers are only approximate figures.

Initially, one self-contained module was envisioned that consisted of a processor, memory, and an intelligent interface (Figure 2-1). The result was termed a computer module (Cm). The address-mapping controller (*Kmap*, marked "K" in the figures) performed all the functions necessary for generating external memory requests and responding to external requests for its local memory. So that the capacity for interprocessor communication would not be limited by any single communication path, each Kmap connects to two inter-Cm buses. Memory could be shared even though there was no direct visible connection between the requesting processors and requested memory. For example, consider a request by *P1* to *M4* in Figure 2-1. *Kmap1* would route the request to *Kmap2*, which in turn would route it to *Kmap4*. From Table 2-1 we see that the design was extremely costly, while a simulation / benchmark study [Levy 74] indicated that the bus structure was underutilized. Subsequently, as simple a design as possible was tried to minimize the complexity. Figure 2-2 depicts the simple interface (S.minimal, marked "S" in the figure) design. The minimal interface provides parallel word transfer between the two buses. Some modules were connected by physical links that permitted direct communication. Modules without direct physical links could still communicate via intermediate modules, provided that the delays for the intermediate passing of requests were acceptable. Table 2-1 indicates that the projected performance was low and that for fully interconnected structures, the cost was comparable to the Kmap-per-Cm scheme (Figure 2-1).

Further investigation led to the conclusion that very little performance loss resulted from centralizing the address-mapping and multiple-bus connection functions of individual modules in a Kmap that is shared by a number of computer

[1] After the PMS notation of Bell and Newell [Bell and Newell 71].

Figure 2-1 The Canonical Structure of Cm*

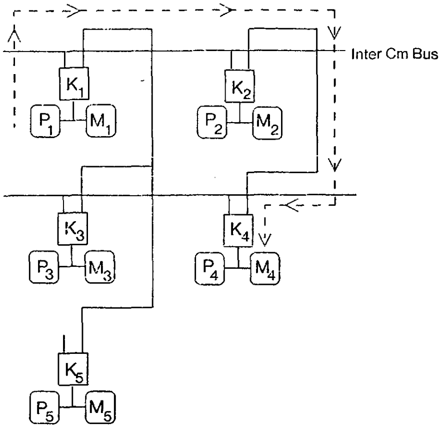
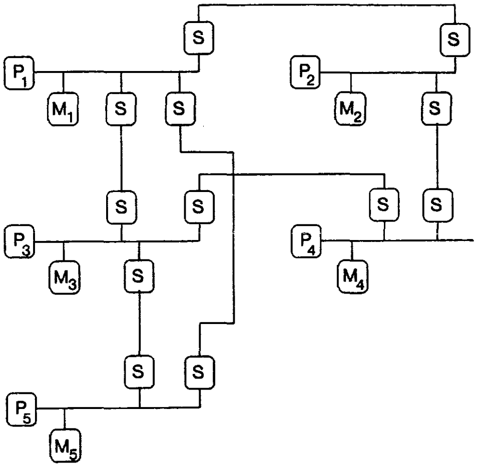


Table 2-1 Comparison of Three Alternative Cm* Implementations

Design	Chips/Port	External connections per Cm	Performance degradation factor for a nonlocal memory read operation with one level of mapping	Representative fully connected system: 10 Cm's	Representative fully connected system: 100 Cm's	Representative fully connected system: Total chips (10 Cm's)
Kmap/Cm (Fig. 2-1)	380	25 to 75	1.5	10 Kmaps	100 Kmaps	3,800
Simple links (Fig. 2-2)	25	24 to 240	8	180 links	19,800 links	4,500
Kmap/cluster (Fig. 2-3)	125 (Slocal) 650 (centralized Kmap)	26 (26 to 78 per Kmap)	1.8	10 Slocals 1 Kmap	100 Slocals 10 Kmaps	1,900

Figure 2-2 Simple Interface between Cm and Bus



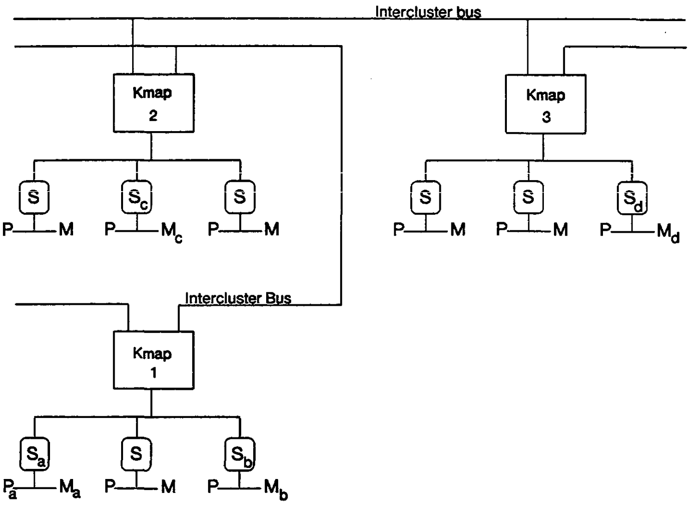
modules (Figure 2-3). The cost savings are quite dramatic. Table 2-1 shows a savings of a factor of two in chip count for comparable structures. Actually, the cost savings are even better than indicated by Table 2-1 because the final shared Kmap design incorporated many features not accounted for in the chip counts for the other designs (for example, 20K bytes of bipolar RAM for microcode and data storage). The programmable high-performance Kmap is shared by several Cm's connected to an inter-Cm bus via simple interfaces (Slocals). The basic function of the Slocal is to provide a buffer between the processor and the inter-Cm bus and to provide sufficient control functions to generate or respond to external memory requests.

The major conclusion of this study is that the architectural design of Cm* was driven by the goal of balancing an expandable canonical parallel architecture with the price/performance characteristics of available components, such as LSI-11 processors. The clustering approach—sharing buses and Kmaps between a number of Cm's—is mainly due to the price/performance/functionality trade-off. This process is inherent not only in the design of Cm* but in any parallel-processor design.

2.2. A Hierarchical Multiprocessor: The Structure of Cm*

This section provides a brief overview of the final structure and components of the Cm* hardware. More detailed descriptions may be found in a number of earlier publications. The original description of the design and implementations of Cm* appears in two papers presented at the 1977 National Computer Conference [Fuller *et*

Figure 2-3 Sharing a Kmap among Computer Modules

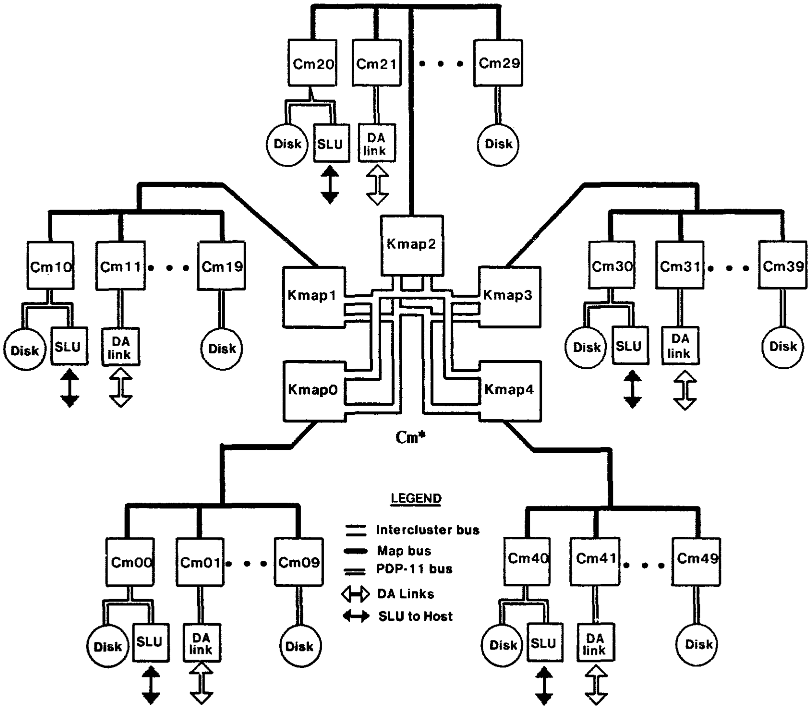


al. 78, Swan *et al.* 77]; the design and switching structure of Cm* and a detailed account of one particular addressing structure are described by Swan [Swan 78].

Cm* consists of 50 Cm's, connected together by a hierarchical, distributed switching structure as depicted in Figure 2-4. The lowest level of the switching hierarchy consists of Slocals, local switches that connect individual Cm's to the rest of the structure. Cm's are grouped together into *clusters* that are presided over by high-speed microprogrammable communication controllers called Kmaps. A Kmap provides the mechanism for Cm's in its cluster to communicate with each other and cooperates with other Kmaps to service requests from its Cm's to access Cm's in nonlocal clusters. Since Kmaps are microprogrammable, it is usual to implement key operating-system functions in the microcode of these processors, in addition to the normal function of address mapping.

All communication mediated by the Kmaps is implemented via packet switching rather than circuit switching, to avoid deadlock over dedicated switching paths. Packet-switched communication also allows the processing of requests by the Kmaps to be overlapped, since switching paths are no longer allocated for the duration of a request. This leads to considerably better utilization of the switching structure. The interconnection structure of Cm* at the level of clusters is essentially arbitrary. The Kmap of each cluster has two bidirectional ports, each of which may be connected to a separate *intercluster bus* to implement a variety of interconnection schemes. In the usual hardware configuration, all five Kmaps are connected to both intercluster buses, as shown in Figure 2-4.

Figure 2-4 The Structure of Cm*



2.2.1. Cm's and Slocals

Each Cm is a processor-memory-switch combination, consisting of a standard off-the-shelf Digital LSI-11 processor, 64 or 128K bytes of memory, one or more I/O devices, and a custom-designed Slocal that connects the processor-memory combination to the rest of the system (Figure 2-5). When the processor of a Cm initiates a memory reference, the Slocal of that Cm is responsible for determining whether the reference is to be directed to local memory or out to the Kmap for further mapping. As shown in Figure 2-6, the Slocal uses the 4 high-order bits of the processor's address, along with the current address space, to access a mapping table that determines whether the reference is to proceed locally or not. (A more complete discussion of how address translation is performed by the Cm* operating systems may be found in Section 4.2.) References that map to local memory proceed with no loss of performance; references that map to another Cm in the same cluster as the referencing Cm are slower by a factor of three; and references that map to a Cm in another cluster are slowed again by a factor of three. These figures are the best possible ratios that can be achieved on Cm* and therefore correspond to constraints

Figure 2-5 Details of a Computer Module

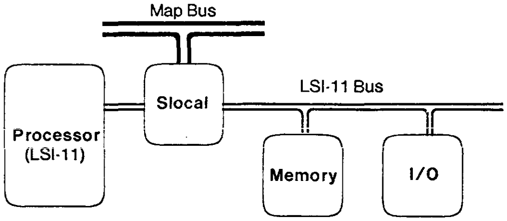
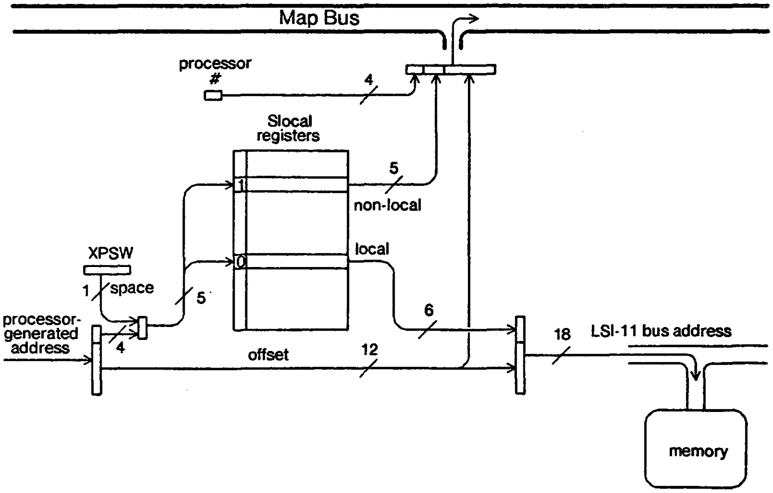


Figure 2-6 Address Mapping in the Slocal



imposed by the hardware itself rather than to any microcode implementation quirks. All I/O devices in Cm* are connected to the various LSI-11 buses. Since there is no interprocessor communication mechanism other than the standard one for memory references, interrupts generated by an I/O device must be fielded by the processor to which the device is directly attached.

2.2.2. *Kmaps: Transaction Controllers*

Standing as the switching center within a cluster of computer modules, and as a node on a network of clusters, the *Kmap* is the primary source of synchronization and of communication in Cm*. A fast (157-ns. cycle), horizontally microprogrammed (80-bit wide) microprocessor, the Kmap provides the basic address mapping, communication, and synchronization functions in the system. The Kmap itself consists of three tightly coupled processors (Figure 2-7). The bus controller, or *Kbus*, acts as the arbitrator for the bus that connects Cm's in the local cluster to their Kmap; the *Linc* manages communication to and from the Kmap to other Kmaps; and the mapping processor, or *Pmap*, responds to requests from the Kbus and Linc, and performs most of the actual computation for a request for service. The Pmap also directs the Kbus and Linc to perform any needed operations on behalf of the request being processed.

The Kmap is envisioned as a transaction controller, sending to and receiving from computer modules and other Kmaps message packets that contain requests and replies, following a protocol designed by the microprogrammer. Because one of the common transactions the Kmap is expected to handle is the mapping of a memory access issued by the processor of one computer module to a location in the memory of another computer module—a transaction that must be performed very rapidly and with little delay if any reasonable system performance is to be obtained—the Kmap contains some special hardware features designed to assist it in controlling many transactions at a high rate of speed.

The Kmap hardware supports eight separate Pmap processes, known as *contexts*, each with its own set of general-purpose registers and its own microsubroutine stack. Typically, each context is in charge of one transaction. When one context needs to wait for a message packet to return with the reply to some lower-level request, it has the Pmap switch to another context so that work on another transaction can proceed concurrently.

The Kmap also contains 5,120 words of random-access memory, called the *data RAM*, which the Pmap can read and write at the expense of a few microinstructions. Because the data RAM is shared by all contexts, it typically holds information that is of interest to more than one transaction, such as cached pieces of address translation tables and mechanisms for synchronizing the use of other resources among different contexts.

Taken individually, each Kmap appears to the Pmap microprogrammer as a nonpreemptive, hardware-scheduled multiprogramming system (since contexts are never interrupted, but are suspended only when the microprogrammer directs). Taken collectively, the network of all Kmaps presents the microprogrammer with a distributed system based on message-packet intercommunication.

2.2.3. The Interface between Kmap and Computer Module

The Pmap communicates with the computer modules in its cluster via the *map bus*, a packet-switched bus controlled by the *Kbus*. The Kbus fields requests and replies from computer modules, coordinates the transfer of data across the map bus between computer modules or between a computer module and a Pmap context, and keeps track of which Pmap contexts are free to service new requests. Two queues, the Kbus *out queue* and the Pmap *run queue*, provide the interface between the Kbus and the Pmap. Refer to Figure 2-7.

A request by a computer module to the Kmap is said to invoke some *Kmap operation*. The most basic Kmap operation, and certainly one that is expected to be invoked frequently, is the mapping of the nonlocal access to a location in the physical memory of some computer module in the cluster. Figure 2-8 traces such an *intracluster memory reference*, step by step. Except where noted, the description also applies to other Kmap operations. Because it is so common, a mapped nonlocal memory access usually is just called a *mapped reference*. The intracluster access shown in Figure 2-8 is one example. Each memory access issued by the processor of a computer module passes through its Slocal, which either routes the access directly to local memory or sends it out to the Kmap. A memory access handled by the Kmap also can be mapped back to the local memory of the issuing processor, but a direct local access is about three times faster.

Whenever the processor of a computer module issues a nonlocal memory access (1), a *service request* is signaled to the Kbus. The Kbus allocates a Pmap context, reads the virtual address for the memory access via the map bus (2), and activates

Figure 2-7 The Components of the Kmap

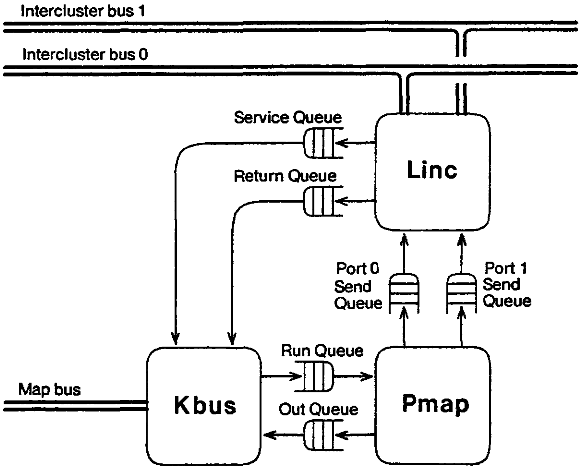
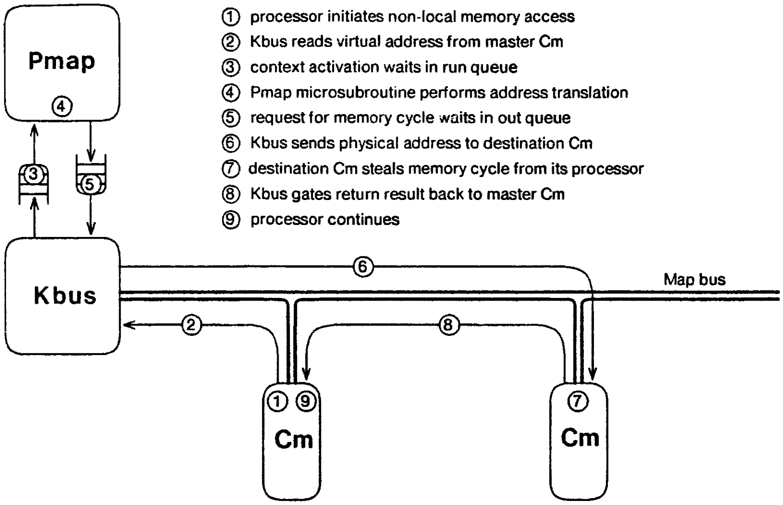


Figure 2-8 The Steps in an Intracluster Memory Access



① processor initiates non-local memory access
② Kbus reads virtual address from master Cm
③ context activation waits in run queue
④ Pmap microsubroutine performs address translation
⑤ request for memory cycle waits in out queue
⑥ Kbus sends physical address to destination Cm
⑦ destination Cm steals memory cycle from its processor
⑧ Kbus gates return result back to master Cm
⑨ processor continues

(or reactivates) the new context by placing an entry in the Pmap *run queue*. This entry contains the number of the Pmap context to be activated, along with a small amount of other data, such as the virtual address of the processor's nonlocal memory access or the result data from a Pmap-initiated memory access. The computer module that invokes a Pmap context to process its memory access is called the *master computer module*, or *master Cm*.

A newly activated context does not automatically gain control of the Pmap, however, but instead must wait on the run queue (3) until the Pmap selects it for execution. There are no microinterrupts; the Pmap runs a context until that context explicitly relinquishes it. When this happens, a new context is loaded from the run queue. This event is called a *context swap*, and a context that invokes a context swap is said to *swap out*.

A Pmap context can initiate a physical memory access in any of the computer modules in its cluster. This memory access can be either a *read access*, in which case the result is the data read, or a *write access*, in which case the Pmap also supplies the data to be written and the result serves merely as an acknowledgment. The Pmap context considers the virtual address, the master Cm number, the contents of an address-translation table, and any other desired source of information, and it determines the destination computer module and the physical address within that module to which the memory access is directed (4).

The Pmap then invokes a Kbus operation by loading a request into the Kbus *out queue*. About two dozen different operations are provided, most of which are variations of the Pmap-initiated memory access. After loading a request in the out queue, the Pmap context typically swaps out to let work on some other transaction proceed concurrently with the operation requested of the Kbus. The request waits until the Kbus becomes available (5). At that time, the Kbus carries the memory-access request, via the map bus, to the indicated computer module, called the *destination computer module*, or *destination Cm* (6).

The Kbus has priority for accesses to memory of the destination Cm, so the reference is made via cycle stealing if necessary (7). When the result of the operation becomes available, the Kbus reactivates the requesting Pmap context. This sequence of requesting a Kbus operation, swapping out to run other contexts, and waiting for eventual reactivation is an extremely common occurrence in Pmap microcode. The Pmap context completes processing the service request and signals a *return request* to the Kbus, which gates the result back to the master Cm (8) in a return message. Finally, the master Cm resumes processing (9).

To obtain better performance from a nonlocal memory reference, the Pmap microcode can actually use a special case of the Pmap-initiated memory access —one in which any data to be written is supplied by the master Cm instead of the Pmap context and, provided that no error is indicated, the acknowledgment from the destination Cm is routed directly back to the master Cm, without ever reactivating the Pmap context. If an error occurs, the automatic bypass is forgotten and the result data goes back to the Pmap context, under the expectation that maybe the context could do something intelligent about the problem. In the expected case in which no error occurs, however, a nonlocal, intracluster memory access requires only one Pmap context activation and three map bus cycles. From the time a processor issues a nonlocal memory access until it receives the result data, a total delay of about seven microseconds elapses.

The remote memory access is the simplest and most common Kmap operation but by no means the only one. For more complicated operations, the Pmap microprogrammer generally appropriates some subset of the computer module processor's virtual address space, designating specific addresses the processor can use to invoke other, less-ordinary Kmap operations. Details of the scheme used by the StarOS and Medusa operating systems are provided at the end of Section 4.2. These special Kmap operations are invoked in exactly the same manner as I/O operations are invoked on a computer that, like the PDP-11, has memory-mapped device control registers.

Because each cluster has only one Kmap, the Kmap is the logical processor to perform operations that must be interlocked. If, however, the operation involves a context swap (for example, any operation that performs a Pmap-initiated memory access), some mechanism must be used inside the Kmap to prevent other contexts of the same Kmap from violating the interlock. One technique is to implement an internal semaphore in the data RAM.

Other operations that the Kmap might provide are low-level operating-system primitives. In particular, operations that cross domain boundaries, such as a message-transfer operation, and operations that involve much memory traffic, such as a block-transfer operation, are well suited for implementation in the Kmap. The primary objective in microcoding these operations is speed.

2.2.4. The Interface between Kmap and Kmap

Kmaps communicate with each other via an *intercluster bus*, a packet-switched bus that is jointly controlled by the Linc processors in each of the directly connected Kmaps. The Linc maintains queues of incoming and outgoing messages, interacts with the Kbus to activate and reactivate Pmap contexts, and provides the local storage for Pmap contexts to construct and inspect intercluster messages. Each Linc is interfaced to two independent intercluster buses, as shown in Figure 2-7.

An intercluster message contains up to eight 16-bit words of data, of which all words except the first are totally uninterpreted by the Linc. Each intercluster message is sent from an immediate *source Kmap* to an immediate *destination Kmap*. The number of the destination Kmap appears in a fixed place in the message so that the Linc can determine which messages are sent to its cluster. Intercluster messages are of two types: *forward messages*, which invoke a new context at the destination Kmap, and *return messages*, which return to a waiting context at the originating Kmap. A return message contains the context number of the to-be-reactivated Pmap context in a fixed place where the Linc can find it in order to inform the Kbus. These intercluster messages are designed to be used as a mechanism for implementing remote procedure calls between Kmaps.

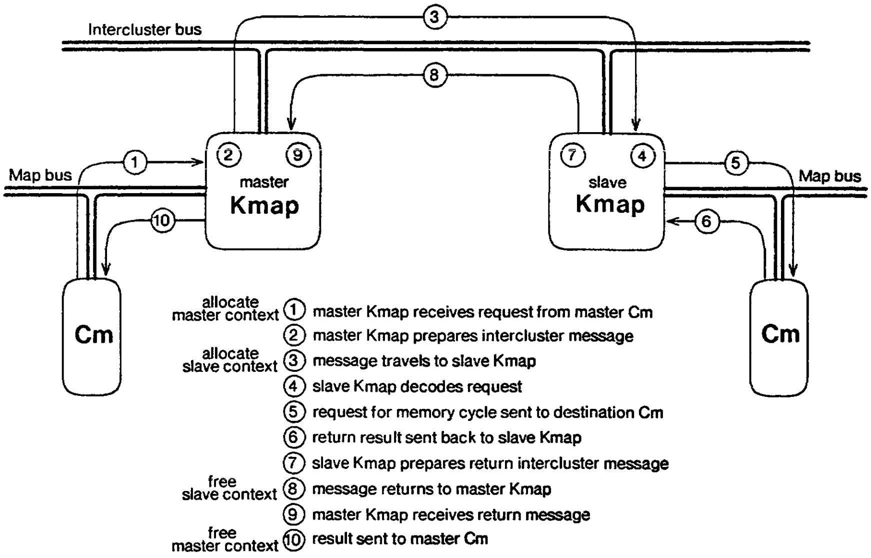
When a Pmap context desires to invoke some operation in another Kmap, it prepares a forward intercluster message, instructs the Linc to transmit it on a specified intercluster bus, and then swaps out. The forward message must include the source Kmap number and the originating Pmap context number so that the remote Kmap will be able to send back a return message. There are standard conventions for this information.

When the Linc receives a forward message, it has the Kbus activate a new Pmap context to examine the message and respond to the request. Presumably, the message contains some operation code that the Pmap context can identify and act upon. After performing the operation, the context prepares a return intercluster message, instructs the Linc to transmit it on a specified intercluster bus, informs the Kbus that the context is now free, and swaps out.

When the Linc receives a return message, it finds the context number indicated in the message and instructs the Kbus to reactivate that context. The context then examines the return message, extracts the result of its requested operation, and continues on with whatever processing it had been doing.

A Simple Multicluster Kmap Operation. From a logical point of view, there is not much difference between the invocation of a Kmap operation by a microsubroutine call from within the Kmap, the invocation of a Kmap operation by a nonlocal memory access from a computer module in its cluster, and the invocation of a Kmap operation by a forward message from another Kmap. Practical considerations abound, but all three are really just different methods of invoking some logical operation. It is often quite convenient for a Kmap, while in the middle of performing one operation, to be allowed to invoke a suboperation that is carried out on a different Kmap. For example, the only physical memory that a Kmap can directly access is the memory that

Figure 2-9 The Steps in a Cross-Cluster Memory Access



allocate master context ① master Kmap receives request from master Cm
② master Kmap prepares intercluster message
allocate slave context ③ message travels to slave Kmap
④ slave Kmap decodes request
⑤ request for memory cycle sent to destination Cm
⑥ return result sent back to slave Kmap
⑦ slave Kmap prepares return intercluster message
free slave context ⑧ message returns to master Kmap
⑨ master Kmap receives return message
free master context ⑩ result sent to master Cm

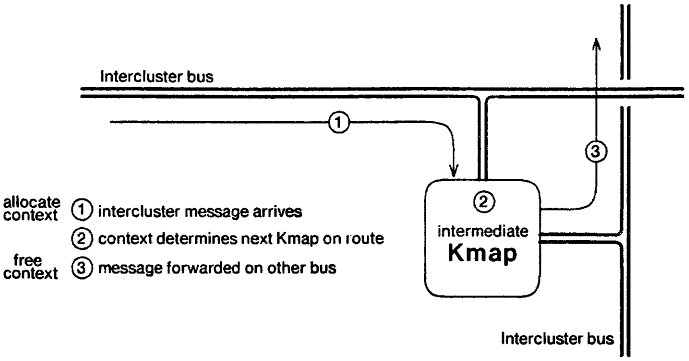
resides in its own cluster; to effect an access on memory in some other cluster, the Kmap must send an intercluster message to that cluster's Kmap, asking it to perform the access and to send back the result. The simplest example of such a multicluster Kmap operation is the mapping of a nonlocal memory access to a location in the physical memory of another cluster (Figure 2-9).

A computer module initiates a nonlocal memory access, which activates a context in its cluster's Kmap. This first-activated context is called the *master context*; the Kmap is called the *master Kmap*. The master context performs the address translation, discovers that some other Kmap will have to perform the access, sends a forward message to that Kmap, and swaps out to await a reply.

When the message arrives at the *slave Kmap*, its Linc signals the Kbus to activate a new context. This context, called the *slave context*, decodes the request, performs the memory access inside its cluster, and sends the result in a return message back to the master Kmap. The return message identifies the waiting master context, which is reactivated to transfer the result back to the master Cm. As far as the master Cm can tell, the only difference between a nonlocal memory access that is mapped to a computer module in another cluster and a nonlocal memory access that is mapped to a computer module in the same cluster is the extra time required for the out-of-cluster operation.

Forwarding Intercluster Messages. In a configuration of the Cm* system, it is quite

Figure 2-10 Forwarding a Message across Intercluster Buses


Intercluster bus
①
③
allocate context
① intercluster message arrives
② context determines next Kmap on route
free context
③ message forwarded on other bus
②
intermediate
Kmap
Intercluster bus

possible that two particular Kmaps have no intercluster bus in common and thus cannot send messages directly to one another. Each Kmap connects directly to two intercluster buses, however, and as long as some path through a series of intermediate Kmaps can be found, the two Kmaps in question can still communicate, providing each of the intermediate Kmaps cooperate by forwarding the message closer to its ultimate destination, as illustrated in Figure 2-10.

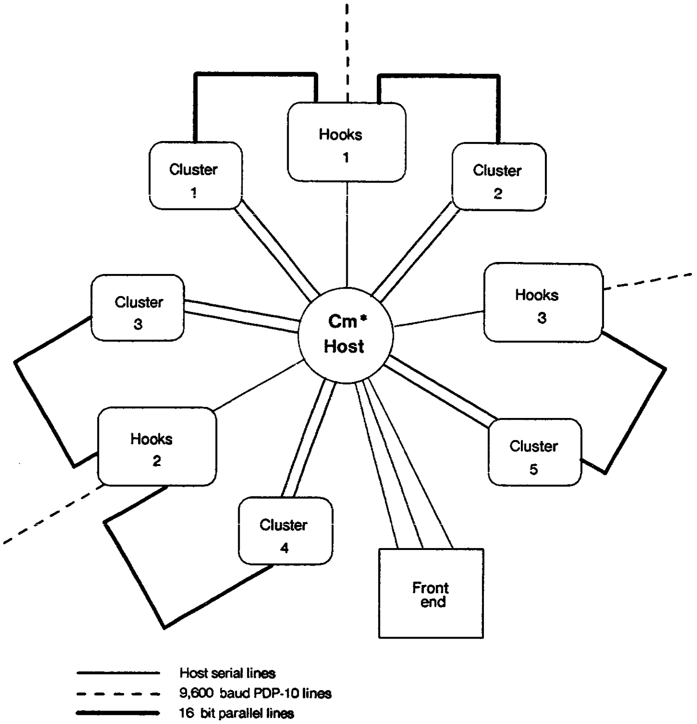
This example differs significantly from the previous example in the way it uses Pmap contexts. In the previous example, contexts are allocated in a nested fashion. The allocation of the master context lasts throughout the entire operation; for a period of time within that allocation, a slave context also is invoked. The potential series of contexts that are allocated at intermediate Kmaps to forward an intercluster message do not wait for a reply but instead accomplish their entire task by passing a message on to another Kmap.

2.3. Communication with Cm*

When software was first being developed on Cm*, there arose a need to let the software developer allocate certain resources (such as clusters and debugging tools) to himself and to protect his resources from being disturbed by other users. The *Cm* Host* system is a serial-line-oriented resource-management facility. It was initially implemented on a PDP-11 / 10 with 28K words of memory.

Several communication lines connect the Host to components of Cm* and, through the Front End, to terminals and to the other Carnegie-Mellon University Computer Science Department computers (Figure 2-11). There are serial-line connections from the Cm* Host to 10 individual Cm's. These connections permit programs or data to be loaded directly from the Host to these Cm's, 2 of which are in each of the 5 clusters. Because the other 40 Cm's lack serial-line connections, they

Figure 2-11 Lines Connected to the Cm* Host



must be loaded from other Cm's via a Kmap or from peripheral devices to which they are attached. The Front End lines have two purposes. First, they allow the user to communicate with other computers from Cm*, as many programs on Cm* interact with programs on other machines. The Front End lines also permit the user to monitor debugging information on both machines simultaneously from one terminal using the *banner* facility explained below.

In addition to its 50 Cm's, Cm* has 3 LSI-11's known as *hooks processors*, which are used for debugging the Kmaps. These processors control the "hooks," which is a collective term for a set of hardware that has been designed into each Kmap to permit complete external control and diagnosis. The hooks consist of several control registers and other hardware within the Kmap, an LSI-11 interface to make the hardware accessible from an LSI-11, and a bidirectional *hooks bus* used to transmit information between the control hardware and the LSI-11 bus interface. The hooks appear to an LSI-11 as a group of 8 words in its physical address space. By reading and writing these words, the hooks processor has almost total control over the internals of the Kmap. It may load microcode; start, stop, and single-cycle the Pmap / Linc and Kbus clocks; read out most and write some of the internal registers of the Pmap; disable certain error checks within the Pmap; and initialize the Kmap.

The *diagnostic processor* (DP) (not shown in Figure 2-11) has been added to the Cm* system to collect hardware reliability and availability information. It hosts a program called the *Auto-Diagnostic Master*, which runs diagnostics on those Cm's that are otherwise idle. The DP is an LSI-11 with 28K words of memory. It has two serial-line connections to the Cm* host. One connection provides a user interface through which one can request status reports about particular Cm's, particular clusters, or the entire Cm* processor. The second serial-line interface is used by the Auto-Diagnostic program as a command interface to the Host. The program logs in over the second line, directs the Host to run diagnostics for it, and on operator request, transfers the statistics file to a PDP-10.

The command structure of the Host resembles that of the PDP-10, which is familiar to most Cm* users. The Host performs several of the functions of a primitive operating system.

Security. The Host protects Cm* from unauthorized use by providing an account system. To use any Cm* resource, one must *log in* to the system by typing in a valid user number and password.

Protection. A resource on Cm* must be assigned to a user before it may be used. Once it is assigned, no other user may access the resource until it is deassigned. The resources of Cm* are thus partitioned among users. Development work on several projects, even several different operating systems, may be carried out simultaneously using disjoint sets of resources (different clusters, for example). Meanwhile, each user is protected from having his working environment accidentally disturbed by another user.

Resource control. The Host has commands for controlling a variety of assigned resources. Among these are commands that load programs from tape or from the PDP-10; commands that start, halt, or single-step a processor; commands that guard against buffer overflow; and commands that communicate with a resource via a direct terminal link.

Communication. The Host allows the user to monitor output from all assigned resources simultaneously by requesting that messages sent to him by each resource be printed at the terminal. As an option, a *banner*—that is, an abbreviation of the name of the resource—may be printed in front of each message to indicate where the terminal output came from. Such a feature helps the user to monitor the interaction of resources for debugging purposes. A user also may send a message to any serial line from the Cm* Host. This option enables the user to communicate simultaneously with several of his or her processes or with other users. This feature is used by the Cm* Auto-Diagnostic system to notify the diagnostic processor of errors detected on other resources.

The facilities provided by the original Cm* Host were limited by a lack of memory and the obsolescent PDP-11/10 processor. Matt Reilly [Reilly 83] designed a new Host to run on a VAX 11/780. It incorporated an improved flow-control strategy to prevent runaway processes from swamping the Host with data and was better able to interact with other programs, such as those written to monitor experiments. It provided a capability for the on-line logging of experimental results, freeing experimenters from the need to write or adapt such code especially for each experiment. The new Host had no Cm*-specific knowledge "wired in," so it could be configured to serve as a testbed for any multiprocessor. The new Host was never implemented, however.

Because the system software developed for Cm* is written, compiled, and stored on a remote machine, object code frequently must be transferred to Cm*. To facilitate high-speed transfers, a *DA Link* was developed to provide a 10-megabaud parallel DMA link between Cm* and a DEC-10. Between an unloaded LSI-11 and an unloaded DEC KL-10, these links can transfer more than 600,000 words per second. A file-transfer system residing on the DEC-10 provides reliable file transfers. The file-transfer system is composed of two parts. The first is a set of low-level reliable transfer routines. These routines are designed to provide error-free transmission of data packets between Cm* and the DEC-10. The higher-level file-transfer routines interface the reliable packet-transfer software to the DEC-10 file system. The file-transfer routines also can multiplex data from several files, allowing a single physical link to become a multiuser system.

The reliable transfer routines transfer all data across the DA Link in packet format. The file-transfer system employs a packet quota mechanism to prevent the receiving site from overflowing its buffer. Acknowledge packets carry both packet-acknowledgment and packet-quota information. File-transfer routines perform the functions of opening and closing files, reading and writing data to and from files, and moving file pointers. To provide a multiuser environment for the DA Link, the file-transfer system implements virtual channels, bidirectional communication paths over which control data and information may pass. The file-transfer system allocates a buffer for each virtual channel and provides buffer-destination information for the data. A file is associated with each virtual channel, and all data placed in the virtual channel's buffer is transferred to this file.

2.4. Summary

The initial proposal for the Cm* architecture was a mesh-organized interconnection of computer modules. Although its structure could be considered "crystalline," it allowed for memory to be shared by different processors with the help of communication and address-mapping controllers. The original proposal was revised according to cost and performance constraints to derive the final Cm* architecture.

The final Cm* architecture consists of 50 Cm's, connected together by a distributed switching structure. The lowest level of the switching hierarchy consists of *Slocals*, local switches that connect individual Cm's to the rest of the structure. Cm's are grouped together into *clusters* that are presided over by high-speed microprogrammable communication controllers called *Kmaps*. A Kmap provides the means for Cm's in its cluster to communicate with each other and cooperates with other Kmaps to service requests for memory references to other clusters. In addition to address mapping, key operating-system functions generally are implemented in microcode.

All communication that involves Kmaps is implemented via packet switching rather than circuit switching to avoid deadlock over dedicated switching paths. Packet-switched communication also allows the processing of requests by the Kmaps to be overlapped, since switching paths need not be allocated for the duration of a request; this results in improved utilization of the switching structure. The interconnection structure between Cm* clusters is essentially arbitrary. The Kmap in each cluster has two bidirectional ports, each of which may be connected to a separate *intercluster bus* to achieve a variety of interconnection schemes. In the implemented configuration, all five Kmaps are connected to two intercluster buses.

Cm* is not a standalone computer system. An additional computer, the Host, is used to load and communicate with Cm*. Communication lines connect the Cm* Host and individual Cm's with parts of the Carnegie-Mellon University (CMU) Computer Science Department computer facilities. A set of special communication lines for loading and debugging connect directly to Kmaps through a number of processors known as hooks. Finally, a diagnostic processor was added to Cm* to collect hardware availability and reliability information. This whole complex comprises the hardware support for the Cm* research laboratory. The next chapter describes hardware measurements undertaken with the support of this laboratory.

Acknowledgments. The introduction to Chapter 2 was adapted from Section 2.3 of [Swan 78]. Section 2.1 was taken from Section 3 of the Cm* paper in the 1975–76 *Computer Science Research Review* [Swan *et al.* 76]. The introduction to Section 2.2 and Section 2.2.1 was written by Pradeep Sindhu for [Jones and Gehringer 80]. Section 2.2 was adapted from material written by Thomas Rodeheffer for [Jones and Gehringer 80]. Section 2.3 was adapted from material originally written by Gregg Lebovitz for [Jones and Gehringer 80].

3. Measurements on the Cm* Hardware

One of the first activities undertaken on a new architecture is to explore the limits of hardware performance. These limits represent the maximum potential of the hardware structure. More complex virtual machines achieved through adding layers of microcode and software reduce the effort required by users at the expense of reduced performance.

We use the term performance in its broadest sense and include not only operations per unit time but also errors per unit time (i.e., reliability). Section 3.1 describes experiments that measured maximum potential throughput, while Section 3.2 presents reliability data.

3.1. Hardware Throughput Studies on Cm*

A number of benchmarks were hand-coded by Levy Raskin to execute directly on the Cm* hardware, with only simple Kmap microcode providing shared memory access. These benchmarks included asynchronous iterative methods for the solution of partial differential equations (PDEs), quicksort, and set-partitioning integer programming. (See Sections A.1, A.2, and A.3 for more details of these applications.)

Measurements were made using both specially designed hardware and standard measuring equipment. Each map bus was attached to a *map bus monitor*, which observes communication on a bus between Cm's and their Kmap. These monitors serve as multiprocessor front panels and are capable of producing trigger signals on selected map bus events. Specially designed logic allows particular addresses or data values passing to and from a given computer module to be displayed selectively and counted. For example, the references to a particular memory page could be counted to determine the reference rate. A standard logic analyzer and counter were connected to the Kmap microinstruction address lines and were used to monitor the Kmap micro-operations and determine what fraction of Kmap time was spent in different operations.

3.1.1. Execution Speed of Computer Modules

It is usually assumed that all Cm's execute at the same speed. To test this assumption, Kong [Kong *et al.* 83] set up an experiment to measure the execution speed of the computer modules in Cm*. This experiment involves timing the execution of a piece of code stored in the local memory of each Cm. Once the program execution begins, the Kmaps are not involved. Figure 3-1 shows that all of the 34 computer modules tested had execution speeds within 4.6 percent of each other. No attempt was made in subsequent experiments to factor out variations in processor speed. Hence these variations became a part of the experimental error.

Figure 3-1 Histogram of Execution Time of Benchmark on 34 Cm's

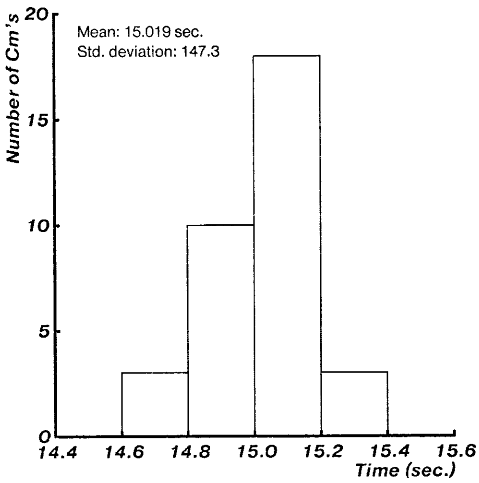


Table 3-1 Single-Processor Performance on Three Different Applications

Application	Reference rate (K words / sec.)	Time between memory references (μs.)	Average number of memory references per instruction	Million instructions per second (MIPS)
PDE	270	3.7	1.45	0.186
Quicksort	300	3.33	3.75	0.171
Integer programming	285	3.51	1.75	0.163

3.1.2. Single-Processor Performance

The three applications (PDE, quicksort, and integer programming) were executed on a single computer module to determine the maximum potential execution rate of the LSI-11 processor. These rates, summarized in Table 3-1, represent the maximum obtainable with no degradation due to the interconnection structure.

3.1.3. Throughput of Mapped Memory References and the Effect of Contention

Three different microcode systems were written for Cm*: Smap (Appendix D), MEDUSA (Section 5.1), and STAROS (Section 6.1). The performance of mapped memory references was measured on each system when it was completely idle except for the operations under test. The three microcode systems are not functionally identical. Smap was written to provide a simple, logically uniform addressing

Table 3-2 Nonlocal Reference Times

Microcode	Intracluster reference time (μs.)	Intercluster reference time (μs.)
Smap	8.3	26.2
MEDUSA	8.3	30.8
STAROS	8.6	35.3

environment along with a few synchronization primitives that could be used in writing parallel programs. Both the MEDUSA and STAROS microcodes were written to support particular operating systems. Consequently, they provide much more powerful and convenient environments than Smap.[1]

Because of the distributed nature of the Cm* hardware, mapped references entirely within a cluster are necessarily faster than mapped references involving another cluster. Table 3-2 shows the elapsed time from the moment a processor initiates a mapped read reference to the moment it receives the result and is able to continue. The times can be compared to those for unmapped references, which can be made at a rate of one every 2 to 2 1/2 μs., depending on the type of physical memory.

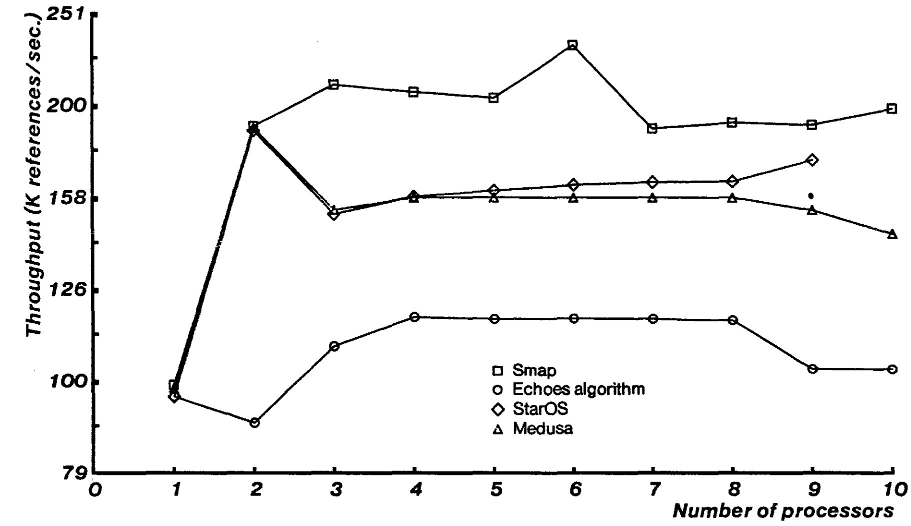
For intracluster references, all three systems add an offset to the base address of a segment to compute a physical address and then perform a Kbus operation to make the memory reference. MEDUSA and STAROS also perform bounds, type, and rights checking to make sure that the reference will not violate protection constraints. Both MEDUSA and STAROS cache addressing information in the Kmap's data RAM. The extra time for a STAROS reference is because it takes two more Kmap microcycles to do the cache lookup.

The cache structures of MEDUSA and STAROS reflect different trade-offs between the speed of memory references and the cost of purging a descriptor from the cache. In MEDUSA's cache structure, if a window points to a descriptor, then the descriptor is guaranteed to be valid (see Section 5.1.2). Thus a mapped reference need not check whether the descriptor pointed to is the correct one. In STAROS, however, a window may point to an incorrect descriptor, so each reference must verify the validity of the descriptor. This check accounts for the extra microcycles in a STAROS mapped reference. MEDUSA avoids the extra cycles by linking together all windows that point to a descriptor so that the windows can be invalidated when the descriptor is purged from the cache. This guarantees that a window is either null or points to the correct descriptor. STAROS does not require such links because a descriptor can be purged without changing the pointers in the windows that refer to it.

Figure 3-2 shows the throughput when all computer modules are making

[1] The STAROS microcode for intercluster references was optimized more carefully than MEDUSA's; see the "implementation issues" at the beginning of Chapter 7.

Figure 3-2 Intracluster Throughput with Slocal Contention



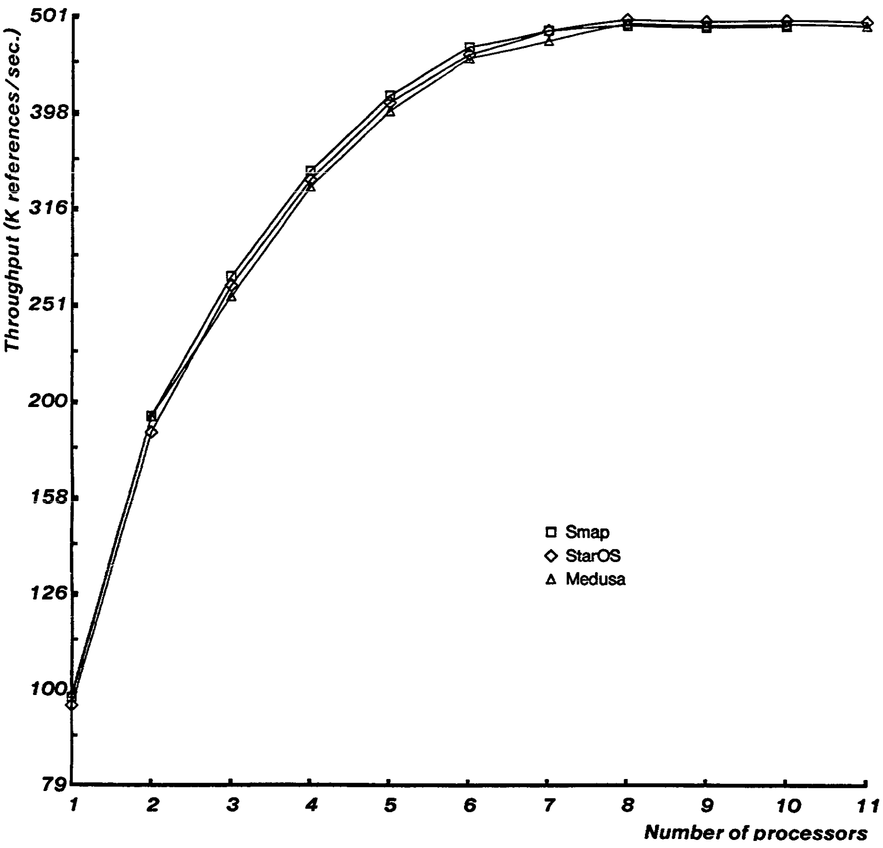
references to the same memory location.[2] As the number of modules making references increases, the Slocal and memory become bottlenecks. For all microcodes, the throughput increases out to about two or three Cm's and then levels off. The maximum throughput supported by Smap (210K references/second) is essentially determined by the hardware because little time is spent on Kmap microcode. The simple algorithm that Smap uses to resolve contention at the destination Cm could, however, cause references to be starved. The maximum throughputs of Medusa (189K refs./sec.) and StarOS (187K refs./sec.) are lower than Smap's, but there is no possibility of starvation because requests are served in a manner that ensures no request will wait forever.

Figure 3-3 depicts the throughput of intracluster memory references on the three systems when each computer module is referencing a different Cm's memory. Each data point was measured with *n* Cm's in a ring configuration; that is, Cm *i* directed all its memory references to the memory of Cm $(i + 1)$ mod *n*. Since there is no contention for the target Cm, the contention-resolution algorithm makes no difference, and the three systems perform almost identically. Figure 3-3 indicates that the Kmap saturates when six or seven Cm's are mapping all their memory references.

In the case of intercluster references, Medusa and StarOS have to do a little

[2] The curve labeled "Echoes algorithm" refers to the Echoes operating-system experiment, described in Section 9.3.

Figure 3-3 Intracluster Throughput without Slocal Contention



more work than Smap at the destination cluster. Smap sends a physical address to the destination Kmap, which then makes the reference; both MEDUSA and STAROS send descriptor names instead. MEDUSA then searches the cache at the destination Kmap to learn the physical address of the location to be referenced. STAROS sends the destination Kmap extra bits in its Linc message that constitute a guess of where the descriptor is in the cache. Though not guaranteed to be accurate, the guess speeds up repetitive references as long as the desired descriptor is still present in the cache. If MEDUSA used this descriptor-guess strategy, its intercluster reference time could also be improved.

Figure 3-4 shows the throughput of intercluster references when all references are made to the same destination cluster but distributed within the cluster so that no Cm saturates. The destination Pmap, therefore, becomes the bottleneck. At saturation, Smap delivers about 210K refs. / sec., and STAROS about 145K refs. / sec.

Figure 3-4 Intercluster References—Saturation of Destination Cluster

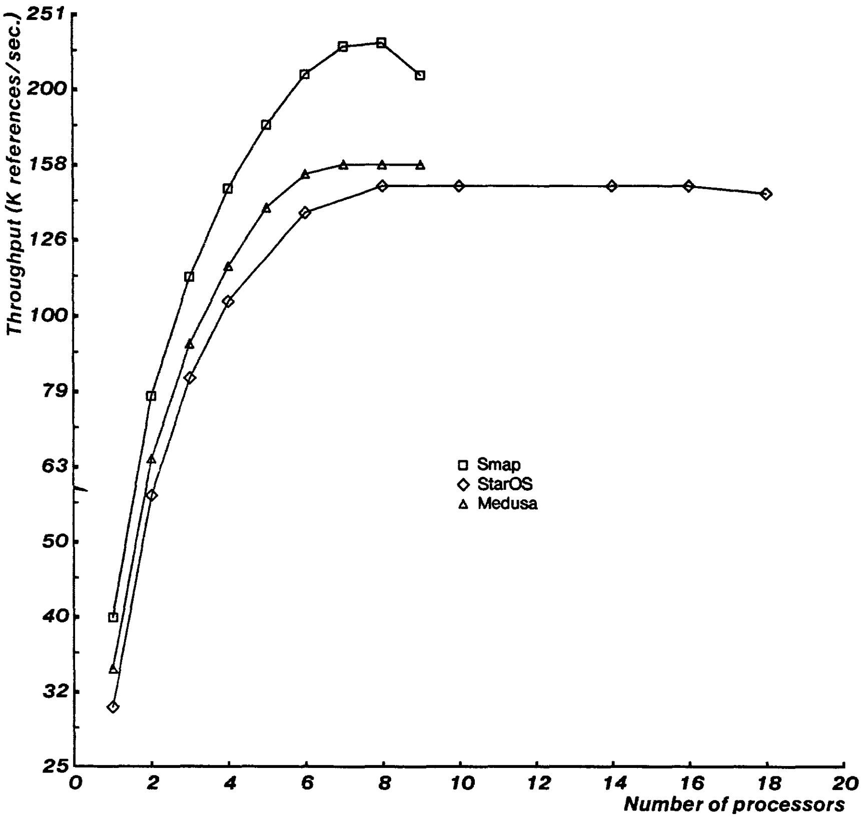
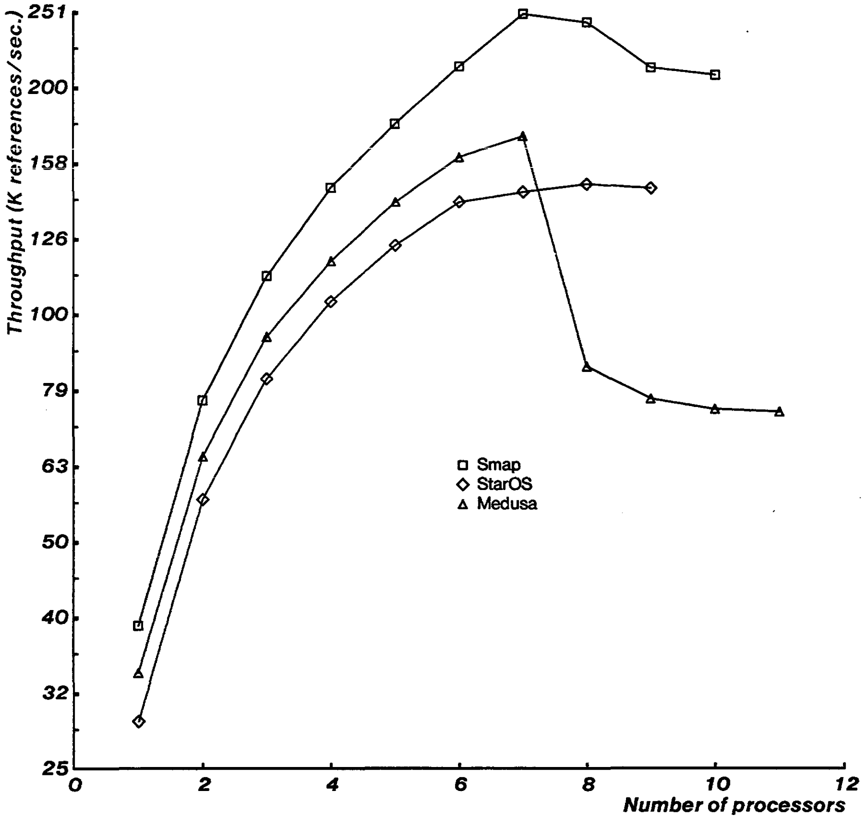


Figure 3-5 plots the throughput of intercluster memory references when the source cluster is the bottleneck—each Cm in the source cluster makes references to a different Cm in some other cluster. The maximum throughput for Smap is approximately 250K refs./sec., for MEDUSA about 170K refs./sec., and for STAROS about 150K refs./sec. Smap performs substantially better in the presence of contention, but the algorithm it uses may cause starvation. The decrease in throughput between seven and eight Cm's for MEDUSA, and a much smaller decrease between eight and nine Cm's for Smap, are due to their scheme for avoiding deadlocks over contexts. MEDUSA's scheme is more costly, but it guarantees freedom from starvation as well.[3] There is no corresponding decrease for the STAROS microcode because it does not address the problem of deadlock over contexts.

[3] This expense could be reduced significantly if the implementation were optimized for simple operations, such as memory references.

Figure 3-5 Intercluster References—Saturation of Source Cluster



We have been discussing the performance of mapped memory references under various loads. Many of the curves, especially in Figure 3-2, exhibit characteristically different shapes, which can be understood by studying their contention-resolution algorithms.

When several Cm's repeatedly reference the memory of another Cm in the cluster, contention occurs at the destination Slocal. The algorithm used to resolve this contention is implemented by Kmap microcode. The curve labeled Smap in Figure 3-2 shows the performance of the relatively simple but starvation-prone algorithm used by Smap. In this algorithm, whenever a context servicing a request encounters a busy Slocal, the Pmap waits for 20 microcycles and then lets the context retry the reference. The wait is intended to increase the chances of the reference in progress completing before the retry is done. There is a trade-off in this scheme between the length of time the Pmap is allowed to idle and the probability that a request may have to retry many times before succeeding. Increasing the wait time lowers the probability of starvation but wastes Pmap cycles, thereby slowing down requests for other operations that could proceed in parallel.

In the algorithm used by StarOS, a bit vector of length 16 is used to record contexts that are waiting for the Slocal. The vector is divided into 2 bytes: one records which of the eight contexts have been waiting a "short time" for service; the other records which have been waiting a "long time." If a context is not waiting for service, both of its bits are zero; otherwise, either its "short" bit or its "long" bit (but not both) are one. When a context attempts a reference, it sets its "short" bit. When the Slocal becomes free, in round-robin fashion the Kmap first scans the "long" byte. The first time a one bit is encountered, its context is serviced. If no one bits are encountered in the "long" byte, the contents of the "short" byte are copied to the "long" byte, and the "short" byte is zeroed.

This algorithm is a good approximation to FIFO (the Kmap hardware does not permit a perfectly FIFO algorithm to be implemented). The scheme uses between 20 and 24 Pmap cycles per completed memory reference and is the least expensive of the three starvation-free algorithms. An interesting aspect of the implementation is that the number of Pmap cycles per completed reference actually decreases as the number of contexts increases (since a shorter average search is necessary to locate a one bit). This decrease is easily noticeable as a gradual increase in throughput as the number of Cm's is increased.

The algorithm used by Medusa involves placing waiting contexts in a nearly FIFO queue (see Section 5.1.2). The cost of insertion and deletion is independent of queue length. This scheme uses between 27 and 31 Pmap cycles per completed memory reference and is therefore slightly slower than the bit-vector implementation of StarOS under heavy contention.

The last algorithm characterized is the one used by the Echoes microcode. It also uses a nearly FIFO queue to record contexts waiting for a busy Slocal. However, since the implementation was coded in Mumble, and since no special effort was made to optimize it, the throughput is not as high as it is for the other systems—between 46 and 52 cycles are expended per completed reference.

3.1.4. Effect of Memory Locality on Throughput

A typical program is composed of four entities: code, stack, private variables, and global (i.e., shared) variables. Throughput is a function of the relative location (local or remote) of these four entities. The sensitivity to performance can be measured by comparing the effects of local and mapped references to each entity. Six different memory reference patterns were measured for the three experiments (see the first three sections of Appendix A for more details):

All local. All memory references are directed from the processor to its local memory.

Only global variables mapped. Only the memory references that are shared between the cooperating processes are remote and require mapping by the Kmap.

Table 3-3 Uniprocessor Execution Time as a Percentage of Local Execution Time for Different Nonlocal Program Entities

Application	PDE (method 4, asynchronous)	Quicksort	Integer programming
All local	100	100	100
Only globals mapped	104.5	122	102
Only private vars. mapped	107	111	108
Only stack mapped	118.5	119	140
Only code mapped	231	229	223
All mapped	261.5	274	270

Only private variables mapped. Only the memory references to the variable that are private to a process (those that are not shared between processes) are remote and require mapping by the Kmap.

Only stack mapped. Only the memory references to the processor's stack area are remote and require mapping by the Kmap.

Only code mapped. Only the memory references to the application code are remote and require mapping by the Kmap.

All mapped. All the memory references are to nonlocal memory and require mapping by the Kmap.

Table 3-3 summarizes the extra execution time required as a function of different memory-reference patterns. The following conclusions can be drawn:

- When all references are mapped, execution takes 2.6 to 2.7 times longer.
- When code is mapped, the execution takes 2.2 to 2.3 times longer. Hence it is very important for code to be local, even at the expense of multiple copies.
- When the stack is mapped, execution takes 1.2 to 1.4 times longer. The quicksort is worse because it consists of a large number of small routines that require frequent stack access to perform the call / return sequences.
- When private data is mapped, the execution time is 1.1 times longer.
- When global data is mapped, the ratio is 1.02 when global accesses are infrequent to 1.22 when global accesses are relatively frequent. These figures are encouraging because they imply that large shared data structures may be located anywhere in the system without significant performance degradation.

These results underline the importance of localizing code, stack, and private variables in the Cm's local memory. The *hit ratio* (the fraction of all memory references directed to local memory) when only global variables are mapped is on the order of 97.5 percent for the PDE, 90.5 percent for quicksort, and 99 percent for integer programming. Global data can be placed anywhere in a cluster without severe performance degradation.

3.1.5. Hardware Utilization and Growth Potential

The Smap microcode executes a two-microinstruction loop (called the *idle loop*) whenever there is no useful work for it to do. The time in the idle loop shows the utilization of the Kmap. The *Slocal busy loop* is seven microinstructions long and is used to show the amount of contention for Slocals and memories. This loop is executed when the Kmap tries to access a memory but must wait because the Slocal is busy executing a previous reference. A successful reference to an Slocal takes four microinstructions.

For the quicksort (described in Section A.2), employing 8 Cm's with all code local, the Pmap was idle 78.5 percent of the time, waiting on the Slocal 11 percent of the time, and performing mapping references only 10.5 percent of the time. For the PDE (see Section A.1), using 8 Cm's with all code local leaves the Pmap idle more than 96 percent of the time.

Figure 3-2 indicates 3 to 4 Cm's simultaneously making all references to the same Cm before local memory/Slocal saturation was reached, while Figure 3-3 depicts Kmap saturation at 6 to 7 Cm's. As several applications described earlier indicate, hit ratios of 90 percent are commonly achieved simply by making code, stack, and private variables local. Thus the current Cm* hardware could support from 30 to 70 processors per cluster before saturation. Conversely, assuming 14 Cm's per cluster, the Cm* hardware could support Cm's that gave two to five times the performance of the LSI-11.

Figures 3-4 and 3-5 demonstrate intercluster saturation at about 8 Cm's. Again assuming a 90 percent hit ratio and assuming that mapped references are evenly distributed throughout the system and their frequency does not increase as the number of clusters rises,[4] the Cm* hardware can support an arbitrarily large number of clusters with up to 40 Cm's per cluster.

In summary, no single component is a performance bottleneck in the Cm* system. Furthermore, expansion by a factor of two to five in number of Cm's per cluster or performance of individual Cm's is well within the capability of the Slocal/Kmap implementation.

3.2. Reliability Studies on Cm*

The 50-module Cm* system provides a unique opportunity for gathering data about computer failures, both hard (permanent) and transient. Cm* hard-failure data was used in a reliability comparison between industrially produced components and CMU-built one-of-a-kind components. It was found that CMU-built components, which did not use burned-in parts, generally had a higher failure rate. It was also shown that, as expected, the distribution of hard failures follows the exponential distribution. Cm* transient error data was also analyzed for the distribution of inter-

[4] If there are m clusters with n processors per cluster, intercluster traffic becomes $\frac{m-1}{m}$ outbound from this cluster and $\frac{m-1}{m}$ inbound from other clusters for a total of approximately $0.2n$. If saturation occurs at eight simultaneous references, it will occur when $0.2n = 8$, or when $n=40$.

arrival times. It was found that the distribution follows a decreasing failure-rate Weibull function. This is at variance with the standard assumption that transient errors exhibit an exponential distribution with a constant failure rate.

3.2.1. *Hard Failures*

Hard failures, or permanent faults, are continuous and stable, reflecting an irreversible physical change in the hardware. Cm* hard-failure data, collected from the engineering log book, was used to calibrate existing reliability models. The mean time to failure (MTTF) was calculated assuming failures were independent. The MTTF was obtained by dividing the total time by the total failures. To combine data for all the Cm's, a concept called *module time* was introduced. If there are several modules running during a period of time, then the module time is the sum of the time that each module was up. Module time is divided by the total number of recorded failures for the entire multimodule Cm*. This yields the MTTF for a single "typical" module. Because of the small number of failures per computer module, it is a more realistic reliability measure than the MTTF for any particular module. Table 3-4 presents this module-time data and the MTTF, measured in module hours, for Cm*.

The "complexity in chips" mentioned in the table is a measure of the actual number of chips used in each component. In the case of the LSI-11, the actual number of chip sockets used is 76; of these, 72 contain digital integrated circuits. Since unused functions on the chips add up to the equivalent of 4 unused chips, the number of chips used is recorded as 68.

An ANOVA (analysis of variance) on the error-log data shows that uncertainties associated with module commissioning dates i.e., initial power-up and integration

Table 3-4 Cm* Hard-Failure Data from February 1977 to May 1978

Component	Complexity (chips)	# of modules	Total time (mod. hrs.)	Total failures	MTTF (mod. hrs.)
Kbus	3	138	36,696	8	4,587
Pmap	3	106	37,416	12	3,118
Control store	6	116	68,328	4	17,082
Data RAM	3	142	37,082	2	18,540
Linc	3	116	22,608	0	—
DEC LSI-11	14	68	163,200	10	16,320
Slocal	10	126	120,720	5	24,144
4K memory	56	21	260,568	5	52,003
16K memory	104	10	122,280	5	24,456
SLU	28	17	223,248	5	44,650
Power board	6	16	195,456	3	65,152
Refresh	14	16	162,912	0	—

into the system) were insignificant. The failure distribution was shown to follow the exponential distribution—i.e., a constant failure-rate Poisson process used in the Military Standard Handbook (MIL 217B) reliability model [Siewiorek *et al.* 78b].

MIL 217B assumes that the failure of electronic components is a Poisson process and the failure distribution follows the exponential distribution, which is characterized by a constant failure rate over time. The failure rate for a single IC chip can be predicted using the following MIL 217B model parameters: a learning factor based on the maturity of the production process, a quality factor based on the procedure for incoming screening of components, the ambient operating temperature, a factor based on the benignity of the operating environment (considering factors such as humidity and vibrations), and two complexity factors based on the number of random logic gates and the number of memory bits in the component.

Cm* chip-failure data and vendor data were used to calibrate the MIL 217B model and were then compared with its predictions. The model turned out to be too pessimistic by a factor of 16 to 64, compared to the Cm* data. That is, it predicted 16 to 64 times as many failures as actually occurred. One can speculate that MOS technology might not yet have matured by 1972, when the data was gathered for the creation of the 217B model. Since Cm* uses mostly 1976–77 components, there are many MIL 217B parameters that can be altered to take into account the maturity of the production process. Cm* data was compared with the predictions generated by various parameter changes in the model. As a result of these comparisons, a *modified* MIL 217B model was proposed: The complexity factor for MOS chips was *derated* by a factor of 16 (i.e., 16K memory chips were treated as 1K memory chips, etc.). Based on this modified MIL 217B model, a PMS-level reliability model for Cm* was also presented [Siewiorek *et al.* 78b].

3.2.2. Transient Errors

Transient errors are manifestations of faults due to temporary environmental conditions. Very little is known about transient failures. Data collected on Cm* and other CMU computers has contributed to the understanding of this phenomenon. On Cm*, transient-error data was collected by an Auto-Diagnostic program [Scelza 79]. The Auto-Diagnostic continuously exercised the Cm* system by running diagnostic programs on all otherwise idle computer modules. Whenever an error was detected by the diagnostic program, the information was printed on the console terminal. Looking through the console log, one could determine occurrences of transient errors. A more detailed analysis of Cm* error data was presented by Tsao [Tsao 78]. It was found that the interarrival times of transient errors follows a decreasing failure-rate Weibull distribution. This is at variance with the standard assumption of a constant failure rate (Poisson process, exponential distribution) used in reliability modeling. The Weibull distribution observed on Cm* was also observed on several PDP-10 systems [McConnel *et al.* 79a]. A summary of these findings is presented in Section 3.2.3.

A set of four diagnostics are run continuously on the Cm's. These tests exercise (1) the memory, (2) the instruction set, (3) the traps and interrupts, and (4) the Slocal

Table 3-5 Cm* Transient-Error Events Sorted by Mode and by Test Type

	Type of test				
Error modes	Memory	Instruction	Interrupt	Slocal	Total
Burst only	8	6	1	16	31
Simultaneous only	6	1	0	20	27
Burst and simultaneous together	3	1	0	10	14
Isolated	7	6	0	18	31
Total	24	14	1	64	103

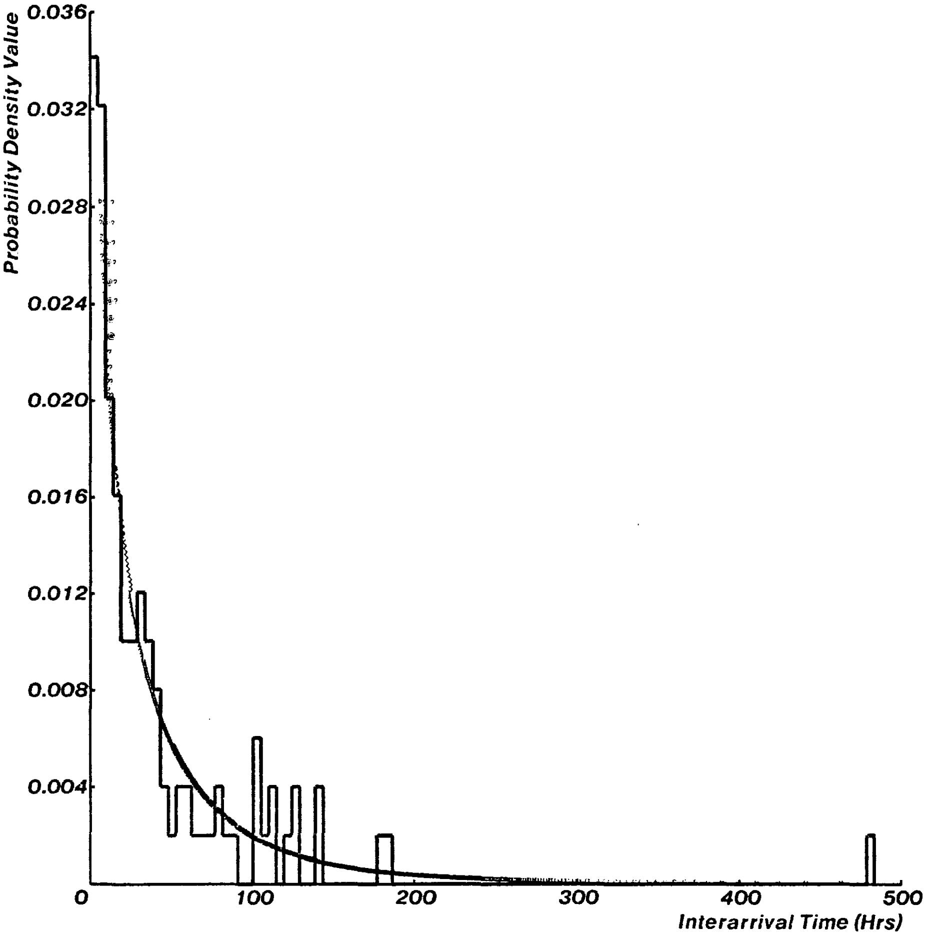
and a small part of the Kmap. The memory test is divided into 13 subtests, which include a gallop test, marching ones and zeros, and shifting ones. It takes approximately 13 minutes to complete one pass through 56K bytes of dynamic MOS RAM. The instruction-set test and the interrupt-and-trap test are designed to test the functioning of the LSI-11 processor. These are short tests, so many passes are done before moving to the next diagnostic. The Slocal diagnostic performs a number of functions. First, it tests the registers and data path of the Slocal. Second, it exercises a part of the Kmap. Finally, it runs a few tests on portions of memory.

Previously reported data [Siewiorek *et al.* 78a] indicated that several computer modules sometimes reported detecting errors almost simultaneously. Three basic patterns of occurrence were noticed in transient errors: multiple errors occurring together in the same Cm (burst), simultaneous errors reported by different Cm's (simultaneous), and isolated errors (isolated). It also was observed that a single transient-error event sometimes would be manifested as both the burst type and the simultaneous type. Table 3-5 groups the recorded errors for the period between September 1977 and August 1978 into these four classes.

Observations indicate that the most common cause of burst errors is the destruction of the diagnostic program. A garbled diagnostic program can cause either spurious halts or a burst of reported errors, since successive restarts of the diagnostic program will be unsuccessful and will result in consecutively reported errors. It also is possible that such burst errors arose when faulty transmission of code caused a bad copy of the diagnostic program to be loaded. But this is not likely because all such transfers are checksummed. Once a checksum error is detected, a reload is started. Bursts also may be caused by transient errors of a duration that is longer than the time resolution of the diagnostic, but the majority of observed burst errors did not fit this hypothesis.

The simultaneous reporting of errors in several Cm's is the most interesting observation. It is conjectured that a systemwide transient failure causes this type of error. Two possible sources were proposed: a Kmap error during Slocal testing or common DC power-supply glitches. It is known that turning power on and off in one Cm causes errors in other modules. These simultaneous errors could be user-

Figure 3-6 Distribution of Cm* Transient Errors, September 1977 to August 1978

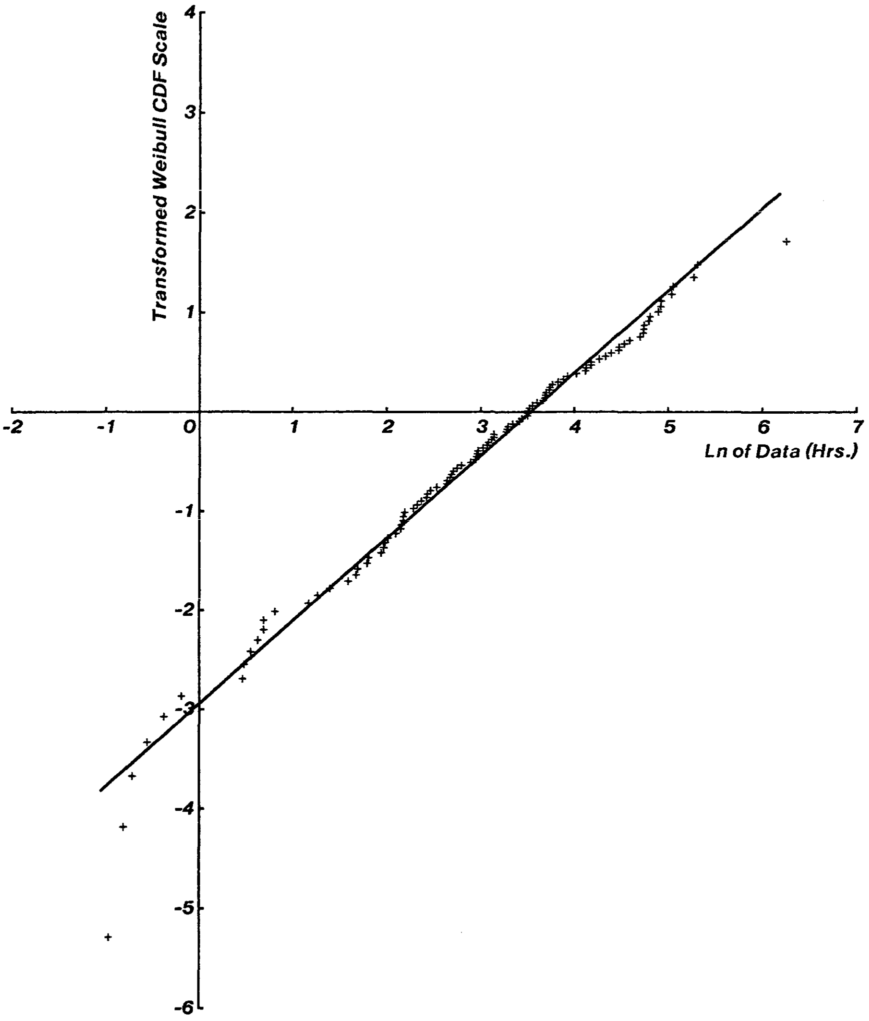


induced events of this type that were not properly recorded in the system log. If these simultaneous errors were truly transient, one-fourth (27 events) of all transient events affected more than one Cm.

3.2.3. *Analysis of Transient-Error Interarrival Time*

Transient-error data was processed and analyzed for the underlying statistical properties, with the aid of the SEADS transient-error statistical analysis program [McConnel *et al.* 79b]. Figure 3-6 shows the adjusted histogram of the interarrival for Cm* transient errors. The histogram of the distribution is overlaid with the max-

Figure 3-7 Weibull Plot of Cm* Transient Errors, September 1977 to August 1978



imum likelihood estimator (MLE) Weibull probability density function. Figure 3-7 shows the interarrival data for Cm* plotted on Weibull probability paper. The straight line drawn on the plot is a least-squared-error (LSE) linear fit to the data. Note that most of the visual deviation is due to relatively few points at the lower end. This deviation is due primarily to the transformation induced by the Weibull probability paper, which is not very accurate for low-end data points. The near collinearity of the data points, tracking the LSE line, shows that the sample follows a Weibull distribution.

Table 3-6 Statistics for Transient Errors

	TOPS-C reload	PDP-10 reload	PDP-10 parity	Cm*	C.vmp[1]
Time (hours)	2,646	8,576	8,596	4,222	4,921
Errors	195	636	74	103	50
Interarrivals	196	640	78	104	51
μ(wall-clock time)	13.5	13.4	110.2	40.6	96.5 (328)
σ	16.5	24.6	244.9	59.8	167.8 (471)
α (Linear)	0.864	0.684	0.500	0.834	0.711
α' (MLE)	0.826	0.639	0.481	0.779	0.654
λ (Linear)	0.0843	0.109	0.0206	0.0294	0.0146
λ' (MLE)	0.0826	0.106	0.0203	0.0288	

[1] The pessimistic value discussed in [Siewiorek *et al.* 78b] is used throughout for C.vmp because there were too few interarrivals in the optimistic value (shown in parentheses for the mean and standard deviation) to be statistically significant.

Table 3-7 90 Percent Confidence Intervals for Weibull Alpha and Lambda

	PDP-10 parity	Cm*	C.vmp
$[\alpha_{high},\lambda_{high}]$	[0.566,0.0307]	[0.893,0.0359]	[0.806,0.0214]
$[\alpha_{low},\lambda_{low}]$	[0.412,0.0134]	[0.693,0.0231]	[0.558,0.0099]

Table 3-6 lists some general statistics about the interarrival times for the five sets of data: TOPS-C system reloads on the CMU Computation Center DEC 2060 (under the TOPS20 operating system), PDP-10 system reloads on the Computer Science Department DEC KL-10 (under the TOPS10 operating system), PDP-10 memory-parity error interrupts on the KL-10, Cm* transient errors, and C.vmp[5] system crashes [McConnel *et al.* 79b]. In all cases, the mean is less than the standard deviation, indicating a decreasing failure rate ($\alpha < 1$). The Weibull shape parameter is α, and λ is the Weibull scale parameter.

For the last three sets of data, the 90 percent confidence intervals for α and λ also were generated. These values are listed in Table 3-7. Note that the range of values for α does not include 1.0, as it would have to if the data did follow the exponential distribution.

[5] C.vmp is a triplicated NMOS LSI-11 microprocessor with majority voting at the bus level [Siewiorek *et al.* 78b].

Confidence-interval tests on the MLE Weibull parameters and the chi-square goodness-of-fit test confirmed the hypothesis that the data follows a decreasing failure-rate Weibull distribution. This is significant because past publications on the problem of transient errors have always assumed the exponential distribution for ease of computation. No other data has been published to support that assumption. This observation of a decreasing failure-rate Weibull distribution means that the problem of modeling transient errors must be reconsidered.

3.2.4. *Transient-Error Data for February and March 1980*

During February 1980, the Auto-Diagnostic reported a total of 45 errors that were actual detected diagnostic faults found by individual test programs on the Cm's. It is evident from the console log that cluster 3, Cm 14 had a hard failure, as 21 errors were reported. Errors reported simultaneously are assumed to be due to the same transient fault, so only one transient error is counted. During this month, there were 4 simultaneous error events that caused the reporting of 5 redundant errors. After discounting these nontransient diagnostic errors, there were 19 transient-error events in February 1980.

Tables 3-8 and 3-9 present the mean time between error (MTBE) for each cluster and the detected errors (sorted by tests). A new test, the parity test, is used on the 50-module Cm*. This test diagnoses the parity generating and checking portion of each Cm. It is evident that the parity test is a very sensitive diagnostic test for transient errors because many more parity errors were reported by this test than by the Slocal test, which, in the past, used to report the largest single group of errors. The newly discovered sensitivity of the parity test is also reflected in the low MTBE, compared with an MTBE of 218 module-hours as reported in [Tsao 78].

3.3. Summary

Once a large parallel processor has been built, several scientific questions arise. Hardware-performance evaluation is critical in validating the design decisions and

Table 3-8 Cm* Transient Errors, February 1980

Cluster	1	2	3	4	5	Total
Module hours	408	312	481	309	212	1,722
Errors per test						
Parity	1	3	7	2	3	16
Slocal	0	0	0	2	0	2
Instruction	1	0	0	0	0	1
Total errors	2	3	7	4	3	19
MTBE (module hours)	204	104	69	77	71	91

Table 3-9 Cm* Transient Errors, March 1980

Cluster	1	2	3	4	5	Total
Module hours	241	212	326	385	365	1,529
Errors per test						
Parity	1	2	3	1	3	10
Slocal	0	1	0	0	0	1
Instruction	0	1	0	0	0	1
Total errors	1	4	3	1	3	12
MTBE (module hours)	241	53	109	385	126	127

calibrating the hardware to provide a basis for software-performance experiments. An orthogonal dimension is the evaluation of the availability and reliability of the parallel processor.

These issues have been studied by a set of experiments on Cm*. Measurements of the speed of individual Cm's showed a variation of ±4.6 percent. As no attempt has been made to compensate for this variation, it is part of the experimental error in all the other results. The mapped memory-reference and contention experiment reveals a trade-off between functionality and performance. For example, Smap provides higher throughput than Medusa or StarOS but fails to prevent starvation. Three workloads—PDE, quicksort, and integer programming—were studied to determine memory reference rates, interreference times, and instruction throughputs both for a single Cm and for multiple Cm's. The same workloads were used to study how throughput was affected by the placement of the code and data. The results emphasize the importance of localizing code, stack, and private variables in the memory of the executing Cm. Global data can, however, be economically located anywhere in the system, since only 2 to 22 percent of the memory references were directed to global memory.

The hardware-utilization experiments stressed the limits of the hardware in an attempt to determine where further expansion of Cm* might become unprofitable. The results indicate that no single component was a performance bottleneck and that the interconnection network and switching structure could support an expansion of two to five times in the number of Cm's per cluster or a similar improvement in individual Cm performance.

Hardware-reliability studies were undertaken to gather data on transient and permanent faults. During an entire year, records were kept of the occurrence of four classes of errors: burst, burst and simultaneous, simultaneous, and isolated. The data shows that one out of every four recorded errors was manifested in more than one Cm. Also, transient errors proved to be twenty times more frequent than permanent errors. The analysis of transient-error data showed that the occurrence of transients closely matched a decreasing failure-rate Weibull probability distribution. In fact, the reliability figures for Cm* were comparable to those of commercial computing systems within the CMU Computer Science Department.

Acknowledgment. Section 3.2 was adapted from the original by Michael Tsao for [Jones and Gehringer 80].

II. Operating Systems

4. Operating-Systems Overview

StarOS and Medusa are the two operating systems that have been written for Cm*. Because they were constructed to run on the same architecture, they exhibit a good deal of similarity in structure. Nonetheless, delving beneath the surface reveals that the goals and philosophies of the two systems are quite different: StarOS strives to provide the services of a general-purpose operating system, while Medusa aims to maximize performance instead of flexibility. This chapter presents a top-down view of the two systems, concentrating on the similarities. The next two chapters give bottom-up system descriptions, showing how the differing goals have affected their structures.

4.1. Basic Goals

While Medusa and StarOS both aim to make the full facilities of a multiprocessor available to users, they are structured in such a way that their internal complexity is minimized. Both are composed of a number of relatively small component parts communicating through rather small interfaces. This kind of *modularity* makes the structure of the system easier to comprehend and facilitates changing or extending the functionality of its components.

An operating system is said to be *robust* if errors in one process or processor do not jeopardize the correct operation of others. Both operating systems count robustness among their main goals. They should be able to continue running even when some of their resources are temporarily unavailable and should be able to reconfigure themselves dynamically in response to an increase or decrease in workload. The distributed nature of the hardware aids in minimizing the number of *singularities*—single resources without which the system cannot run.

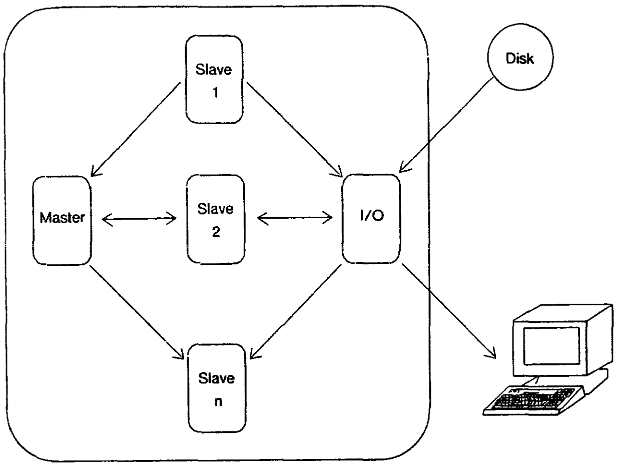
4.1.1. Task Forces

A *task force* is a way of abstracting a unit of work. It is a collection of relatively small processes that cooperate to achieve some goal. For example, the task force that performs an application such as matrix multiplication might be made up of:

- Several identical processes that produce equal portions of the result matrix.
- One *master process* that is responsible for parceling out the work and starting the other processes.
- Another process to read commands input by the user and report progress and results to the terminal (see Figure 4-1).

More detail on the structure of task forces for specific applications can be found in Section 11.4.

Figure 4-1 A Simple Task Force for Matrix Multiplication



It is important to note that the processes that make up a task force are quite small. In a serial program written for a uniprocessor, functions such as output or terminal communication are performed by procedures, not separate processes. On a multiprocessor, both the nature of parallel programs and the availability of processors suggest that dedicated processes be provided for these functions. Accordingly, both MEDUSA and STAROS endeavor to make the management of processes cheap so that efficiency considerations do not discourage users from decomposing algorithms into small processes. Both systems, however, expect process creation and deletion to be less frequent than process-management operations, such as scheduling, interrupting, or communicating with a process. Thus optimization efforts have focused on the latter functions. A common paradigm is for a single process to act as a *server*, handling requests from several other processes for a particular set of operating-system services.

The task-force concept is basic to both systems, which are themselves structured in terms of task forces. Most of the functions provided by MEDUSA are provided by utilities, which are task forces. MEDUSA's definition of a task force is specific: It consists of all processes that routinely share memory at run time (see Section 5.1.1). The STAROS notion of a task force is a bit more general, permitting *any* group of cooperating processes, even the entire operating system, to be considered as a single task force.

4.1.2. Messages

Both operating systems are *message based*. Communication between task forces is almost exclusively in terms of messages. The component processes of a single task force, too, often communicate via messages rather than shared memory. Both systems have invested considerable care in making message operations efficient so that messages can be used with the same freedom with which procedure calls are used in a traditional operating system.

MEDUSA conveys messages by copying data (by value), whereas STAROS messages are conveyed mainly by passing pointers (by reference) [Sindhu and Singh 83]. Nevertheless, the MEDUSA and STAROS message systems have many aspects in common. Messages are sent from one process to another via a receptacle known in MEDUSA as a *pipe* and in STAROS as a *mailbox*. Each receptacle provides its own buffer to hold messages; thus congestion in one receptacle does not prevent other receptacles from accepting messages.

Mailboxes and pipes perform a buffering function similar to that of an I/O subsystem. An I/O subsystem permits a process to exchange information with some external medium, such as a terminal or a disk. Since disks and processors do not, in general, access data at exactly the same speed, data is *buffered* in some area of memory between the time it is produced and the time it is consumed. A message system allows a process to exchange information with other processes. Consider a producer process sending data to a consumer process in the form of a message. If the producer delivers data faster than the receiver can consume it, the data is buffered in the pipe or mailbox.

A ***Receive*** operation causes data to be copied out of the mailbox and into some program data structure. The ***Receive*** specifies where the data is to be placed. If the receiver has already performed a ***Receive*** and is waiting for the message to arrive, the message need not be buffered in the mailbox. Instead, the message system can copy it directly to the destination specified by the receiver. The message is copied only once. Otherwise, the message would have to be copied twice, once when put into the buffer and once when taken out. Both MEDUSA and STAROS perform this optimization.

When a suitably authorized process performs a ***Send*** to a mailbox or a pipe, the message is buffered, unless the receptacle is full. In this case, the result returned to the sending process informs it of this fact. In MEDUSA, in case of a full pipe, if the ***Send*** is an ***Unconditional Send***, the sending process is suspended, until space becomes available, at which time the ***Send*** is completed. A STAROS ***Send*** behaves like a MEDUSA ***Conditional Send***: if the buffer is full, the sending process is merely informed and allowed to continue.

Similarly, an authorized process may perform a ***Receive*** from a mailbox or a pipe. A message is retrieved from the receptacle unless the receptacle is empty. In this case, the result returned to the receiving process informs it of the empty status. In case of an empty receptacle, the receiving process (in either operating system) can choose to be suspended until a message becomes available, at which time the ***Receive*** is completed. A process also has the option of performing other work in the interim, relying on the event system to notify it when a message becomes available. For efficiency's sake, all message operations in MEDUSA and nearly all in STAROS are performed by Kmap microcode. Message operations are summarized in Table 4-1.

Table 4-1 Message Operations and Their Parameters

Send	Receive
Mailbox or pipe	Mailbox or pipe
Message to be sent	Destination for message
Concurrency	Concurrency

"Concurrency" indicates whether the process wants to be notified or awakened later if the operation cannot be performed due to the mailbox's being full or empty. "Destination for message" refers to the location where the message is placed when it is received from the mailbox or pipe. Note that these specifications are idealized; the implementations in both operating systems differ in detail from this model.

Closely associated with the message system is the *event system*, which informs a waiting process that a message has arrived. If a process is suspended,[1] the process once again becomes eligible for execution. A process may also wait on an entire set of events (this is called a ***Multi-Event Wait*** in MEDUSA). It will be awakened as soon as any one of the events occurs. Section 5.3.3 provides an example of how ***Multi-Event Waits*** are used. Both event systems are flexible enough to associate events with other phenomena besides the arrival of messages, although in STAROS no such events have been defined. MEDUSA associates events with semaphores, files, and task forces, as well as with pipes.

In a uniprocessor system, procedure calls are performed synchronously. The calling procedure does not continue execution until the called procedure returns. The caller is effectively *suspended* while the called procedure executes. In a multiprocessor, by contrast, it is possible for the caller and callee to execute simultaneously on different processors. Conceptually, the caller sends a message to the callee, which may or may not be executing on a different processor. The message contains the arguments of the procedure call. When the called procedure finishes, it sends a message back to the caller. If a result is to be returned, it is included in the message. As with all message operations, the caller may choose to continue execution while waiting for the calling procedure to reply.

Since message operations are more expensive than ordinary procedure calls, both operating systems implement many procedure calls in the traditional way, using ordinary LSI-11 instructions. Asynchronous procedure calls are used where the benefits of concurrency outweigh the expense of message operations, or where the caller and the callee must be protected from each other. All requests for operating-system services fall in the latter category. A typical case is a request to allocate more memory. The memory manager must update sensitive data structures that user programs must not be allowed to modify. Executing the memory manager in a separate process guarantees that its data can be protected from user programs that invoke it.

[1] A STAROS process may choose not to suspend execution while waiting for a message but rather to perform other work in the meantime.

The two operating systems differ slightly in how they grant requests for system services. In MEDUSA, these requests are serviced by preexisting processes in special task forces called *utilities*. A utility call by a MEDUSA user process is treated differently from other calls performed by message passing in that it automatically suspends the caller. This prevents a runaway user process from flooding the system with service requests. It also means that utility calls are treated the same way as the microcoded kernel operations, which are also requests for system services that require the invoker to wait. In STAROS, by contrast, a request for a system service may cause creation of a new process. A STAROS process is allowed to continue execution while its request is being serviced.

4.1.3. Policy versus Mechanism

MEDUSA and STAROS share the philosophy that the policies adopted by an operating system should be separate from the mechanisms that carry out those policies [Levin *et al.* 75]. Mechanisms can be built efficiently at low levels of the operating system without needlessly constraining the policies pursued by the rest of the system. For example, consider process scheduling. An efficient mechanism for interrupting one process and resuming another is a basic requirement of multiprocessing. The decision of which process to resume next depends on policy trade-offs between conflicting scheduling goals, which may change in importance as the nature of the workload changes or experience with the multiprocessor system increases.

Both operating systems have placed code for process switching, or *multiplexing*, in an operating-system kernel,[2] which is replicated in each Cm. The kernel provides mechanisms for suspending a process, either at the end of a time slice or in response to some external event, and resuming the next process, where the "next" process is determined by the policy-level scheduler.

A higher-level program, called the *scheduler* in STAROS, decides the priority of a process and places it on a queue with processes of the same priority for later selection by the multiplexer. In MEDUSA, the *Task Force Manager* makes these policy decisions, striving to allow all the processes of a task force to execute concurrently. This is called *coscheduling* and will be explained in more detail in Section 5.5. The scheduler (and the *Task Force Manager*) need not be part of the kernel, and they need not be replicated on each processor. Multiple schedulers, may be active simultaneously, however, with responsibility for scheduling different sets of processes.

4.1.4. Objects

In both operating systems, memory references are directed to *objects*. Some objects function much like the segments or pages found in conventional virtual-memory

[2] "Kernel" is the MEDUSA term; STAROS uses "nucleus."

systems. In StarOS, these objects are called *basic objects*; in Medusa, they are called *page objects*. Memory words within these objects may be read and written using ordinary LSI-11 instructions. Other objects, such as mailboxes and pipes, are treated differently. Ordinary *read* and *write* references may not be directed to them; only special operations such as ***Send*** and ***Receive*** may access their memory.

Memory is allocated only when objects are created. The software that performs this function is known as the *Memory Manager* in Medusa and the *Object Manager* in StarOS. Both programs allocate objects of various types: for example, page, pipe, and semaphore objects in Medusa and basic, mailbox, and process objects in StarOS (these object types will be described in later chapters). They both return a protected pointer for the object that has been allocated. This protected pointer is called a *descriptor* in Medusa and a *capability* in StarOS. The memory can henceforth be accessed only by using the protected pointer or a copy of it. Medusa uses a reference-count scheme to free memory: the memory belonging to an object is deallocated when the last descriptor for it is deleted. StarOS frees storage by garbage collection.

Most objects can be manipulated by only a small number of operations. For example, Medusa pipes implement the particular abstraction of a bounded buffer of messages. The two operations, ***Send*** and ***Receive***, that can be performed on Medusa pipes enforce a FIFO queueing discipline. If the memory associated with pipes could be read or written arbitrarily, correct message queueing could not be ensured. An errant or malicious program could overwrite data before it was delivered to the recipient, or even overwrite control information, which might disable the pipe altogether. Some mechanism is necessary to allow the ***Send*** and ***Receive*** operations to read and write the memory belonging to a pipe but prevent other software from doing so. An object that is protected in this way is known as an *abstract-type*.

Amplification is the means by which abstract objects are protected. Each abstract type has a particular set of procedures that perform operations on objects of the type. The procedures are components of the abstract type's *type manager*. In Medusa, a task force serves as a type manager. In StarOS, a type manager is made up of one or more *modules*. (More detail will be provided in Section 6.1.3.)

Processes that use an abstract object hold a capability (or a descriptor) for the object. The capability is not powerful enough by itself to allow the object to be read and written by ordinary processor instructions. The capability can, however, be passed to the type manager, which has the ability to *amplify* the capability (make it more powerful), thereby gaining the right to read and write the memory associated with the object. The concept of amplification is common to StarOS and Medusa, but the details differ considerably and will be discussed in Chapters 5 and 6.

4.2. Addressing

Descriptor-based Medusa and capability-based StarOS are each built on top of the same Cm* hardware. Although presenting very different-looking address spaces to the user, the two operating systems build on a common set of addressing structures provided by Cm*. Cm*, in turn, provides mechanisms to extend the address space of the LSI-11's.

Ordinary address relocation is performed by the Slocal attached to each Cm. Each Slocal contains a set of 32 mapping registers, known as *Slocal registers*. As mentioned above, both operating systems have a kernel, or nucleus, process assigned to each Cm. Sixteen of the registers are assigned to the kernel process and 16 to the current nonkernel, or "user," process.

The most significant 4 bits of the processor-generated address select the register used to translate a particular memory reference. Collectively, the kernel and user Slocal registers are referred to as *kernel* and *user space*, respectively. A processor may switch between kernel and user space by simply toggling the *space bit* in its extended processor status word (part of the processor state). This facility allows the kernel to respond swiftly to interrupts.

An Slocal register is loaded by a ***Load Window*** instruction, which is performed by Kmap microcode. This instruction causes one of 15 *windows*[3]—Kmap registers—to be loaded with a particular descriptor (MEDUSA) or capability (STAROS). On the first local reference to the object named by the descriptor or capability, its address is propagated back to the register set of the Slocal from which the request came. As shown in Figure 4-2, the address of an object is contained within a MEDUSA descriptor; a STAROS capability instead points to a descriptor, which contains the object's address.

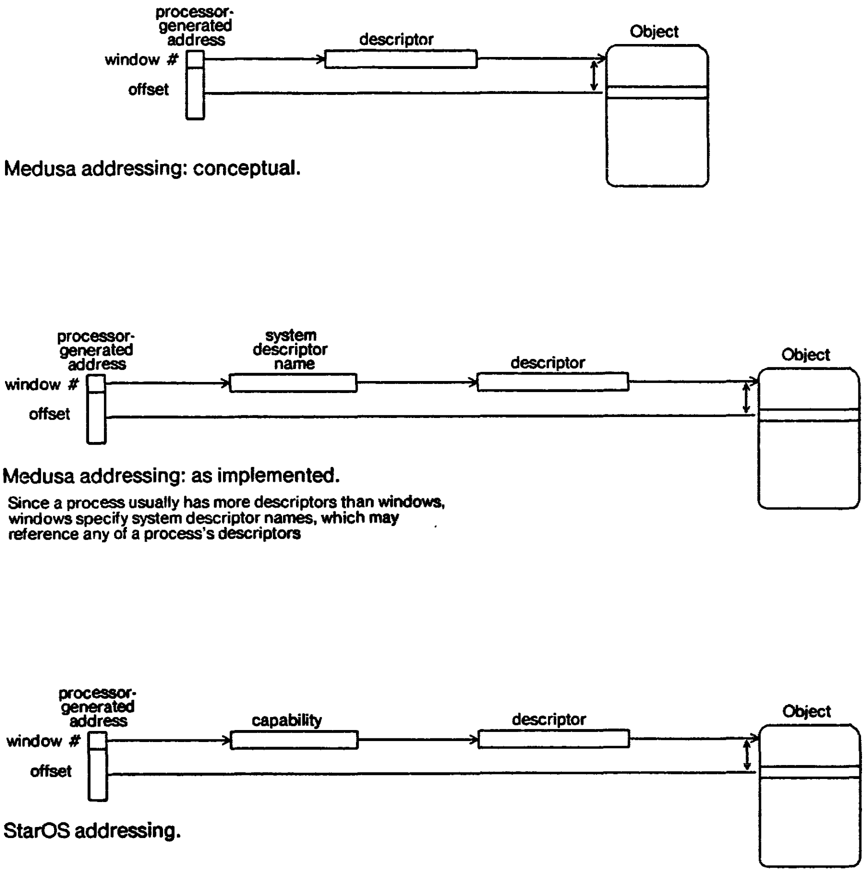
The 4K bytes of memory addressable through an individual window is known as a *page*. An object that is (actually, whose descriptor or capability is) loaded into a window may be up to 4K bytes long. If a memory reference accesses a 4K-byte object that begins on a page boundary in local memory, it may be relocated by the Slocal; otherwise it must be passed to the Kmap. (If the object is exactly 4K bytes long, then any possible 12-bit displacement falls within the object bounds, so bounds checking by the Kmap is not required (see Figure 2-6, reproduced here as Figure 4-3). The Kmap must, however, perform bounds checking on a reference to an object shorter than 4K bytes, even if it resides in local memory. A reference that is translated by the Kmap is known as a *mapped reference*.

To decide which memory references should be mapped, the Slocal mediates every memory reference generated by the processor in its Cm and concatenates the window number to the space bit from the extended processor status word. The resulting 5-bit quantity is used to index into the 32 Slocal registers. Each Slocal register is 8 bits long. The first bit is known as the *map bit*. It determines whether references to the corresponding window in kernel or user space are to be mapped.

When a ***Load Window*** instruction is performed, the map bit is turned on so that the first reference to each object will be *mapped*—that is, directed to the Kmap. On the first reference to an object, the Kmap decides whether references to the object

[3] The windows correspond to the first 15 of the 16 Slocal registers; the 16th is used for communication with the operating system, as described later.

Figure 4-2 MEDUSA and STAROS Addressing



Medusa addressing: conceptual.

Medusa addressing: as implemented.

Since a process usually has more descriptors than windows, windows specify system descriptor names, which may reference any of a process's descriptors

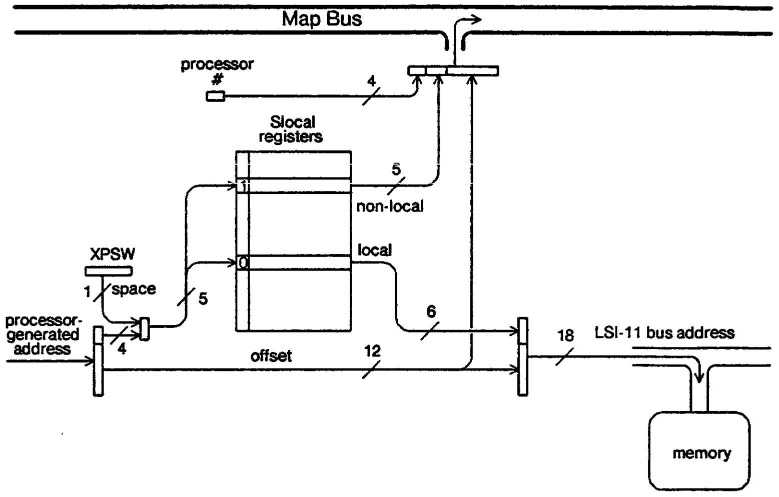
StarOS addressing.

can be relocated directly by the Slocal, and if so, it turns off the map bit.

If the reference need not be mapped, the Slocal concatenates the low 6 bits of the Slocal register with the 12-bit offset, forming an 18-bit LSI-11 address, which is used to access local memory. If the reference needs to be mapped, the 5-bit (*space bit*, *window number*) pair is passed to the Kmap, which will use its own internal data structures to find the object. (In this case, the low 6 bits of the Slocal register are unused.)

The remaining bit (between the map bit and the 6-bit page number) in each Slocal register is the *read-only bit*. If it is set, write references to the corresponding object in memory are disallowed. The read-only bit is used to authorize references relocated by the Slocal; if references are made through a Kmap, the Kmap performs this checking.[4]

Figure 4-3 Address Mapping in the Slocal



A memory reference that must be mapped is sent to the Kmap, where it is translated by using the Kmap *window register* determined by the concatenation of its (the reference's) Cm number, space bit, and window number. The window register contains the name of the referenced object (in StarOS, it actually contains a capability for the object). The object name is used to find the descriptor for the object.

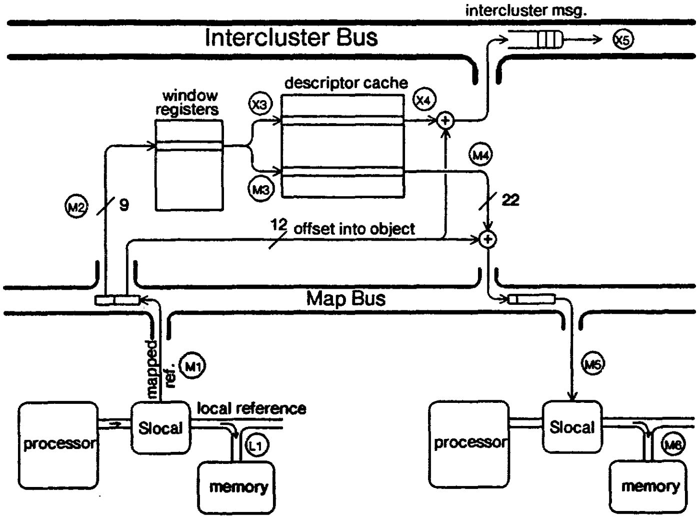
The Kmap maintains a cache of recently used descriptors. The cache is analogous to the associative memory used by other virtual-memory systems for page and segment tables. The entire process of translating a nonlocal reference is very similar in both operating systems and is summarized in Figure 4-4.

As explained above, ordinary read and write references may be made through windows 0–14. References to page 15 are always passed to the Kmap (because the map bit in Slocal register 15 is always on). When the Kmap notices that the reference is to page 15, instead of performing a memory reference, it invokes one of the microcoded Kmap procedures.

Conventionally, only *write* references to page 15 are used to invoke the microcode. The first 4 bits of a page 15 address are always 1; the Kmap treats the next 12 (Medusa) or 8 (StarOS) bits as an operation code to select a particular Kmap procedure. The data that the processor has attempted to write to the page 15 location is treated as a parameter. If more than 16 bits of parameters are required

[4] In the case of an intercluster reference, in Medusa the checking is done by the Kmap in the cluster that contains the object; in StarOS, the checking is done by the Kmap in which the capability for the object is cached. (Caching is described later in this section.)

Figure 4-4 Medusa and StarOS Addressing



Local references

L1. Slocal passes an 18-bit address to local memory across the LSI-11 bus.

Intracluster references

M1. Slocal asserts a 21-bit address on the map bus. This address consists of the following:

- 1 space bit
- 4 bits identifying the source Cm
- 4 bits of window number
- 12 bits of offset within object

M2. Upon receipt by the Kmap, the address is divided into a 9-bit window specifier and a 12-bit offset.

M3. The window register gives an object name, which is looked up in the descriptor cache.

M4. If the object is in this cluster, the 22-bit base address (4 bits of Cm number, 18 bits of bus address) is added to the offset within object. (If the object is in another cluster, proceed to step X4.)

M5. The 22-bit address is passed over the map bus. The leading four bits determine the destination Cm.

M6. The destination Slocal passes the 18-bit bus address to the target memory.

Intercluster references

X3. The window register gives an object name, which is looked up in the descriptor cache.

X4. If the object is in another cluster, the base address is added to the offset.

X5. The resulting address is packaged in an intercluster message and sent to a remote Kmap.

for an instruction, or if the instruction returns result values, the data is treated instead as the address of a *parameter block*, containing a vector of parameters.

4.3. Summary

Two operating systems, StarOS and Medusa, have been designed, implemented, and evaluated on Cm*. Both of them are classic examples of multiprocessor operating systems and have influenced the evolution of operating systems in many ways. They were among the first operating systems to provide a small kernel or nucleus supporting policy/mechanism separation, message-based communication, shared memory, object orientation, task forces, and special features to enhance robustness.

The Cm* hardware structure presented both a challenge and an opportunity for the operating-system designers. The small address space of the LSI-11 is probably the single most important architectural limitation. Both operating systems attack this problem through object orientation. The programmable address mappers (Slocal and Kmap) provided an opportunity for supporting this object model at the firmware level. Furthermore, because the Kmap is a much faster processor than the LSI-11, most speed-critical operations are implemented in Kmap microcode. Consequently, the Cm* operating systems are not easily portable to multiprocessors of a different structure.

Medusa goes farther than StarOS in reflecting the structure of the Cm* hardware and thereby obtains definite performance advantages. It has some of the aspects of a shared-memory multiprocessor operating system, although because messages are passed by value, it can also be viewed as providing a tightly coupled local area network. StarOS is more general and flexible, abstracting the Cm* architecture to a symmetrical global shared-memory model. This approach has the advantage of presenting the user with a conceptually simple yet flexible model of computation. The major differences between Medusa and StarOS are due to their different views of the architecture. Both systems were, however, developed and evaluated in the same place and in approximately the same time frame. The technical interaction between the two design groups provided superb cross-fertilization and synergism. The next two chapters will present the main concepts of Medusa and StarOS and contrast their differences.

5. MEDUSA

Of the two Cm* operating systems, StarOS has endeavored to build a general-purpose environment for parallel programs with widely diverse characteristics; Medusa has not attempted to shield the user from the nature of the underlying hardware. For the reader who is already acquainted with the structure of Cm*, MEDUSA is a good place to begin the study of Cm* software.

MEDUSA is composed of two major levels; the kernel and the utilities. User programs running on top of MEDUSA can be considered as a third level. The *kernel* is responsible for handling device interrupts and providing low-level scheduling *mechanisms* (Section 4.1.3). It chains together device commands to improve throughput for high-speed devices. It converts device interrupts into messages that are sent to the appropriate utility. It provides the mechanism for process switching but does not decide which process should run next. The *Task Force Manager*, a utility, makes this decision.

A separate copy of the kernel resides on each Cm. The kernel on a particular processor consists of approximately 1K byte of code plus about 500 bytes for each type of peripheral attached to the processor. In addition, the kernel includes about 4K eighty-bit words of Kmap microcode shared by all the processors.

Processes in MEDUSA are known as *activities*. The kernel can multiplex up to 16 activities per processor. A utility known as the *Task Force Manager* decides which process is to execute next; the kernel simply carries out its decisions. The kernel issues commands to peripheral devices and responds to the interrupts they generate. The kernel converts these interrupts into messages that are sent to the appropriate utility.

MEDUSA *utilities* provide services such as scheduling, memory management, and a file system. Utilities are structured as task forces, whose various processes are capable of servicing requests from different processes simultaneously. Utilities are endowed with special privileges that differentiate them from ordinary user task forces. They can call certain internal utility procedures that user processes cannot call, and they can read and write memory that user programs cannot—for example, descriptor lists (Section 5.1) and file control blocks (Section 5.3).

User programs are structured as task forces whose processes may execute in parallel; a serial user program is simply a degenerate case. The processes of a task force may communicate among themselves via shared memory, as well as by messages. A task force, however, communicates with the outside world only via the message system.

Sections 5.1 and 5.2 focus on the MEDUSA kernel. Section 5.1 explains how memory is organized and protection enforced by the kernel. It stresses features that have been built into the kernel to enhance reliability. Section 5.2 discusses one of the more important parts of the kernel, the message system that is responsible for interprocess communication. Sections 5.3 and 5.4 concentrate on the utilities. Section 5.3 lists the utilities and their functions and explains how utilities are called. It also emphasizes how the utilities have been designed to provide service in a reliable manner. Section 5.4 describes how errors, especially errors in utilities, are detected and handled. Finally, Section 5.5 explores special scheduling constraints that arise in a multiprocessor system and how MEDUSA deals with them.

5.1. Facilities Provided by the Kernel

All the memory allocated to a task force is reachable via descriptors held by the task force. As mentioned in Section 4.1.4, memory is divided into objects. Only one kind of object, a *page object*, is readable and writable by activities of a user task force using ordinary LSI-11 instructions. Other kinds of objects may be referenced only through operations that make use of the Kmap. References to a 4,096-byte page in local memory may be relocated by the Slocal without passing through a Kmap.

A second class of objects, including pipes and semaphores, are operated on only by Kmap microcode. *Pipes* are used to buffer messages. Activities may access pipes only by invoking the microcoded ***Send*** and ***Receive*** operations. MEDUSA pipes are similar to Unix pipes, with the main difference being that each MEDUSA message contains a byte count; a ***Receive*** is guaranteed to obtain the same amount of data that was transmitted by the sender. Pipes may be used for synchronization, as well as for information transfer. *Semaphores* are designed to be a faster means than pipes for synchronization among the processes of a single task force. Operations on semaphores are also implemented in Kmap microcode.

The last class of objects are those that are managed by utilities. Among these objects are file control blocks, which are manipulated by the software of the *File System* utility, and descriptor lists, to which the *Memory Manager* utility has read/write access. An activity that desires to copy a descriptor, for example, must do so by sending a request to the *Memory Manager*. The *Memory Manager* then executes a software routine that reads the descriptor from one slot in a descriptor list and writes it to another. If the activity were allowed to read and write into descriptor lists via ordinary processor instructions, it could easily gain access to any portion of memory by simply manufacturing a descriptor for it. Requiring the intervention of the *Memory Manager* takes longer, but this is of little concern because copying a descriptor is an infrequent operation.

The objects belonging to a task force are not restricted to being located in the same Cm, or even in the same cluster.[1] Distribution of objects facilitates communication among activities that reside in different clusters, since they may access the same objects (although it increases the complexity of synchronizing accesses to an object). For efficiency reasons, objects containing executable code must always be located in the same Cm that executes the code, since a processor executing remote code could run at no more than one-third speed.

[1] By contrast, a STAROS object always resides in the same cluster as its descriptor. *Capabilities* for STAROS objects may, however, be freely distributed throughout Cm*; see Section 6.1.2.

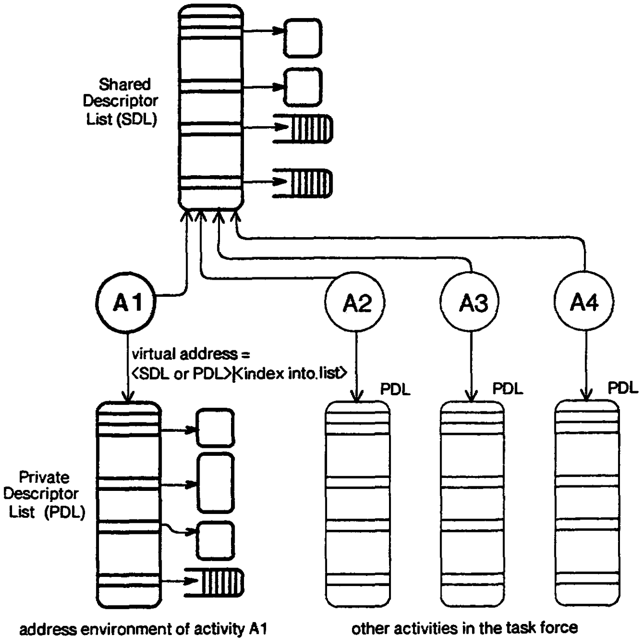
5.1.1. Descriptor Lists

A user task force possesses two kinds of descriptor lists—private and shared. Each process has its own *private descriptor list*, or PDL (Figure 5-1), whose descriptors cannot be used by any other process. The *shared descriptor list*, or SDL, is accessible to all processes in a task force. Hence, an object is either private to a particular activity or shared among all activities in a task force. The capability addressing of STAROS allows more flexibility in sharing but at the cost of an extra level of addressing indirection (see Section 6.1.2).

A descriptor list is made up of a number of *descriptor slots*. All operations on descriptors must name a slot within a descriptor list, so an activity cannot create an arbitrary bit pattern and have it treated as a descriptor. Each nonempty descriptor slot holds a descriptor, which contains a physical object address and length specification. Consequently, MEDUSA does not need a central table to hold information on all the objects in the system. This is advantageous from the standpoint of robustness, since failure of a portion of memory cannot wipe out all the object-mapping information in the system.

Each MEDUSA object contains a backpointer to each of its descriptors. Without the existence of backpointers, it would be impossible to move an object, since the phys-

Figure 5-1 The Address Structure of a MEDUSA Task Force


Shared
Descriptor
List (SDL)
A1
A2
A3
A4
virtual address =
<SDL or PDL><index into.list>
PDL
PDL
PDL
Private
Descriptor
List (PDL)
address environment of activity A1
other activities in the task force

ical address in each descriptor must be updated. A central object table would serve the same purpose but the backpointer scheme enhances robustness because if a descriptor is damaged, it can be reconstructed from information in the backpointer. The order of fields in a backpointer is different from the order in a descriptor, so the same error in both will produce a detectable difference. Lost descriptor lists may be rebuilt from a scan of objects, and lost backpointers may be recovered from a traversal of descriptor lists. Such a search would be prohibitively expensive if applied to all on-line storage but is quite feasible if limited to primary memory.

5.1.2. Microcode Reliability

Many MEDUSA microcode operations, including those that perform intercluster addressing, use more than one Kmap context (see Section 2.2.2). This raises several issues commonly encountered in distributed systems, including coherence of descriptors cached in Kmaps and starvation and deadlock over Kmap contexts. These problems are addressed by the MEDUSA microcode in a unified way.

Cache Coherence and Synchronization of Descriptor Access. At any given time, several activities may simultaneously need to read a descriptor to access the object to which it points. For example, several activities may attempt to read a page object whose descriptor is in the SDL. Some of these activities may be in different clusters, so to provide efficient addressing, copies of the descriptor will be cached in different Kmaps (Section 4.2). Simultaneously, other activities may need to update the descriptor for the same object if, for example, the object is moved. The problem is to make sure that no activity uses an outdated descriptor to access the object.

The MEDUSA solution guarantees that at least one cached descriptor is always current; the current descriptor is cached in the Kmap in the cluster where the object is located. Other cached descriptors simply contain pointers to the cluster where the object is "thought" to be. They are treated as "hints," which may be invalid at any time. If a cache fault is encountered on following such a hint (indicating that the object is no longer in that cluster), the descriptor is read from memory, and the hint is corrected.

There are several reasons that a descriptor may have to be read from memory (for instance, when an invalid hint has been followed or when the descriptor has previously been purged from a full cache). Regardless, the request travels first to the Kmap in the cluster in which the descriptor is located and then to the Kmap in the cluster where the object is located. The cached copy (if any) in the object's cluster is overwritten. When a descriptor must be written, the request follows the same route, first updating the copy in the descriptor list, then the copy in the object Kmap's cache. Thus all accesses to a descriptor occur in the same order—descriptor Kmap first, then the object Kmap. Inter-Kmap service requests are guaranteed to arrive in the order in which they were issued; hence it is not possible for two requests to interact in such a way that an outdated descriptor copy remains cached in the object Kmap.

In some ways, this scheme provides a more general addressing structure than the corresponding STAROS mechanisms. Not only can MEDUSA objects be moved (something that was never implemented in STAROS), but an object may be located in a different cluster from the descriptor list that contains its descriptor.

Deadlock Prevention through Resource Ordering. There are several known techniques for preventing deadlock in concurrent systems. Each of them works by denying one of the conditions necessary for deadlock. The MEDUSA microcode employs the notion of *resource ordering*, which uses a static ordering of all the resources used by the microcode. A microcode operation acquires resources in order: lower-numbered resources are acquired first and higher-numbered resources later. If, at some point, an operation finds that it needs a resource whose number is lower than that of a resource it holds, it must release the higher-numbered resources and try again to acquire the resources in the proper order. This strategy prevents deadlock by denying the *circular-wait* condition; it makes it impossible, for example, to have operation *A* waiting for resources held by operation *B* at the same time that *B* is waiting for resources held by operation *C*, which in turn is waiting for resources held by *A*.

As an example of how the microcode uses ordering to avoid deadlock, consider the allocation of Kmap contexts to a microcode operation such as intercluster memory references. Each reference requires two Pmap contexts—a *master* context in the Kmap from which the reference emanates and a *slave* context at the Kmap in whose cluster the object is located. Suppose that in each of two previously quiescent clusters, eight Cm's simultaneously make intercluster references to the other cluster. All eight contexts in each Kmap are allocated as masters, leaving no contexts available to service the intercluster requests as slaves. Deadlock!

MEDUSA statically divides Pmap contexts into master and slave groups. At the outset, an operation has no idea whether it will require one context or two, so it is randomly allocated a context from one of the two groups. If it acquires a slave context and later discovers that it needs another context, it is required to release the (higher-numbered) slave context and request a master context first, then a slave context later. Since all contexts follow this discipline, and since slave contexts are prohibited from attempting to acquire other contexts, deadlock over contexts is impossible.

Preventing Starvation. In most systems, starvation is a performance problem, preventing a process from being served in a reasonable amount of time. In Cm* it is a reliability problem as well because a Kmap context has no way of identifying the sender of a return message. It cannot recognize and discard messages from "orphan contexts" in other Kmaps—slave contexts that have been timed out under the assumption they have failed. If the master context has been freed and reallocated in the meantime, the return message could be misinterpreted as a return from another valid context, causing an error in the new master context [Sindhu 84]. Starvation must be prevented so that contexts are timed out only when they have, in fact, failed.

Medusa avoids starvation by never busy-waiting. Instead, activities and contexts wait explicitly and are serviced by disciplines that are known to be "fair." Where possible, this is done by ensuring that the service discipline is first-come first-served (FCFS). Utilities use the Kmap-provided synchronization operations (***Indivisible Increment*** and ***Indivisible Decrement***) to manipulate FCFS queues. Kmap operations do not require any explicit synchronization because the Pmap runs a context until that context voluntarily relinquishes it (Section 2.2.3); there are no microinterrupts.

It is not always convenient to implement FCFS service where efficiency is the utmost concern. This is the case in the most heavily used Kmap operations, nonlocal references. MEDUSA optimizes for the most common case, that of no contention for the target Cm. In this case, no other memory-reference request is waiting for a particular Slocal, either when a reference to that Slocal begins or when it finishes so queue-manipulation microinstructions can simply be skipped. The Kbus maintains 2 bits of state information for each Slocal: *busy*, which is *true* if the target Slocal is in use, and *hold*, which is *true* if another memory reference is waiting for the Slocal to become free. When service of a request begins, the queue is not manipulated if *busy* is false, and when the Slocal is released, it is bypassed when *hold* is false.[2] Although this does not help in the case of severe contention, it reduces the amount of queue manipulation considerably. STAROS uses a more sophisticated algorithm (Section 3.1.3), which performs somewhat better under heavy contention.

5.1.3. Amplification

Objects that are managed by a utility must be read/write protected from user activities but readable and writable by the utility in charge. For example, the *File System* must be able to update the pointers in a file control block whenever data is read or written. This raises two questions: How does the user activity pass the descriptor for the object to the utility? How does the utility use this descriptor to gain read/write access to the object? Unfortunately, an activity cannot simply pass a descriptor to a utility through the pipe it uses to invoke the utility (see Section 4.1.2). A message is simply a byte stream; the utility would have no way of guaranteeing that the user activity had not just forged the descriptor. Through its invocation pipe (to be described in Section 5.3.1), however, the utility *can* obtain the system name of the PDL of the invoker. The utility presents this name to the Kmap and requests that the PDL be mapped onto its *external descriptor list*, or XDL. The XDL is not a separate utility list but an *alias* that a utility can use to reference the descriptors in the invoker's PDL. The Kmap verifies that the requester is a utility, which it can do by

[2] This discipline is not completely FCFS because the *hold* bit is not set until the Kbus actually *attempts* a memory reference to a busy Slocal. Assume that one memory-reference request has encountered a busy Slocal. It is then placed in the Kbus's run queue. Now imagine that a second memory-reference request comes along before the Kbus attempts the first memory reference. If the Slocal has been freed in the meantime, the second memory reference will not find the Slocal busy and will be performed immediately, before the first memory-reference request. In practice. this anomaly occurs rarely and has not been observed to cause starvation.

interrogating a special bit in the requester's extended processor status word, and then establishes the new XDL.

Objects named by descriptors in the PDL of the invoker are now directly accessible. If the desired object resides instead in the SDL, then the utility must consult the *activity control block* (ACB) to discover the system name for the SDL. The utility reads the descriptor for the ACB from the PDL of the invoker, fetches the system name of the SDL, and presents it to the Kmap as before.

Now that the utility possesses a descriptor for the desired object, it must gain access to the object's memory. Since the object is not of type page, its memory is not ordinarily readable or writable. A utility, however, is able to invoke another Kmap operation to *amplify* its rights to the object. This gives it read / write access.

This simple mechanism greatly enhances reliability within the system because it permits access to utility-managed objects to be controlled on a "need-to-know" basis. User activities need not be given read / write access, which they could use to destroy the consistency of the objects. A utility need not be given permanent access to all the objects it is responsible for managing; this limits the damage that an errant utility can do.

MEDUSA is not the first operating system to provide protection for object types supported by the system. HYDRA [Wulf *et al.* 81] and CAP [Wilkes and Needham 79] are earlier examples. STAROS is structured in terms of modules (see Section 6.3) that are also capable of amplifying capabilities to gain access to the internal memory of objects. More recently, the Intel 432 [Organick 83] has provided a very flexible system of intermodule protection with two kinds of amplification. MEDUSA's utilities, however, play a more trusted role than modules in these other systems. A utility is given temporary access to a large part of the address space of the process that invoked it; it is assumed that it will not use this privilege to damage its caller. In addition, MEDUSA's scheme is less general: User task forces cannot protect objects that *they* implement from outside modification in the same way that utilities can.

The advantage of MEDUSA's approach is that the kernel has to provide less support for protection. This is reflected in the size of the kernel: MEDUSA's is only about 1K byte of LSI-11 object code plus about 4,000 eighty-bit words of microcode, compared with nearly 12K bytes plus 2,200 words of microcode for the STAROS nucleus and about 260K bytes for the HYDRA kernel, which provides much more flexible protection than either MEDUSA or STAROS.

5.2. Messages

Messages are a fundamental component of the MEDUSA operating system. They constitute the primary means of interprocess communication as well as the only method of requesting system services. It is important for message communication to be fast; otherwise the "system overhead" would be unacceptably high.

Messages are conveyed via pipes (see Section 5.1). There are two notable differences between MEDUSA pipes and Unix pipes, both of which have been made to improve reliability. First, it is important for the receiver to be able to determine the length of each message it receives. A mistake about the length of one message could easily cause subsequent messages to be misinterpreted. Consequently, MEDUSA's message-system microcode associates a byte count with each message. Each ***Receive*** is guaranteed to retrieve exactly the same amount of information that was sent, making it impossible for a malicious or errant sender to sabotage the receiver. Second, the microcode also places an indication of the sender's identity in each message. This enables an activity that is awaiting a message from another activity (such as a *return* from a utility invocation; see Section 5.3.1) to determine without a doubt when the desired message has arrived.

Allowing messages to contain descriptors as well as uninterpreted byte streams could enhance reliability still further. It would, for example, remove the need to make return pipes globally accessible. The need to make messages fast, however, effectively forces them into the microcode, where the need to handle descriptors would impose additional complexity. It is more convenient to implement descriptor operations in the software of the *Memory Manager* utility, where it is easier to provide a high level of functionality.

The ***Send*** and ***Receive*** operations are fully symmetric: A ***Send*** may be performed unless the target pipe is full; a ***Receive*** can be performed unless the pipe is empty. An activity can choose between two forms of ***Send*** and ***Receive***. If it chooses a *conditional* operation, the activity is notified whether the operation can be performed. If it chooses an *unconditional* operation, it is suspended until the activity can be completed and is then reawakened. This differs from the STAROS message system, in which a receiver performing a *registered* (unconditional) ***Receive*** can choose either to suspend or to continue execution and be notified of the operation's completion by the setting of an event variable. Alternatively, STAROS provides no unconditional ***Send*** operation.

The designers of MEDUSA expected waiting to be a common occurrence, and they optimized for it in several ways. Unconditional ***Send***s and ***Receive***s eliminate the need for busy-waiting, and ***Multi-Event Wait***s remove the need for polling. Both these operations help to conserve processor resources. Unnecessary copying of messages is also avoided. As detailed in Section 4.1.2, if a message is sent when its receiver is waiting, the message is copied directly into the address space of the receiver, bypassing the pipe altogether.

Another serious manifestation of overhead in message communication is its tendency to induce spurious process switches (or "context swaps"). Suppose that one activity is periodically sending messages to another. Unless the receiver consumes the messages exactly as fast as the sender produces them, the faster of the two will have to wait for the slower on each iteration. The faster activity will be dislodged from its processor, and another activity will take its place. Then when the message arrives, the waiting activity will regain its processor. The two context swaps can easily add an order of magnitude to the time required for message communication; the time needed to set up and transfer the message is insignificant by comparison. This phenomenon effectively prevents fine-grain interaction between the two processes.

To circumvent this problem, MEDUSA uses a simple heuristic. When an activity performs an unconditional message operation, it is allowed to specify a small *pause* *time*. During this time, the processor assigned to the activity executes an idle loop in kernel space, with the waiting activity's process state still assigned to the processor. If a message arrives during this time period, the waiting activity can be resumed immediately; if not, it is suspended as explained above. The faster process is, in effect, slowed down to match the speed of the slower.

Statistics on the distribution of pause times are gathered almost automatically by the MEDUSA kernel. Each activity control block contains a set of counters that form a histogram of pause times. When the kernel enters its idle loop during a pause, it initializes a counter to the pause time and then repeatedly decrements it. If the activity becomes runnable before the count reaches zero, the kernel saves this counter, and, at the beginning of the next pause time, increments the appropriate counter in the histogram.

The pause-time histogram can be used to help in load balancing, particularly by utilities [Ousterhout 80]. If most of the pause times are near, or equal to, zero, there is almost always work for the utility to perform, so it may well be overloaded and probably should create a new activity to help it. A preponderance of large pause times, however, indicates either that (a) the activity is underloaded and should consider allocating its work to other activities and deleting itself or that (b) messages are buffered in invocation pipes, but the activity is not currently waiting on those pipes. (When no more activation records remain for a pipe, the activity ceases to wait on that pipe, so requests placed in the pipe will remain there until coroutines are able to service them.) This is due to a dearth of activation records, so the proper response is to create more activation records.

The ***Send*** and ***Receive*** operations in MEDUSA are *overloaded*: They apply to several object types besides pipes. The implementation of overloading is closely related to the concept of an *event*. For a pipe, an event is the presence of a message in its buffer. An object that has an associated event is called *eventable*. A ***Receive*** operation tests for or awaits the occurrence of an event. For a semaphore, an event is a positive value; thus a ***Receive*** on a semaphore has the same semantics as a *P* operation, and a ***Send*** implements a *V*. Not all eventable objects are implemented in microcode. For example, for a file control block, an event is the presence of data in its buffer.

5.3. Utilities

Utilities are the method MEDUSA uses to deliver operating-system services. Each utility is structured as a task force and exports a particular abstraction to the rest of the system. There are five utilities, providing the services of memory allocation, file and device I / O, task-force management, exception handling, and debugging.

The functions of the *Memory Manager* include allocating and deallocating primary memory, maintaining descriptor lists, and implementing operations on page, pipe, and semaphore objects. When an activity needs to create a new object, it requests the *Memory Manager* to allocate memory for it. If the object is a page, pipe, or semaphore, the *Memory Manager* initializes its memory; otherwise initialization is the responsibility of the invoking activity. Working in conjunction with the Kmap microcode, the *Memory Manager* maintains descriptor lists by creating new descriptor lists; creating, copying, and deleting descriptors; and maintaining a count of the outstanding descriptors for each object. When all the descriptors for an object have been deleted, the *Memory Manager* returns the object's storage to the free list.

The *File System* provides functionality similar to the Unix file system but extended in several ways. The directory structure is hierarchical, but it allows several "current directories" instead of the one permitted by Unix. While Unix allows a maximum of 16 files open simultaneously, MEDUSA lets any descriptor slot in an activity's descriptor list hold a file descriptor so that the number of open files is limited only by the length of the descriptor lists. Files on disk contain redundant information to facilitate recovery from crashes. The strategy is similar to that of the ALTO file system [Lampson and Sproull 79]. Files are stored in blocks of fixed length. Into each block is written a label containing the name of the file to which the block belongs, the number of the block within the file, and pointers to predecessor and successor blocks. When directory information has been corrupted, it can be restored from the labels, and vice versa.

Among the *Task Force Manager*'s duties are the creation, scheduling, and deletion of task forces. Any existing task force may request creation of a new one by passing the name of a task-force description file to the *Task Force Manager*, supplying either descriptors or ordinary data as parameters for the new force. As the owner of the new task force, the old task force obtains a descriptor for the new task force, allowing it to start, halt, restart, and single-cycle the task force, as well as to delete activities selectively. In fact, a complete debugger has been written whose only special privilege is ownership of the task forces that it debugs. The *Task Force Manager* also is responsible for *coscheduling* a task force's activities, as Section 5.5 will describe.

Whenever an exceptional condition is detected, either by the hardware, the microcode, or the utilities, the *Exception Manager* springs into action, deciding which entity is to be notified of the exception and then delivering a report to the proper authority. The other function of the *Exception Manager* is to attempt to recover from the exception. It contains hardware-level handlers for all exceptions and software handlers for certain utility exceptions that might cause deadlock if implemented elsewhere. Centralizing exception management in a single place serves to encapsulate code that might otherwise be hard to test exhaustively and hence would be error prone. It also lets user programs invoke the same powerful reporting and recovery facilities employed by the utilities.

The MACE utility provides MEDUSA with debugging and measurement capability. It consists of a single activity and runs on a dedicated Cm, where all its resources are allocated. It possesses utility privileges, as described below, and communicates via the standard MEDUSA message system. Because it is essentially independent of the functioning of other activities, it is not compromised by errors in them, nor does it unduly perturb their performance.

5.3.1. Utility Calls

In general, utilities communicate with the outside world by means of messages. A utility is invoked by a ***Send*** to one of its *invocation pipes*. As a side effect of that ***Send***, the invoking activity is suspended, to be reactivated when the utility returns. The invocation message contains arguments for the utility and an indication of a *return pipe* through which the utility may pass back results. This return pipe is the counterpart of a return address in an ordinary procedure call. Thus a utility call is effectively a call by value-result.

Descriptors for invocation pipes for all utilities are publicly available in a *utility descriptor list*, or UDL, located in the memory of each Cm. The mapping of UDL slots to utilities is the same on all processors. A UDL is analogous to the trap vector in a traditional operating system.

This method of providing access to utilities enhances fault-tolerance because damage to a UDL affects only the ability of one processor to communicate with the operating system. In addition, the UDLs of different processors may direct requests for the same function to different activities of a utility task force. This makes it easy to distribute the utilities physically. It also means that when a utility activity gets too busy, the system can respond by replacing the UDL entry for its invocation pipes by those of a less heavily loaded sibling. The utility may even spawn a new activity and distribute descriptors for its invocation pipes to several UDLs.

Since messages consist of uninterpreted byte streams, it is not possible for the invoker of a utility to pass a descriptor for the return pipe in its invocation message. When one utility invokes another, it passes a *transaction identifier* instead. The transaction identifier tells which slot in the UDL contains a descriptor for the return pipe. Utilities are trusted not to misuse the return pipes, but return pipe indices are encoded in such a way that an errant utility is unlikely to generate a legal one by accident.

As an additional check, the invoked utility passes the transaction identifier back to the utility that called it, so that the first utility can verify the authenticity of the return message. The transaction identifier also distinguishes between return messages if the calling utility has more than one utility call outstanding simultaneously.

If the invoker is a user activity, the return pipe must be specified in a more secure fashion. The user process must not be permitted to "lie" about its identity in order to cause the return message to be directed to someone else. To forestall this, the message-system microcode makes the identity of the sender available to the receiver by providing the system name of its PDL. The invocation message thus contains the index of the return pipe in the invoker's PDL or SDL (since the SDL is accessible from the PDL via the activity control block).

As mentioned in Section 4.1.2, a user activity invoking a utility is required to wait until the utility call returns before continuing execution. The same restriction does not apply, however, to utility calls invoked by other utilities. While it is prudent to limit the rate at which user activities can generate utility calls, it is not wise to limit the rate at which utilities can service these calls. Hence utilities are allowed to use a special microcode operation to perform a utility call that does not block the invoker.

5.3.2. Deadlock Avoidance

Although it might not be apparent at first glance, the activities of a utility cannot be allowed to handle service requests at random, or deadlock might result. Consider this example from Ousterhout [Ousterhout 80]. An activity requests the *File System* to open a file. Then the following series of events occurs (Figure 5-2):

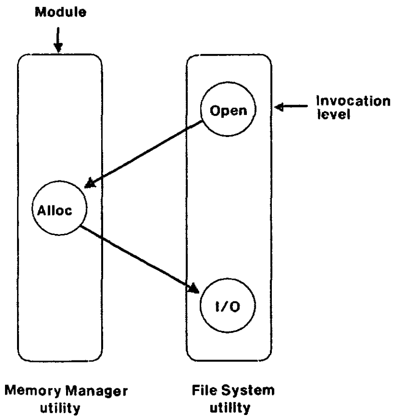
- The *File System* receives the request and attempts to open the file. It must allocate a new file control block (FCB).
- The *File System* requests the *Memory Manager* to reserve memory for the FCB. No free block of memory is large enough, however, so some object must be swapped out.
- The *Memory Manager* requests the *File System* to swap an object out to secondary storage.

Assuming there is only one *File System* activity, deadlock has now occurred. The *File System* is waiting for the *Memory Manager* to deliver memory, and the *Memory Manager* is waiting for the *File System* to swap an object out. Even with more than one *File System* activity, an unhappy timing of service requests can still produce deadlock. Some strategy is needed to avoid this prospect.

The crux of the problem is that there is a circularity in the dependencies between modules. As in this example, a circularity of dependencies can occur even if invocations are hierarchical (i.e., if each invocation is directed to a routine at a lower level of the system [Habermann *et al* 76].)

To avoid deadlock, MEDUSA introduces the concept of *service classes*. A service class is defined as the set of functions provided by a particular utility at a single invo-

Figure 5-2 A Hierarchical Sequence of Calls that Can Produce Deadlock



cation level. For example, the *File System*'s ***Open File*** and ***Close File*** operations fall into one service class, I/O functions make up a second class, and memory allocation and deallocation functions make up a third. Each service class is statically allocated a separate set of resources, including memory, which is sufficient to allow it to carry out its functions.

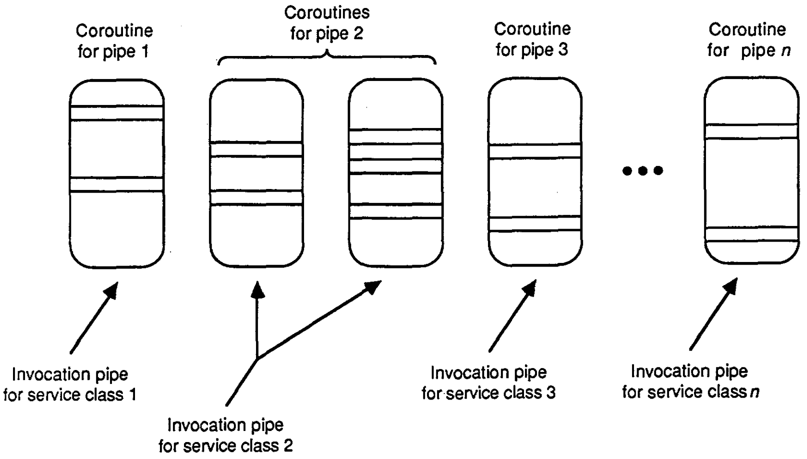
Activities are another resource that must be distributed among the service classes. If all the activities of a utility attempted to service requests from the same class, none would be left to handle any lower-level requests for the utility that might ensue. One solution would be to allocate separate activities of the utility to each service class, but in MEDUSA, this would not be an efficient use of resources, since some service classes are invoked very infrequently.

Instead, MEDUSA uses a strategy reminiscent of Kmap contexts (Section 2.2.2): Different service classes are served by different coroutines of the same activity. Each coroutine is statically allocated its own stack. A utility has one invocation pipe for each service class (Figure 5-3). The coroutines are statically associated with different invocation pipes so that a heavy demand for one service class does not impede the utility from handling requests for other service classes.

When a message arrives from an invocation pipe, one of that pipe's coroutines is allocated to service the request. Upon completion of the service, the coroutine is returned to the free pool to await another request. In addition to avoiding deadlock, the coroutine scheme also has the effect of increasing the throughput of the utilities, since an activity which is waiting for service from another activity can execute another coroutine in the interim.

The coroutine structure of utilities imposes little additional complexity. In particular, synchronization constraints are almost unaffected because utilities already

Figure 5-3 The Coroutine Structure of a MEDUSA Activity


Coroutine for pipe 1
Coroutines for pipe 2
Coroutine for pipe 3
Coroutine for pipe *n*
Invocation pipe for service class 1
Invocation pipe for service class 2
Invocation pipe for service class 3
Invocation pipe for service class *n*

contain potentially concurrent activities. The only real limitation is that no coroutine may be allowed to busy-wait on a lock, since that would deny other coroutines of the same activity the opportunity to release it.

5.3.3. Multiplexing Utilities

When a utility finishes processing a service request, it executes a ***Multi-Event Wait*** (see Section 4.1.2) on all its invocation pipes and its return pipe. If no message is present in any pipe, the utility remains suspended until one arrives. If the message arrives through an invocation pipe, a coroutine is allocated to service the message, and execution begins at the coroutine's start address. If the message arrives via the return pipe, the transaction identifier in the return message is verified, and execution of the coroutine resumes at the point where it was suspended.

In effect, the transaction identifier takes the place of a capability in authenticating the return message. Conversely, in StarOS, when a module is invoked, a capability for a return mailbox is passed as a parameter. In contrast to Medusa, where return pipes are globally available to all utilities, the StarOS module has no foreknowledge of the return mailbox. Its possession of a capability for the mailbox, however, is prima facie evidence of its authority to return to the return mailbox's owner.

5.3.4. Robustness Considerations

The functionality that we have described thus far is included in the standard version of Medusa the version used in the experiments reported later in this book. There is another, experimental version of Medusa whose utilities have been enhanced to provide still greater ability to tolerate and recover from errors [Sindhu 84]. In this version, all utilities contain at least two activities, which are constrained to execute in separate clusters. As in the standard system, each activity is capable of providing all the services furnished by the utility.

This organization provides a good deal of redundancy. There are at least two copies of all code and nonvolatile data and at least two activities that can provide any given functionality. Because the activities are identical, it is easy for one activity to use its copies to rescue the other activity from errors in its code or data.

Despite the high degree of replication, the shared objects in the utility's SDL still constitute a *singularity*, or single point of failure. Medusa copes with this problem by duplicating critical objects and caching noncritical objects or distributing them around the system (Figure 5-4) so that a single failure affects only a small part.

The critical objects are those items needed by the utility to function; counted among them are the SDL itself, the description of the task-force state in the *task-force control block* (TFCB), and a few pages containing data needed to restart failed utility operations. Two copies of these objects are maintained in different clusters to protect them from faults that are confined to a single cluster. The extra copy of an object is known as a *co-object*. There are two varieties of co-object (Figure 5-5):

Figure 5-4 The Structure of a Utility Task Force

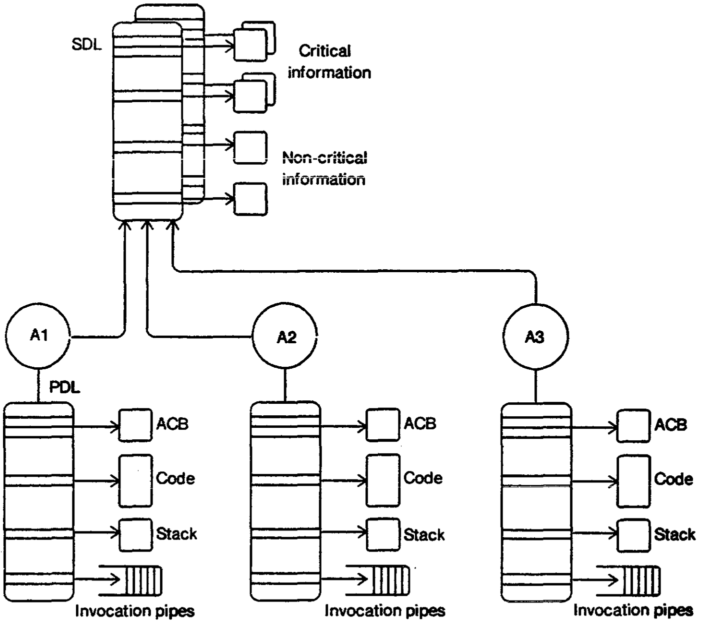
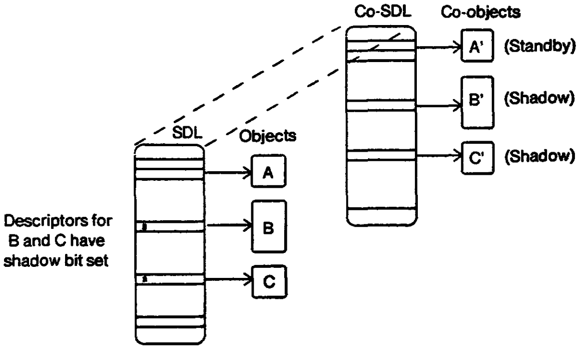


Figure 5-5 Co-Objects—the Mechanism for Object Replication



- *shadows*, which are always identical to the other object because all writes are directed to both an object and its shadow.
- *standby* objects, which are copies made from the object at some earlier point in time.

Descriptors for co-objects are kept in a special descriptor list called the *co-SDL*. Each such descriptor contains a bit specifying whether the co-object is a shadow or a standby and may be toggled from one to the other by the activities of the task force.

Writes to co-objects are implemented in Kmap microcode. When a Kmap encounters a nonlocal write reference, it checks the cached copy of the descriptor to see whether the *shadow bit* is set (Figure 5-6). If the shadow bit is set, the object is being shadowed. If the write to the primary object succeeds, the corresponding word of the shadow is also written. The shadow is located through its descriptor in the co-SDL; the offset is the same as the offset of the descriptor for the primary object in the SDL. Descriptors for co-objects can be cached just like descriptors for primary objects—the first reference causes the descriptor to be fetched from main memory; subsequent references can proceed apace.

Measurements performed by Sindhu indicate that the cost of shadowing is quite acceptable (Table 5-1). A replicated write usually takes 42.7 μs., which is five times as long as an ordinary write (Table 3-2). In three utilities, however, the fraction of memory references that were writes to shared objects was never greater than 0.53 (Table 5-2), so the overhead is only about 2 percent.

It is straightforward to recover from an error using a shadow. When an error is detected in an object, the error handler for that object type reads the other copy of the errant word or words from the shadow and writes them into the corresponding locations in the primary object.

Section 5.3.2 discussed the problem of deadlock in normal utility calls. Recovery from exceptions raises still other possibilities for deadlock, if the recovery procedures use parts of the system that have been damaged. Sindhu [Sindhu 84] has identified three such cases:

Figure 5-6 The Implementation of Shadows

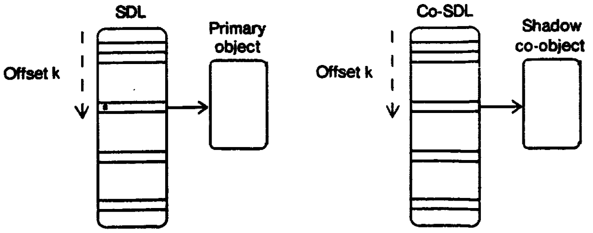


Table 5-1 The Cost of Replicated Writes

	Reference time (μs.)		
Configuration	Primary part	Shadow part	Total
Primary in cluster but nonlocal Shadow in cluster but nonlocal	12.1	13.9	26.0
Primary in cluster but nonlocal Shadow in nonlocal cluster	12.1	30.6	42.7

Table 5-2 Frequency of Reads and Writes (in Percentages) to Different Classes of Objects

	Code		Local data (stack)		Shared data		Fraction of mem. shared between
Utility	Writes	Reads & writes	Writes	Reads & writes	Writes	Reads & writes	
Memory Mgr.	0.0	65.4	13.0	28.4	0.33	0.85	0.20
Task Force Mgr.	0.0	71.5	10.7	24.3	0.03	0.08	0.15
File System	0.0	63.9	11.6	27.4	0.53	2.90	0.27
*Exception Mgr.**							0.13

*Measurements for the *Exception Manager* and MACE have been excluded because they do not operate in the normal case and thus have little effect on performance. Values for shared data include only references to utility-owned objects, not to objects passed as parameters.

- Corruption of a shared object may disable all the activities of a utility. If the utility is invoked during error recovery, deadlock occurs.
- If a private, unshared object is corrupted, deadlock can result if the activity whose memory was damaged is invoked during the course of recovery.
- At any time, several activities that have made calls to a utility may be blocked waiting for the utility to return. These activities may have objects locked. If recovery requires use of these objects, deadlock occurs.

The first case is handled using shadows or standby objects, depending on whether a slightly out-of-date version of the damaged object will suffice for recovery purposes. Using standby objects costs less but cannot always allow successful recovery. In either case, the *Exception Manager* copies the descriptor for the co-object into the SDL so that the (presumably) undamaged copy of the object is used during recovery.

Another approach is needed in the case of a damaged private object, as private objects are not replicated. The activities of all utilities are divided into two or more groups, with each group including at least one activity from each utility. (Recall that every utility activity is capable of providing all the services of the utility.) As shown in

Figure 5-7 The Cross-Utility Communication Structure

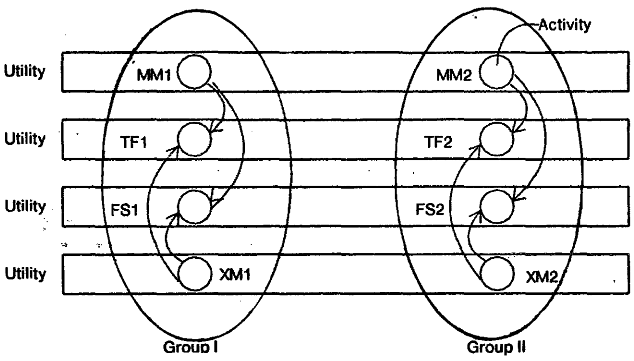


Figure 5-7, calls to utilities other than the *Exception Manager* are generally directed within the group (e.g., *Memory Manager* activity 1 calls *Task Force Manager* activity 1). A call *to* the *Exception Manager* is always directed to the next higher group (or to group 1, if it emanates from the highest-numbered group). Consequently, a sequence of utility calls beginning at the *Exception Manager* will steer clear of the damaged activity.

Neither of the strategies just described will take care of a situation where recovery must use an object that is undamaged but is *locked* by an activity that remains blocked while recovery is in progress. In this case, one solution is to prohibit additional utility calls for the duration of recovery and then to abort and restart all calls affected by the error. MEDUSA uses a heuristic approach: It simply waits a while to give calls that are not affected by the error a chance to complete. This is called the *closed-system recovery prelude* and is invoked whenever recovery requires the use of other utilities. It allows most blocked calls to finish, provided that they are blocked for reasons unrelated to the damage. Any calls that remain blocked are then aborted. This means that an unusually long but undamaged call is occasionally aborted, but this is a small price to pay because the call can be retried anyway. Once active computations have been flushed, calls during recovery can acquire locks without risking deadlock.

5.4. Exceptions

An *exception* is an unusual occurrence during program execution that may or may not be an error. (A more formal treatment may be found in [Levin 77].) The detection of an exception raises two questions: Who should be told about it, and what should be done about it? The first question leads to a consideration of *exception reporting*, the second, to a discussion of *exception handling*. Three issues are raised: the detection, reporting, and handling of exceptions. Each will be examined in turn.

5.4.1. Detection

Both the microcode and software of MEDUSA have well-developed strategies for detecting exceptions. Approximately one-tenth of the microinstructions are devoted to consistency checks. Common portions of exception-handling code have been structured as in-line procedures to limit the amount of microcode that needs to be tested.

A microcode exception may be detected by either a master or a slave context. If it is detected by a slave, and the slave is unable to handle it, it releases its resources and reports back to its master, which in turn reports the error to the invoking activity. If an exceptional condition occurs that cannot be handled by a master, the master aborts any slave acting on its behalf and reports the error to the invoker. The master thus serves as a clearinghouse for reporting exceptions to the invoker. Many exceptions detected by microcode—in addition to those detected by software—are handled in software because the limited memory of the control store is best devoted to operations that will be invoked frequently.

The utilities also contain several mechanisms for error detection. Transaction identifiers (Section 5.3) are a case in point, as are the block labels used by the *File System*. Whenever a disk block is read in, the label is checked against the expected value, as determined by the directory structure. Finally, backpointers (Section 5.1.1) are checked against descriptors whenever the *Memory Manager* accesses its allocation tables.

5.4.2. Reporting

Consider what happens when the hardware detects a parity error in a memory page. Obviously, the activity that was executing should be notified. But it is also prudent to notify other activities that hold descriptors for the failing page so that they may defend themselves, perhaps by attempting to load a new copy before continuing execution. MEDUSA uses the terms *internal report* to refer to the notification of the activity (known as the *victim*) whose explicit action uncovered the exception and *external report* for the notification of other affected activities. An internal report is always made; an external report is made only if the error affected a shared abstraction. Internal reports may be made either *in-line* or *out-of-line*, as detailed below. External reports also can be divided into two categories—*buddy reports* and *parent reports*—on the basis of where they are directed.

Internal Reports. If an internal report is made in-line, no special action is taken. Information about the exception is written into the victim's ACB, but the victim's execution state is not affected, so the victim may continue execution as though the exception had not occurred. Consequently, it is the activity's responsibility to check (in-line) for an occurrence of the exception. In-line handlers are employed primarily by the *Exception Manager* itself, which cannot be allowed to generate exceptions that it must report itself.

Out-of-line reports are designed to implement the exception-handling constructs provided by most modern high-level languages. The activity provides the address of the exception handler to the *Exception Manager* in advance. When an exception is detected, the *Exception Manager* simulates an interrupt and causes the activity to continue execution at the handler's start address.

External Reports. As noted above, an internal report is made for every exception. Sometimes it is possible for the victim to recover from an exception by itself, but at other times, the assistance of another activity is needed. In this case, an external report is the vehicle for notifying the other activity, which may be either a *buddy* or a *parent*, as explained below. To deliver an external report, it is not sufficient simply to interrupt the other activity; information about the exception also must be conveyed. Besides, another report might arrive while the first one is being handled; generating another "interrupt" would impose a last-come first-served discipline, whereas FCFS service would be fairer and immune to starvation.

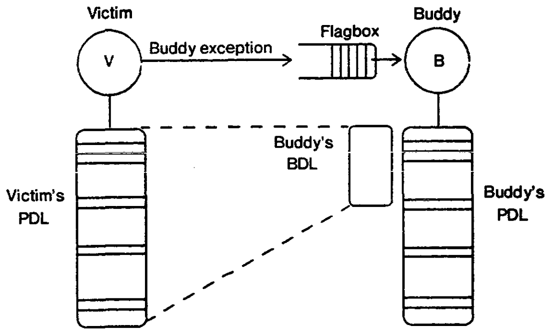
MEDUSA's approach is to queue up external reports in *flagboxes*. Each activity has a flagbox; each external report in a flagbox is called a *flag*. When the *Exception Manager* delivers a flag, it interrupts the flagbox's owner, which then runs its flagbox interrupt handler, performing whatever recovery action is appropriate. If the handler is already running when the report arrives, the report is merely placed in the flagbox without interrupting the activity. If the flagbox is nonempty when the handler finishes, the activity is reinterrupted.

Flagboxes are used to report two kinds of exceptions, known as *object exceptions* and *buddy exceptions*. An object exception notifies an activity of an error in a shared object. A buddy exception is used to make a buddy report, which requests that an exception be handled by another activity in the same task force as the victim. This activity is called a *buddy*.

An activity may designate a buddy to handle specific exceptions for it. The buddy must be reasonably independent of the victim so that it is unlikely to encounter the same exception. The buddy must also be sufficiently knowledgeable about the victim to take intelligent action on its behalf. For example, the corruption of one of an activity's code pages is an exception from which it might be difficult for an activity to recover, since its recovery code could be affected. Another identical activity, however, would have no difficulty copying one of its code pages into the address space of the victim. To do this, it must have access to the victim's PDL. When a buddy handler is invoked, the microcode gives it access to the PDL of the victim; this descriptor list is known to the buddy as the *buddy descriptor list*, or BDL (Figure 5-8). After handling the exception, the buddy invokes the *Exception Manager*, which continues the victim and invalidates the BDL.

The last kind of external report is known as a *parent report*. When a task force does not provide a handler for a particular exception, an event describing the exception is entered in the activity's task-force control block. It becomes the responsibility of some activity with a descriptor for the TFCB (called the parent) to handle the exception. Because the parent activity usually lacks detailed knowledge of the functioning of its children, it may be limited to generic actions, such as aborting a child or restarting it. The extensive privileges available via the TFCB descriptor, however, permit parent reporting to be exploited profitably by a debugger.

Figure 5-8 The Buddy Mechanism for Reporting and Handling Exceptions



5.4.3. *Handling*

MEDUSA differentiates between *coarse-* and *fine-grain* exception handling. A fine-grain exception handler provides recovery from a small set of closely related errors. A coarse-grain exception handler is invoked when a fine-grain handler fails or when none exists for a particular exception. A particular coroutine, activity, or microcode subsystem may have several fine-grain handlers, but it has only a single coarse-grain handler. The coarse-grain handler may have to "unwind" and retry a much larger amount of computation than a successful fine-grain handler. To simplify the coding of coarse-grain handlers, a closed-system recovery prelude is performed whenever one is invoked.

For example, the utility *activity coarse-grain handler* [Sindhu 84] is located in the exception manager. It uses the redundant information in the victim's buddy to restore the state of the computation. It assumes that any part of the victim might be damaged and therefore checks all its memory. Its execution is divided into four stages:

Phase I: Structure. Checks whether the victim contains the same number and type of objects as its buddy and whether those objects are structurally consistent. If there are discrepancies, it performs repairs.

Phase II: Private information. Compares the victim's private objects with the buddy's to make sure the information is intact. Corrects any errors.

Phase III: Shared information. Compares the shared objects with their co-objects and corrects any differences.

Phase IV: Execution state. Restarts the victim by resetting its execution state to a default initial value.

If any of these steps fails, the buddy aborts the victim and clones itself to create a new utility activity.

Since recovery often requires partially completed operations to be abandoned and restarted, a key question is how computation can be undone. One method is to include code in each activity that "backs up" its state at convenient intervals. This approach is quite complicated in the case of utilities because a utility may execute concurrently with other utilities that it has called (see the last paragraph of Section 5.3.1). A better approach is to provide a centralized facility for undoing the effect of writes and completed cross-utility calls. Such a system has been designed (but not implemented) for MEDUSA.[3] It divides utility operations into *execute* and *commit* phases and has much in common with the two-phase commit protocols used in distributed databases. The performance degradation imposed on utility calls by this protocol was estimated by Sindhu [Sindhu 84] as 20 to 28 percent, depending on whether the invocation made any cross-utility calls.

5.5. Coscheduling

MEDUSA's efforts at optimizing message operations have been aimed at making it possible for activities to interact with each other rapidly. Yet the best the message system can do is to ensure that an activity is not thrown off a processor prematurely. Another mechanism is needed to make sure that the activity has been scheduled on a processor in the first place. This is not a problem if processors outnumber activities in the system, but if there are not enough processors to schedule every runnable activity, the scheduler must pick and choose judiciously so that processes that are likely to interact can do so with a minimum of delay.

Consider, for example, a task force that solves partial differential equations by dividing up the grid among several activities (see Section A.1). Suppose that synchronization is required on every iteration. If even one of these activities is descheduled while the rest are scheduled, all the other activities will have to wait for it. This slows down the task force tremendously and may even cause the other activities to lose their processors.

We can define the *activity working set* analogously to a working set of pages—as the minimum number of activities that must be *coscheduled* (scheduled simultaneously) on processors for the task force to make acceptable progress. In this example, all the activities that participate in solving the equation are members of the activity working set. The master process (see Figure 4-1), and any processes that communicate with terminals or other peripherals, probably are not members. Similarly, *activity thrashing* occurs when the scheduling of processes whose services are required induces the descheduling of other processes whose services will soon be needed.

Since the task force is the basic group of cooperating processes, we made it the unit of coscheduling. Consequently, when a task force is coscheduled, all its activities must be assigned to processors. There is less flexibility in processor assignment than in page assignment, however, because each process must be scheduled on the Cm where its code resides, while a page may be placed in any page frame. Thus working-set results cannot automatically be translated to statements about

[3] It is too involved to be described here; the details may be found in [Sindhu 84].

coscheduling. Investigation of new algorithms is necessary. Ousterhout [Ousterhout 80, Ousterhout 82] studied three such algorithms by simulation.

An ideal coscheduling algorithm would not require processors to fritter away time on activities of task forces not currently coscheduled. In other words, the ideal algorithm would always be able to run coscheduled activities on each processor that had runnable (nonblocked activities). The ideal would be achievable in a system with, say, 50 processors where each task force consisted of 25 activities. Unfortunately, the ideal is not attainable in a practical system, where task forces consist of differing numbers of activities. The most effective of the three coscheduling algorithms, the *undivided algorithm*, usually managed to keep 60 to 70 percent of its processors busy running coscheduled activities, even under conditions of high system load. It is adversely affected, however, by increasing system load and increasing task-force size, relative to the number of processors. Performance is less sensitive to the frequency with which processes are blocked, but the best results still are obtained when most processes are runnable. A complete description of the algorithms and results is presented in Appendix B.

5.6. Summary

In its structure, the MEDUSA operating system reflects the distributed architecture of Cm*. MEDUSA consists of two major components: the kernel, which includes both Kmap microcode and LSI-11 software, and the utilities, which have special privileges and deliver specific operating-system services. The kernel is responsible for interrupt handling, managing and amplifying descriptors, and interprocess synchronization and communication. The kernel addresses several issues encountered in object-oriented parallel operating systems, such as sharing and protection of objects, coherence of shared descriptor caches, and starvation and deadlock over shared resources (e.g., Kmap contexts).

The five MEDUSA utilities are each structured as a task force:

1. The functions of the *Memory Manager* include allocating and deallocating primary memory, maintaining descriptor lists, and manipulating page, pipe, and semaphore objects with the aid of the Kmap microcode.
2. The *File System* is functionally similar to the Unix file system with somewhat extended capabilities.
3. Among the *Task Force Manager* functions are creation, scheduling, and deletion of task forces.
4. Whenever an exceptional condition arises at any level of hardware, firmware or utility, the *Exception Manager* is invoked to decide where and when to report the abnormal condition. The other function of the *Exception Manager* is to attempt recovery from the exception.
5. The MACE debugging and measuring utility consists of a single activity running on a dedicated Cm.

Aside from performance and correctness concerns, an operating system for a parallel processor also should address the issue of robustness. An experimental version of MEDUSA attempted to determine the weak points in the reliability of the operating system and to modify the system to increase robustness. Many such modifications were implemented, and their cost was estimated. All utilities are replicated, and co-objects are maintained to provide extra copies of critical code and data. There are two types of co-objects—shadows and standby objects. The shadow is a "hot" copy of the object, continuously updated; the standby is a copy of the object made at some earlier time. The overhead for shadowing has been measured at no more than 2 percent.

The robustness of MEDUSA is enhanced by another type of redundancy known as a buddy activity. When a "victim" activity encounters an exception, its structurally identical buddy is capable of handling the exception. After an exception has been detected and isolated, a recovery process should take place. Write operations may have to be undone to bring the system to a consistent state. The experimental design divided utility operations into *execute* and *commit* phases and provided a centralized facility for undoing the effect of writes. The predicted performance degradation was between 20 and 29 percent.

From the perspective of five years of experience with MEDUSA, the following observations can be made:

- MEDUSA was the more popular of the two operating systems with novice users because fewer concepts had to be learned than in the case of STAROS.
- The division of a distributed system into a small kernel and utilities was a pioneering effort in the development of parallel operating systems and local area networks.
- Other features, such as object orientation and robustness techniques, even if not yet incorporated into other systems, have a great potential for the future development of operating systems.

The next chapter will present the STAROS operating system and show how it exploits the Cm* hardware in a different manner than MEDUSA.

6. STAROS

A familiarity with the MEDUSA operating system and its relation to Cm* is a good foundation for discussing STAROS, since it presents abstractions that are a level above those found in MEDUSA. To give the reader a good feel for the similarities and differences between the two systems, our description of STAROS will closely parallel the discussion of MEDUSA in Chapter 5, although the emphasis will vary slightly, reflecting the most interesting and highly developed aspects of the design.

The STAROS operating system is composed of two levels—the *Nucleus* and user-level software, which consists entirely of STAROS *modules*. These modules provide many operating-system services that, in MEDUSA, are provided by utilities. It is a testimony to the flexibility of protection in STAROS that most of the operating system requires no greater privileges than user software.

The *Nucleus* is composed of the Kmap microcode, together with some additional software. Like the MEDUSA kernel, a separate copy of the software *Nucleus* resides on each Cm. The whole *Nucleus* amounts to fewer than 12K bytes. It fits in three 4K-byte pages, one of which is used only for I/O. The microcode comprises 2,200 eighty-bit instructions. *Nucleus* functions have been implemented in microcode if they inherently involve multiple Cm's or if they need to be especially fast. Mapped memory references and capability operations are examples. Functions that have neither of these characteristics are performed by *Nucleus* software.

A STAROS module is a static entity from which dynamic processes may be created, or *instantiated*. Like an Ada package, the module exports a certain number of *functions* to the outside world; each time a function is invoked, it is executed by one of the processes instantiated by the module. STAROS modules usually communicate via messages, but the generality of capability addressing enables memory to be shared among any number of processes, whether they belong to the same module or not.

The first two sections of this chapter focus on the STAROS *Nucleus*. Section 6.1 describes objects, repositories for all information in the system, and the capabilities that are used to reference them. Section 6.2 relates the philosophy and mechanics of the STAROS message system. Sections 6.3 and 6.4 concentrate on STAROS modules. Section 6.3 introduces modules and the functions they provide. It explains when processes are created at a function call. Section 6.4 introduces the STAROS facilities for error detection. Section 6.5 describes scheduling in STAROS, which follows the principle of policy/mechanism separation. In STAROS, storage is reclaimed by garbage collection; Sections 6.6 and 6.7 explore how this is accomplished and how much it costs.

6.1. Facilities Provided by the *Nucleus*

The StarOS *Nucleus* is implemented partially in Kmap microcode and partially in LSI-11 software. Two classes of functions have been placed in the Kmap microcode:

Functions that the architecture forces to be there. References by the LSI-11s to nonlocal memory and other aspects of interprocessor communication must necessarily be performed by the Kmaps because the only data paths between LSI-11s go through one or more Kmaps.

Functions that must be performed efficiently. Capability operations and the operations on certain object types, such as stacks and deques, fall into this class. Implementing a function in microcode often will speed it up by a factor of 10 to 20, though precise comparisons are difficult to make because of the dissimilarity of the instruction sets of the LSI-11 and the Kmap.

Two other classes of functions have been placed in the *Nucleus* software, which is replicated in each Cm:

Functions that involve management of the processor resource. Trap and interrupt handling must be performed by software because the Kmap cannot access all of the processor state necessary to perform these functions. For the same reason, any operation that requires stopping one process and starting another requires activity by the *Nucleus* software.

Functions that operate on the executing process. Process operations such as ***Block*** and ***Terminate*** change the state of the executing process. The result of these functions may determine whether the process should continue to run. These functions can be performed expeditiously by the *Nucleus* process assigned to the same processor.

Our examination of the *Nucleus* begins with the microcoded portion, which is largely concerned with providing the abstractions required by the object model.

6.1.1. Objects

Objects are the basic building blocks of StarOS. Each set of information in memory—whether it is a code segment, an array, a mailbox, a list of capabilities, or something else—is contained within some object. Each object has a specific set of functions that can be applied to it, and every action performed by StarOS is the application of some function to an object.

For example, the action of reading and writing words in an ordinary data array consists of applying the ***Read*** and ***Write*** instructions to a basic object that contains the array. Sending a message to a process is accomplished by applying the ***Send*** instruction to some mailbox. A process is temporarily stopped from executing when it performs a ***Block*** instruction on its process object.

In StarOS, the only way to refer to an object is by using a capability. Certain

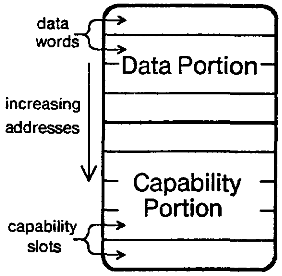
StarOS functions take as an argument a capability for the object; others require both a capability and an offset into the object. For example, sending a message takes a capability for the mailbox but does not require an offset into the mailbox. Reading or writing a word of a basic object demands a (*capability, offset*) pair. An ordinary 16-bit address generated by an LSI-11 implicitly references a capability by naming a window number (see Section 4.2). Further, each capability contains an *object name*. In fact, for many purposes, we think of the capability as being the object name.

A StarOS object occupies a contiguous block of memory. The size of an object is fixed at its creation; objects can neither grow nor shrink. Figure 6-1 depicts an object. Notice that it consists of a *data portion* followed in memory by a *capability portion*, which is sometimes called a C-list or a capability list. Either of these portions may be null, but not both. The data portion is treated as a vector of bytes. The first 4 bits of a processor-generated address name a window, and the remaining 12 bits specify the offset into the data portion.

The capability portion is treated as a vector of two-word *capability slots*. A capability is selected by its slot number; the first slot in a capability portion is slot 0. All references to capabilities are performed entirely by microcoded StarOS instructions. Ordinary LSI-11 instructions are prevented from reading or writing the capability portion in the following way. The descriptor (Section 4.2) for an object contains: the base address of the object, the length of the data portion of the object, and the number of capabilities in its capability portion. The Kmap relocates each mapped data reference relative to the base address and compares its offset to the length of the data portion. References with offsets greater than the size of the data portion are disallowed. Unmapped references need not be performed by the Kmap; with only a 12-bit offset, it is impossible to address past the end of the data portion. As a result, data references are never able to manipulate capabilities; thus capabilities are protected from being overwritten by software.

StarOS objects come in both *representation* and *abstract* varieties. The instructions that apply to representation objects are provided by the StarOS *Nucleus*. The most frequent ones, such as reading or copying capabilities and almost all message instructions, are implemented in microcode. Some of the more complicated instruc-

Figure 6-1 The Structure of an Object



tions, such as blocking or preempting a process, are performed by *Nucleus* software. Abstract objects, on the other hand, are implemented by (user-level) StarOS modules.

StarOS defines 12 representation object types. Among these are basic objects, deque objects, data and capability mailboxes, module objects, process objects, and directories. The data portion of *basic objects* behaves just like an ordinary segment in a virtual-memory system, which means that bytes and words can be read and written in the ordinary way by machine instructions, provided that the segment bounds are not violated. Similarly, a capability ***Copy*** or ***Transfer*** directed to the C-list of a basic object simply copies or moves a capability between the designated slots.

A basic object also can be used for synchronizing multiple processes through the ***Indivisible Increment*** and ***Indivisible Decrement*** instructions. These instructions prevent a second process from accessing a particular word within the data part between the time the first process reads it and writes its new value.[1] If a ***Decrement*** is attempted on a word whose value is already 0, the word will not be decremented. ***Decrement*** returns the old value of the word to its caller, which may then decide whether to *block*.

A basic object can have up to 4K bytes (2K words) in its data portion and 256 slots in its capability portion. It is most efficient to access a basic object with a 4K-byte data part because the Slocal performs address relocation without involving the Kmap (see Section 4.2). This makes it possible for a memory reference to be completed in about 3 μs., rather than the 8.6 μs. that would be needed if the Kmap intervened.

Among their many uses, basic objects serve as code segments, data segments, and carriers. They are also used to hold the process stack for an active process.

Like basic objects, *deque objects* have an uninterpreted capability portion. Unlike basic objects, read and write references to their data portion are interpreted as deque instructions—pushes and pops from both the front and the rear of the deque. A deque instruction pushes or pops a one-word *item*. These instructions automatically update the *front* and *rear* pointers, which are stored in the data portion of the deque. The capability portion has no special semantics and may be used exactly like the capability portion of a basic object. Timings of operations on deques and other representation object types are presented in Section 7.3.

Mailbox objects possess numerous similarities to deques in structure and semantics. Two of the major differences are that mailboxes implement a *queue* of messages—only operations that correspond to ***Push Front*** and ***Pop Rear*** on a deque are defined for mailboxes—and that special handling is performed when a process attempts to remove a message from an empty mailbox.

A StarOS mailbox buffers capabilities or one-word data messages, depending on whether it is a data mailbox or a capability mailbox. Instead of sending a multiword message, as in Medusa, a process typically creates a basic object in which it then stores the message and buffers in the mailbox a capability for the message.

[1] Actually, the implementation is to lock the descriptor for the object, which prevents any other process from accessing the object in the interim.

When a mailbox is empty and receivers are queued waiting for messages, the mailbox is said to be in *registration mode.* A capability for the process object of each waiting receiver is queued in the capability portion of the mailbox, and an integer called a *portal number* (which indicates where in the process's address space the message is to be delivered) is recorded in the data portion of the mailbox.

In structure, a module object is much like a basic object—exactly the same instructions are defined on it. Many of its data words and capability slots are, however, reserved for particular uses, such as information on how to create processes to run functions of the module and pointers to the process objects of processes that have already been instantiated from the module. It also may contain capabilities for data structures that are shared among all processes executing functions of the module.

A process object is the root of the *object graph* of a StarOS process. This means that any object that is accessible to the process can be reached by beginning at the process object and following some chain of capabilities.

A capability itself does not directly point to the object it names. Instead, it indirects (points indirectly) through a *descriptor*, which contains the physical address of the object. This makes it easy to move an object without the need for the backpointers used in Medusa. Descriptors reside in *directories*, which are a representation object type. There is one directory in each cluster. Since an object's name includes its cluster number, it is not possible to move objects between clusters in StarOS (although this can be done in Medusa).

6.1.2. *Capabilities: Structure and Uses*

Capabilities in StarOS are 32 bits long—two 16-bit words, known respectively as the *rights word* and the *data word* (see Figure 6-2). The rights word contains a 3-bit capability type field and up to 13 bits that can be used to specify what rights the holder of the capability has to manipulate the capability and the object it names. The data word usually contains the object name.

The rights in a capability are represented by a bit vector. If a particular bit is on, it means that the corresponding right is present; if it is off, the corresponding right is absent. For example, there are nine rights associated with a capability for a basic object:

Figure 6-2 The Structure of a Capability

Rights Word		Data Word
Capa. Type	Rights (bit vector)	Object Name (in rep. & abstract capas.) Token Value (in token capas.) Data (in data capas.)

Destroy is required to destroy the object.
Copy is needed to make a copy of the capability.
Restrict is required to remove rights from the capability.
Read and *Write* grant the power to read and write the data portion of the object.
C-list Read and *C-list Write* apply to the capability portion of the object.
C-list Restrict is needed to remove rights from the capabilities in the capability portion of the basic object.
Modify, in addition to the rights that authorize the specific operation, is required by any operation that modifies the representation of the object in any way.

Each time an operation authorized by a particular right is invoked on an object, the Kmap checks to make sure that the capability for that object contains that right; otherwise the operation is aborted.

The 3-bit type field determines the *type* of the capability. We will consider four types. *Representation capabilities* are used to access representation-type objects. If the proper rights are present in a representation capability, a process with the capability can access or change the state (i.e., the contents) of the named object. An *abstract capability* names an object that has been assigned an abstract type. An abstract capability authorizes no access to the state of the object but may be used as a parameter to a function of the type manager for the abstract type.

Two types of capabilities do not name unique objects. A *data capability* is an expedient that avoids the need to create one-word objects. Its data word contains an arbitrary 16-bit value instead of an object name. If a process desires to share a small amount of data—one word or less—with another process, instead of placing the data in a one-word object, it can request creation of a data capability to encapsulate the data. A *token capability*, also known simply as a *token*, is a special kind of capability whose purpose is to identify its holder as possessing some special type of authority.

One important use of token capabilities is as type tokens, which play an important role in amplification, as described in the next section. There are also other cases in which a process must identify itself in order to carry out a privileged operation. For example, the *Garbage Collector* must be able to read the capability portion of all objects so that it can determine which objects are garbage. For this purpose, it has the *garbage-collector token*, which it presents to the Kmap in lieu of a capability for each object whose capability portion it wishes to read. The Kmap itself has a copy of this token, which it compares with the token presented by the *Garbage Collector*. If the tokens match, the Kmap allows the *Garbage Collector* to read capability lists. Since no other process has a copy of the garbage-collector token, no other process can subvert protection by masquerading as the *Garbage Collector*.

Similarly, each time the *Object Manager* creates a new object, it must be able to manufacture a capability for that object. This means that it must be able to specify the bit representation of the first capability for the new object. The *Object Manager* possesses the *create token*, which it presents to the Kmap for authorization to invoke the function that fabricates a new capability from any specified bit string.

Table 6-1 Creating and Using an Abstract Object

To create an abstract type:

- Type manager creates type token.

To create an abstract object:

- Type manager creates representation object.
- Type manager deamplifies capability.
- Type manager passes capability back to the client process.

To manipulate an abstract object:

- Client process passes capability for object to type manager.
- Type manager amplifies capability.
- Type manager operates on object.
- Type manager deamplifies capability and passes it back to client process.

6.1.3. Amplification

Abstract-type objects are managed—that is, created, operated on, and perhaps eventually destroyed—by a *type manager* (Table 6-1). Type manager is the collective name for the set of procedures that perform functions on an abstract object. These functions usually will comprise one StarOS module, but they also may be distributed among several different modules. Processes performing these functions are identified as belonging to the type manager by the fact that they possess a copy of the type token for the particular type. One of the type-manager modules must invoke a ***Create Capability*** instruction to create the type token before any abstract objects of that type can be created.

After the type token has been created, it can be copied just like any other capability. If other modules are to be part of the type manager, they will be given copies of the type token. Otherwise, the type manager guards its type token jealously, since it does not want to let untrustworthy processes scribble on objects of its new abstract type.

Creating an abstract object consists of creating a representation object and then making it abstract. The representation object is created by the type manager itself, upon the request of some other process, called the *client*. An object is made abstract by requesting the Kmap to write the abstract-type field in the descriptor, passing both the type token and a capability for the object that is to be made abstract.

Once an object has been made abstract, an abstract capability can be passed back to the client process. The type manager already possesses a representation capability for the object. It uses the ***Deamplify*** instruction to make this capability abstract (which consists merely of changing the value in the type field of the capability from "representation" to "abstract"). Then it returns the abstract capability to the client process.

The client process is now unable to perform any representation operations on its new abstract object. All the microcode functions that perform these instructions (reading values from basic objects, sending messages to mailboxes, and so forth) work only when they are passed a representation capability. The only thing that the client process may do with the abstract capability is pass it to the type manager as an argument to one of the type manager's functions. Moreover, the mere possession of an abstract capability does not confer the right to invoke *all* the type manager's functions. The rights field in an abstract capability includes rights that confer the ability to perform each function. A function may be invoked only if the capability contains the proper rights.

When the client process desires something else to be done to its abstract object, it passes a capability for it to the type manager. The type manager first converts it to a representation capability using the ***Amplify*** instruction. Since amplification requires a type token as one of its arguments, only the type manager is able to ***Amplify*** the capability. Amplification returns a *fully privileged* capability for the representation object—a capability that allows any instruction that is defined on the representation object to be performed.[2] The type manager may now perform its function on the abstract object, using functions on its representation.

6.2. Messages

StarOS messages provide a powerful interprocess communication mechanism. One process may have access to an effectively unlimited number of mailboxes; multiple senders and receivers may simultaneously have access to each mailbox. Using a capability message, one process can send arbitrary amounts of information, as well as an arbitrarily structured graph of objects. A received capability is sufficient to allow the receiver the ability to access any object within an arbitrary graph structure of objects.

In Section 6.1.1, it was noted that there were two different types of mailboxes, data and capability mailboxes. The same applies to messages. Data mailboxes can hold (or "buffer") only data messages; capability mailboxes can hold only capability messages.

6.2.1. Send *and* Receive

There are two different message instructions: ***Send*** a message to a mailbox, and ***Receive*** a message from a mailbox. There are two different varieties of ***Receive***: ***Conditional Receive*** and ***Registered Receive***. If there is a message in the mailbox

[2] This is in contrast to Hydra [Wulf *et al.* 81], in which amplification uses an *amplification template*, which only turns on certain rights in the representation object.

when either kind of ***Receive*** is performed, then that message is received (into a portal, as described in the next section). The two instructions differ when there is no message to be received. In that case, ***Conditional Receive*** merely returns with a code indicating that no message was received, but ***Registered Receive*** also causes the receiver to be "registered" in the mailbox.

When a process that wants to perform a ***Receive*** is registered, a capability for its process object is placed in the capability portion of the mailbox. Registered receivers are, in fact, queued in the mailbox, so that, if more than one process has attempted to do a ***Registered Receive***, the first process will receive the first message that arrives, the next process will receive the next message, and so forth. When all the registered receivers have received messages and have been dequeued, the mailbox returns to buffering mode. Future messages may arrive before any process has tried to receive them. They will be buffered in the mailbox and wait there for the first receiver.

Now the semantics of ***Send*** should be obvious:

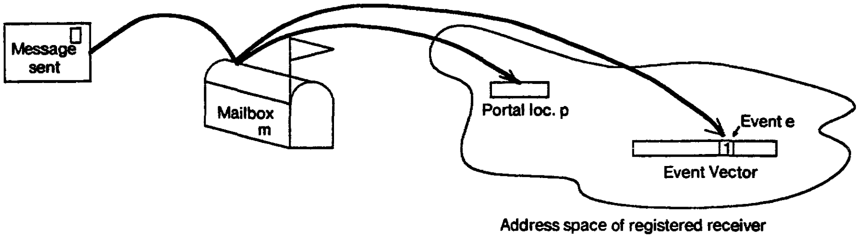
- If there are no registered receivers waiting in the mailbox, the ***Send*** function will buffer the message in the mailbox.
- If registered receivers do exist, the oldest registered receiver is dequeued from the mailbox, and the message is delivered directly to that process without being buffered in the mailbox. This is called ***Portal Delivery***.

The ***Send*** instruction fails if the mailbox is full of messages. (In StarOS, unlike Medusa—see Section 5.2—the sender cannot be suspended, awaiting room to leave the message.) The ***Registered Receive*** instruction also can fail if there is no space left in the mailbox to register another receiver. Thus there are three possible outcomes to a ***Registered Receive*** instruction: message received, receiver registered, or mailbox full of registrants. The result returned by ***Registered Receive*** indicates which has occurred.

When a message is received, it comes out of the mailbox and has to go somewhere. Its destination is determined by a *portal*, which is an ordered set consisting of these three items: a mailbox, an event (optional), and a location in the address space of the receiving process, called the *portal location*.

In StarOS (unlike Medusa), a process is never blocked automatically. If it has performed a ***Registered Receive*** and no message is buffered, it may decide whether or not to block until the message arrives. If it decides to block, it blocks on the event associated with the portal. Then, when a message arrives, the event is set, causing the process to be awakened. Alternatively, if the process has decided not to block, it will test the event from time to time to find out whether the message has arrived. The argument to block is in fact a bit vector, which allows more than one event to be specified (in Medusa, this is accomplished by ***Multi-Event Wait***, a special operation). This means that a process can be simultaneously blocked on the events associated with more than one mailbox. In this way, it can wait for the first message to arrive in any of a set of mailboxes.

Figure 6-3 Portal Delivery of a Message to a Registered Receiver



6.2.2. *Portal Delivery*

The portal is used whenever a message is actually received, either by a ***Receive***—registered or conditional—when there are messages buffered in the mailbox, or by dequeueing a registered receiver when a message arrives at a mailbox that is in registration mode. The message is taken out of the mailbox and copied into the location named by the portal.

Although the message is always placed in the portal location, regardless of when it is received, StarOS *implements* the delivery in two different ways: If a message is already buffered when a ***Receive*** is performed, it is transferred to the portal location. If the message arrives at a mailbox that contains registered receivers, it is transferred to the portal location by the *Nucleus* ***Portal Delivery*** instruction. In addition to the delivery of the message to the portal location, ***Portal Delivery*** also sets the event associated with the portal, if any, so that the newly unregistered process may discover that it has been unregistered.

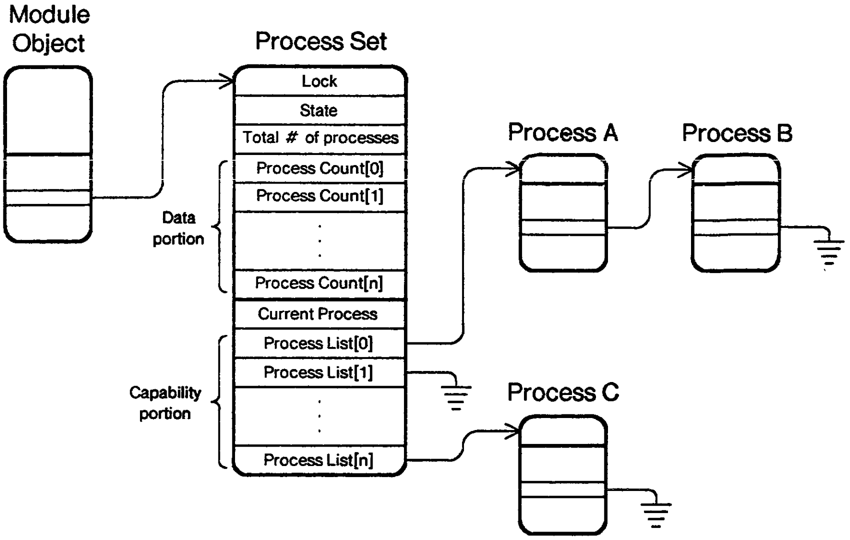
To illustrate ***Portal Delivery***, consider a process that has associated a portal location *p* with an event *e*. It then performs a ***Registered Receive*** on an empty mailbox *m*. Figure 6-3 depicts what happens when, eventually, a message is sent to *m*. It will not be buffered in the mailbox but will be delivered directly into *p*, and *e* will be set. The process may or may not have blocked waiting for the message. If it did, it may now resume processing.

6.3. Modules and Functions

All user programs in StarOS are made up of *modules*. Each module exports one or more *functions*, which ideally are closely related. Each *process* is created to perform a specific function; it executes the code for that function. In summary, functions are components of modules, and processes are the instantiation of functions.

A programmer creates logically separate sets of programs. Logical separation means that one set of programs depends only on the specifications—the externally known behavior—of each other set and not on its implementation. In StarOS, each such set of programs is called a module. The partitioning of programs into modules creates a set of boundaries. Programs in one module that want to make use of

Figure 6-4 A Process Set



programs in another module must do so by means of the ***Invoke*** instruction, although programs within the same module can call each other by any means, including simple routine calls.

The organization of a module is defined by a particular kind of basic object called the module object. A StarOS module may define one or more abstract types; its functions are the set of operations defined on the type. The initialization function of the module, for example, might create a new abstract type by creating a new type token and storing it in the module object for later use (e.g., for making a newly created object abstract and for amplification of a capability passed as an argument to a function of the module).

Several processes may be created to perform one of the functions defined in the module object. One common example is having several processes perform the same algorithm on different parts of a large array. To keep track of these processes, StarOS uses a *process set*. A process set is a basic object containing pointers to lists of processes, one list per function defined by the module. As shown in Figure 6-4, the capability portion contains a pointer to each process list. The *current process* capability is used to identify the process being inserted or removed from the set in order to provide enough state to complete an interrupted operation.

In the data portion of the process set, a lock is provided to ensure atomic manipulation of the process lists. The state field records the internal state of the process set to enable the completion of an interrupted operation. The remainder of the data portion is devoted to counts of processes that are performing each individual function, as well as to a sum of the individual counts.

6.3.1. *Function Invocation*

A StarOS function is performed when it is invoked by some process. A function is named by specifying both a capability for the module that contains it and the number of the function within the module. Both these arguments are placed inside a small basic object called a *carrier*. This carrier is called the invocation carrier and is the sole argument of a *Nucleus* ***Invoke*** instruction.

In StarOS, some function invocations cause creation of a new process, and some do not. When a function of a module is invoked, StarOS must determine whether a new process must be created and to which mailbox the carrier should be sent. If the process already exists, the *Nucleus* sends the carrier to a mailbox called the "invocation mailbox." Otherwise, the *Nucleus* first invokes the *Process Creator* to create a process to perform the desired function.

The client of a module (the process that performs ***Invoke***) never needs to be concerned with the details of when, where, and how invocation is accomplished—the carrier will be sent to *some* mailbox, and *some* process will service the request. The details are reserved to the implementor of the module. When the request has been serviced, the results, if any, are stored back in the carrier. The carrier is then returned to its *return mailbox*. (A capability for the return mailbox is stored in a special slot in the carrier.) Although the mechanism of function invocation may seem complex, it does allow the implementor to preserve a particular abstraction for clients while experimenting with a wide variety of implementations.

Absent Functions. A function is either *present* or *absent*, as determined by the value of its *present/absent process* field in the module object. Of the two types of functions, the rules for invoking an *absent* function are the more straightforward. A new process will *always* be created. The new process is linked to the module's process set in the chain corresponding to the alias function number (see Section 6.3). When a process is created, a *private mailbox* is also created. It is to this mailbox that the *Nucleus* sends the carrier that served as a parameter to the ***Invoke*** instruction.

Present Functions. For many functions, it is undesirable to create a new process upon each invocation (for example, because of the overhead of process creation). These functions are defined as *present functions*. For each present function of a module, an *invocation mailbox* is created when the module itself is created. Capabilities for these mailboxes are placed in the module object. Functions are allowed to share an invocation mailbox. When a *present* function is invoked, the carrier is sent to the function's invocation mailbox. To determine whether a process must be created, the module object is consulted. If the *present/absent process* field for this function is nonzero, no new process will be created. Otherwise, a new process is created, the *present/absent process* field is incremented, and the process is linked to the module's process set.

6.3.2. *An Example: The Object Manager*

The *Object Manager* is the StarOS module whose duty is to allocate and deallocate memory for an object, create a descriptor for it, and create the first capability for the object. It is invoked asynchronously using the ***Invoke*** operation of the *Nucleus*. The invoking process may choose to proceed while it is waiting for the object to be allocated, though in practice, it almost always blocks. Object management is a cluster-local activity in that an *Object Manager* accepts requests only for objects in its own cluster. When an object in a remote cluster is requested, the local *Object Manager* forwards the request to the *Object Manager* in that cluster.

The *Object Manager* includes other functions for memory management. All the functions that it provides are listed in Table 6-2 (with their corresponding function numbers).

6.3.3. *Task Forces*

StarOS task forces were designed to achieve three benefits: a low cost/performance ratio, enhanced reliability, and adaptability to hardware changes. All three of these goals can be met if task forces are made up of a large number of small processes. The first two goals can be served by taking advantage of the available parallelism. Improved performance results when the work is decomposed into a large number of independently executable pieces. Reliability is also increased because a large task force can usually be structured so that no process is indispensable. Thus a single failure cannot prevent the task force from successfully finishing its work. The third goal of hardware adaptability can be realized if the task force can grow (or shrink) with the addition (or removal) of processor and memory resources. This is particularly easy to accomplish when the task force is composed, in part, of duplicated processes or data.

Recall that a Medusa task force is composed of those activities that share a single SDL. Capability-based StarOS allows memory to be shared in arbitrary ways; it is not possible to characterize a task force based on the way memory is shared. The StarOS concept of a task force is less precise than Medusa's, since it does not

Table 6-2 Functions of the *Object Manager*

0 ***Deallocate Object***. Deallocate objects in garbage-collector deque (see Section 6.6).

1 ***Exclude Memory***. Remove a Cm's memory from the system. (*Memory Manager*)

2 ***Include Memory***. Include a Cm's memory into the system (for example, at system initialization time or after a malfunctioning Cm has been fixed).

3 ***Allocate Object***. Allocate an object.

4 ***Object Status***. Give the type, location, and size of an object. Requires a capability for the object.

correspond to a specific run-time structure. Rather, it depends more on the observer's viewpoint. A task force consists of those processes that are viewed as closely interacting. A task force may be viewed as comprising a number of related modules, and according to this definition, STAROS itself qualifies as a task force.

6.4. Exceptions

STAROS's exception-handling mechanisms are not as fully developed as MEDUSA's, but there are a few interesting similarities. For example, exceptions are handled by the same process that encountered them, whenever possible. If it is not possible, another process can be invoked to take over; this is called *bailing out* the process. At creation time, a process is endowed with a default bailout function, although an alternative bailout function can be specified if desired.

More specifically, STAROS exceptions are divided into two classes, *serious exceptions* and *nonserious exceptions*. Serious exceptions include errors such as out-of-bounds addressing, attempts to execute illegal instructions, and system or hardware errors. These exceptions set a *serious flag*, which can be cleared by software.

An exception handler is a routine within the same STAROS module that has encountered the exception. Ordinarily, a handler clears the serious flag to inform STAROS that it is capable of handling an exception. If another serious exception occurs before the serious flag is cleared, however, the process is bailed out instead. Unlike exception handling, bailout is performed by invoking another STAROS process. When the microcoded *Bail Out* instruction is executed, the process is stopped and the *Nucleus* invokes the failing process's bailout function. Like any other STAROS function invocation, this involves sending a message to a process executing a particular function (the bailout function) of a particular module (the bailout module). A capability for the bailout module and the number of the bailout function are both obtained from known locations in the process object of the failing process. The invocation message is sent using the *lifeboat carrier*, a special carrier created in advance for just this purpose.

In principle, the bailout function could repair the state of the failing process and return control to it. In practice, recovery code like this is not as well developed as in MEDUSA. The bailout function usually provides only detailed debugging information to the user.

6.5. Scheduling

As mentioned in Section 4.1.3, the responsibility for scheduling in STAROS is divided along policy/mechanism lines: *Schedulers* make the high-level policy decisions of where processes should execute and for how long, while *multiplexers* implement these decisions by selecting the next process to execute and interrupting it when its time has expired. The schedulers decide which processes to send to the various multiplexers. The choice of scheduling policy is up to the user, who may want to experiment to fine-tune his or her task force. Each *Nucleus* process includes a multiplexer that implements scheduling decisions for its own processor.

One of the main functions of the multiplexer is to select the next process to execute on a particular processor and to cause that process to begin execution. It provides a mechanism to support policies dictated by schedulers, which are implemented as user processes. The multiplexer searches a priority-ordered set of *run queues*. It selects the process that has been in the first nonempty queue for the longest time. Multiple processors can share run queues. The multiplexer allows a round-robin (RR), FCFS, or coarse-grain priority (for example, foreground/background) discipline to be implemented with little scheduler intervention.

The multiplexer provides four functions:

Selecting the next process to run on its processor. This function is performed when the multiplexer first starts up after bootstrapping and also after each other operation performed by the multiplexer. It searches the active run queues (see Section 6.5) in priority order and selects the first runnable process it encounters.

Interrupting a process whose time slice has expired.

Reevaluating a process's runnability when it terminates or is blocked.

Preempting a process.

The basic data structure used by the multiplexer is an ordered set of *run queues*, which are simply mailboxes that contain capabilities for process objects. Collectively, these run queues are known as the *search list* because they are searched each time the multiplexer needs to start up a process. They are numbered in priority order, with run queue 0 having the highest priority and run queue *n* having the lowest priority (assuming there are $n + 1$ run queues). The priority of a StarOS process is simply the number of the run queue to which it is assigned. A particular run queue may be private to a particular Cm, or it may be shared by several Cm's. A process is marked for multiplexer service simply by sending it to a run queue. This is typically done by the scheduler or the *Process Creator* module.

The multiplexer relies on the schedulers to place processes on run queues appropriately. A scheduler can alter the search list dynamically and can interrogate the multiplexer status to observe multiplexer operation.

A StarOS scheduler may be a user process, implementing any of a variety of scheduling policies (e.g., priority, proximity, preemption, load sharing, and group scheduling). Each function of a module may specify several *scheduling parameters* to be used for its processes. For example, it may request that a function execute on a particular Cm or in a particular cluster. If processes within a task force communicate via shared memory, it may be advantageous to locate them on the same Cm. Even if they communicate solely via messages, it may be advantageous to locate them within the same cluster. A function also may request that its processes run on a Cm with a particular set of attributes, such as a large amount of memory or an attached disk.

6.6. Garbage Collection

Garbage collection in StarOS is a twofold problem: Garbage collection must run in parallel with other processing in the system, and it must be capable of being partitioned by cluster. In the past few years, several designs for correct and efficient parallel garbage-collection algorithms have appeared, notably that of Kung and Song [Kung and Song 77a], upon which the StarOS *Garbage Collector* is based. The algorithms described below are similar to those of Kung and Song, though rephrased in Cm* and StarOS terminology. As in much garbage-collection literature, the non-garbage-collection activity in the system is referred to by the generic term *list processing*. List processors and garbage collectors may execute concurrently.

Creation of an object in StarOS requires the allocation of an object name, a descriptor where information about the physical representation of the object is stored, and a block of primary memory to contain the physical representation of the object. When an object is deallocated, its name, descriptor, and memory are made available for other objects. Each of these resources is finite, and the system must make provision for its reuse.

The object graph formed by objects and capabilities is a directed graph. An object can be accessed only if there is a *path* of capabilities to it, beginning with some process object. The collection of all processes forms a set of *roots*, a set of objects from which there is a path to any object that can be accessed. There may be, and indeed are, other sets of roots.

Since capabilities may be created and destroyed, it is possible that inaccessible objects may exist. Such objects are called *garbage*, and the purpose of garbage collection is to discover such objects. When a garbage object is discovered, it may then be destroyed to make the resources used by the object available for new objects.

The basic algorithm for identifying garbage is simple. It uses the concept of the *color* of an object, a quantity stored in the color field of the descriptor for that object. The garbage-collector token confers the authority to write the color field.

Some comments about the algorithm in Figure 6-5 are in order. First, the name of a StarOS object enables the descriptor, and hence the physical representation of the object, to be found so that the object may be searched for capabilities. Second, the algorithm is recursive. When an object is marked *black*, the object is searched; this may result in another object being marked *black*, which will result in further search. Finally, the algorithm assumes the StarOS world is static: No new capabilities or objects are created during the period when objects are marked.

The assumption that no capabilities are created during the execution of the garbage-collection algorithm would be a serious constraint on the StarOS system. It would effectively require that all processing stop for the duration of the garbage collection. This would be unfortunate for a multiprocessor system whose principal virtue is the ability to execute several independent activities concurrently. Consequently, the garbage-collection algorithm has been modified to allow other system activity, including the creation of capabilities, to execute concurrently.

Figure 6-5 Basic Algorithm for Garbage Collection

```
mark each object white;
for each object in a set of roots do mark-black;

procedure mark-black = {
    mark the object black;
    for each capability in the object do
        if the object named by the capability is white
            then mark-black;
}

All objects which remain white are garbage objects.
```

Furthermore, the physical structure of Cm* suggests that the system be partitioned by clusters. To the extent possible, the activity within a cluster is independent of the activity in other clusters. Performance and reliability considerations suggest that the task of garbage collection be partitioned on a cluster-by-cluster basis. Thus it is presumed that each cluster contains a *Garbage Collector*. The Kmap within each cluster serves the role of a list processor, creating and storing the capabilities into other objects.

In order for the *Garbage Collector*s in each cluster to operate independently, the following rule will be observed: *If a capability for an object in one cluster is ever stored in another cluster, then that object will not be subject to garbage collection.* This rule partitions objects that are not shared between clusters, and therefore may be independently collected and destroyed, from those objects that are shared, and therefore may not be destroyed without the cooperation of other clusters.

When an object is first created, the first capability for the object is created by the *Object Manager* in the same cluster as the object.[3] Consequently, no object created in one cluster can be accessed by a process within a second cluster without a capability in the first cluster being copied or transferred from the first cluster. Because the Kmap microcode in the first cluster must participate in each such action, the microcode implements the rule as follows: *Each time a copy is made of a capability in this cluster, for an object in this cluster, by a process in another cluster, mark that object* red. Marking an object *red* takes 20.1 μs.

One other activity is required of the Kmap to allow concurrent processing with garbage collection. If the *Garbage Collector* is marking the objects as described above, then each time a new capability is created for an object that has not been marked *black* by the *Garbage Collector*, the *Garbage Collector* must be notified. Since the Kmap is a participant in the creation of all capabilities, the Kmap will perform the operations indicated in Figure 6-6.

The deque is shared by the Kmap microcode, which writes it, and the *Garbage*

[3] This capability may immediately be transferred to another cluster if the request for creation of the object came from a process in another cluster.

Figure 6-6 Notifying the *Garbage Collector* of Capability Creation

```
whenever a capability for an object in this cluster is created in this cluster
then notify the Garbage Collector by
    placing the name of the object in a shared deque.
```

Collector, which reads it. It is known as the *garbage-collector deque*. Deques contain only data, not capabilities, so the garbage-collector deque contains 16-bit object *names*, not capabilities. Writing an object name to the deque takes an extra 35.7 μs. beyond the time normally needed to write a capability. Because the creation of objects happens concurrently with garbage collection, each new object is marked *yellow* to avoid confusing the *Garbage Collector*.

6.6.1. The Trashman Cometh

In StarOS, within a single cluster, the nucleus module object together with all *red* objects constitute a set of roots.[4] Object colors are defined as follows:

White No path to the object has been found.
Yellow Each newly created object is initially assigned the color *yellow*.
Red It may be possible to reference the object from another cluster. An object can be *red* in addition to whatever other color the object may be. For instance, an object might be both *red* and *black*.
Black A path to the object has been found.

There are two phases to garbage collection: *marking* and *collection*. The marking phase, presented in Figure 6-7, identifies garbage objects, and the collection phase deallocates them. At the conclusion of the marking algorithm, all *white* objects are garbage. The collection phase then proceeds as indicated in Figure 6-8.

6.6.2. The Red Menace

So far, there has been no provision for an object marked *red* ever to cease being *red*. Consequently, after a period of operation, the system will become littered with garbage *red* objects. To remedy this situation, the *Garbage Collectors* of each cluster will make a cooperative effort to find such garbage objects. Because this activity need not be frequent, it is acceptable to require that all of the *Garbage Collectors* execute the algorithm in Figure 6-9 concurrently. Note that since all *red* objects are initially marked *white*, at the conclusion of this marking algorithm, only objects in one cluster that are accessible from another cluster are *red*.

[4] The process set of the *Nucleus* module object contains capabilities for all *Nucleus* processes, which in turn have capabilities for all the run queues and the user processes assigned to processors. *Red* objects are included in the set because other clusters may have access to them.

Figure 6-7 Marking Phase of Garbage Collection

```
mark each object which is not red to be white.
mark the Nucleus module object black
    and push its name onto the rear of the garbage-collector deque;
mark each red object black / red
    and push its name onto the rear of the garbage-collector deque;

while the deque is not empty do
    pop a name j from the deque;
    for each capability in j do
        if the object named by the capability
            is in this cluster and not black then
                mark the object black;
                push its name onto the deque;
```

Figure 6-8 Collection Phase of Garbage Collection

```
Push the names of all white objects onto the garbage-collector deque
    and Invoke the Deallocate Object function of the Object Manager
        with the garbage-collector deque as parameter.
```

Figure 6-9 Algorithm for Red Garbage Collection

```
mark each object white.
mark each Nucleus module object black and push its name onto the deque.

until all Garbage Collectors are finished do
    while the deque is not empty do
        pop name of object j from the deque;
    for each object k named by a capability in the object j do
            if the object named k is in another cluster then
                send k to the other cluster;
            else if the object k is not black then
                mark the object named k black;
                push k onto the deque;
    for all capabilities received from other clusters do
        if the named object is neither black nor red then
            push the object's name onto the deque;
        mark the object red.
```

6.7. Performance Aspects of Garbage Collection in StarOS

In Section 6.6, the StarOS *Garbage Collector* was described as a program that runs in parallel with other processes in the system and is capable of collecting garbage in a single cluster or simultaneously throughout the entire system. These two aspects of the *Garbage Collector* raise two performance-related questions: when should garbage collection be initiated to take best advantage of idle resources in the system? When should a local, as opposed to a cooperative systemwide, garbage collection be initiated? Chansler [Chansler 82] has studied both these issues in some detail.

6.7.1. When to Collect Garbage

Ordinary nonconcurrent garbage collectors must decide when memory has become sufficiently fragmented to justify a garbage collection. The StarOS *Garbage Collector* must face this issue, too, but it also must factor into its decision the demand for several other resources. Among the decisions it must make are these: Should objects that could be local to a process be "mis-placed" in a remote Cm to postpone garbage collection? Should idle processors be used to initiate a garbage collection even before the need for garbage collection becomes apparent?

Should Objects Be Mis-placed? When memory is in short supply, the *Object Manager* will be unable to create objects in some Cm's. Although each request for a new object specifies a desired Cm, the system is free to satisfy the request by creating an object in any Cm because the addressing mechanism is independent of object placement. Sometimes it takes longer to access an object placed in another Cm. If such a performance penalty ensues, the object is said to be *mis-placed.* An object is mis-placed if one of the following is the case: It is a 4K-byte basic object anywhere else but in its preferred computer module in its preferred cluster (the Slocal can translate addresses only for 4K-byte basic objects in its local Cm); it is any other sort of object outside its preferred cluster (intercluster references are required to access it).

The performance penalty in these cases is at least a factor of 3 and may be as high as 15 in the case of a StarOS 4K-byte basic object in another cluster (see Table 3-2). Chansler developed a simple model that predicted that with a performance penalty of 3, mis-placing objects would rarely save as much garbage-collector execution time as it would cost. For example, if the system were able to double the time between garbage collections by mis-placing half of all objects, the time lost due to extra remote references would be at least 12 times as great as the savings in garbage-collection time. The model assumes that all objects are referenced equally often.

As severe as the model makes the costs of mis-placement appear, it probably underestimates them. Tables 6-3 and 6-4 present some statistics on the use of memory by two StarOS task forces—a 5-cluster configuration of the StarOS operating system itself and a 31-slave instantiation of the PDE task force. Notice that

Table 6-3 Objects—StarOS Initialized for Five Clusters

Type	*N*[1]	%*N*[2]	No. *red*[3]	% *red*[4]	Mean bytes[5]	Mean capas[6]	Total space[7]	% space[8]	% *red* space[9]
Basic object	395	32	74	6	175	14	91,498	6	1
Nucleus process	40	3	0	0	126	94	20,090	1	0
User process	41	3	13	1	126	75	17,406	1	0
Device object	83	7	1	0	75	0	6,240	0	0
Shadow object	4	0	0	0	3,142	0	12,568	1	0
Capa mailbox	215	17	2	0	17	22	23,042	1	0
Directory	12	1	0	0	2,603	13	31,872	2	0
Deque object	71	6	9	1	526	0	37,336	2	1
Stack object	40	3	0	0	136	0	5,440	0	0
4K basic object	341	27	68	5	4,096	0	1,396,736	85	17

Totals:		
Number of objects	1242	
Size of data parts	1,572,772 bytes	20.0% *red*
Size of capa parts	17,364 slots	16.2% *red*
Space	1,642,228 bytes	19.9% *red*
Means:		
1,266 bytes in data part	14 slots in capa part	1,322 bytes total space

[1] Number of objects of this type.

[2] Fraction (%) of the total number of objects that are of this type.

[3] Number of objects of this type that are *red*.

[4] Fraction (%) of the total number of objects that are *both* of this type and *red*.

[5] Mean size of the data portion of this type of object (bytes).

[6] Mean size of the capability portion of this type of object (capability slots).

[7] Total space occupied by objects of this type.

[8] Fraction of total space occupied by objects of this type.

[9] Fraction of total space occupied by *red* objects of this type.

about one-quarter of all objects are 4K-byte basic objects; these are the largest objects in the system and the most likely to require mis-placement. Suppose that a Cm has run out of memory, so it is necessary to place objects elsewhere. The system can, on average, place only 3 or 4 objects elsewhere before it becomes necessary to mis-place a 4K-byte basic object. Unfortunately, almost all 4K-byte basic objects contain the code or the stack for some process. All 341 in Table 6-3 and 70 of 80 in Table 6-4 did. These objects are referenced more often than other objects (see Table 3-3, for example). Thus the objects most heavily referenced are those most likely to be mis-placed.

Should Idle Processors Be Used to Collect Garbage? Although it usually is not a good idea to have resources standing idle while there is work to do, sometimes a "greedy" allocation strategy fails to produce long-term benefits. For example, the StarOS *Garbage Collector* must meet a real-time constraint: It must examine capabilities at least as fast as other processes produce them. It may be difficult for it to do so if it runs without local code or without a local process stack. Duplicating just

Table 6-4 Objects—PDE Task Force, 31 Solvers*

Type	*N*	%*N*	No. *red*	% *red*	Mean bytes	Mean capas	Total space	% space	% red space
Basic Object	98	35	34	12	96	38	24,346	7	4
User Process	33	12	28	10	126	88	15,774	4	4
Capa Mailbox	66	24	0	0	17	15	5,000	1	0
4K Basic Object	80	29	12	4	4,096	0	327,680	88	13
Totals:									
Number of objects			277						
Size of data parts			342,412 bytes		16.6% *red*				
Size of capa parts			7,597 slots		70.1% *red*				
Space			372,800 bytes		21.0% *red*				
Means:									
1,236 bytes in data part					27 slots in capa part		1,346 bytes total space		

**See notes for Table RC32.*

the code and process stack in the local memory of each Cm would consume about 5 percent of Cm*'s total memory. When initiated, the *Garbage Collector* could determine the location of the rest of the objects it needed (the garbage-collector deque, for example) and make them addressable.

Another concern is whether a cooperative, systemwide garbage collection is required. If so, it cannot be initiated until a processor is available in *each* cluster. If a systemwide garbage collection is not required, however, and an idle Cm contains the *Garbage Collector* code and has room for its stack, then there is no advantage to postponing garbage collection.

6.7.2. *When to Collect* Red *Garbage*

Recall from Section 6.6.2 that an object to which a capability in a remote cluster points is marked *red* and is ineligible for removal by a local (clusterwide) garbage collection. Eventually, the *Garbage Collector*s of each cluster will run simultaneously and perform a garbage collection of the entire main memory, removing *red-and-white* as well as *white* objects. To determine when *red* garbage collection is appropriate, we must be able to estimate the likelihood that an object is *red*. Refer once again to the measurements in Tables 6-3 and 6-4. They indicate that, for both task forces, about 20 percent of allocated memory is occupied by *red* objects.

Chansler [Chansler 82] analyzed the efficacy of *red* garbage collection as follows: Let

M be the size of a cluster's memory
α be the fraction of allocated memory that is *not* garbage
ρ be the fraction of garbage that is *red*

γ be the fraction of memory that is garbage

K_c be some constant that depends on the cluster and represents the speed of the processors and the rate at which capabilities are created within the cluster

Then T_c, the cost of a clusterwide garbage collection, is

$$T_c = M(\alpha + \rho\gamma)\, K_c$$

(Garbage collection examines all accessible objects plus all the *red* garbage.) The benefit of a clusterwide collection, the amount of memory recovered, is proportional to $M\gamma(1 - \rho)$. Let C_c be the ratio of the time required for a systemwide collection to the time for a clusterwide collection. The benefit of a systemwide collection will be $M\gamma$ because all garbage is removed. For a single cluster, a systemwide collection will be advantageous whenever the ratio of benefits to costs is larger for a systemwide collection than for a clusterwide collection; i.e., when

$$\frac{M\gamma}{C_c T_c} > \frac{M\gamma\,(1 - \rho)}{T_c}$$

or

$$C_c < \frac{1}{1 - \rho}$$

If $\rho \approx 0.2$ (that is, if 20 percent of garbage, like 20 percent of allocated memory, is *red*), a cluster will find a systemwide collection attractive only if $C_c < 1.25$.

Table 6-5 compares the cost of clusterwide and systemwide garbage collections at one "snapshot" of a 4-cluster StarOS system. The last column gives the value of C_c. Systemwide collection would be favored only by cluster 5, while clusterwide collections would be favored by the other clusters.

Table 6-5 Cooperative versus Local Garbage Collection

Cluster (Cm's)	Objects	Cooperative*	Local*	Ratio (C/L)
1 (10)	251	2,058	862	2.39
2 (7)	2,13	2,093	690	3.04
3 (5)	201	2,096	626	3.35
5 (9)	434	2,044	1,861	1.10
Total (31)	1,099	8,291	4,093	2.03

*Units approximately 2 ms. (Time is for the search phase for the process in the indicated cluster.)

6.7.3. *Measurements of a Typical Garbage Collection*

Excluding the cost of inspecting the invocation carrier (see Section 6.3.1) and the time to send a return message, the *Garbage Collector* typically spends more than 90 percent of its time examining the object graph. The remainder of its time is spent in two iterations through the directory objects—the first to color all descriptors *white* and the second to compile the list of garbage objects after the object graph has been traversed. Note that deallocation of garbage objects is performed by the *Object Manager*, not the *Garbage Collector*. Synchronization with other *Garbage Collector* processes also consumes some time.

Each directory may contain up to 256 object descriptors, except for the root directory, which contains, at most, a descriptor for itself and 31 other directories that are allocated only as needed. Thus the number of directories searched is proportional to the number of object names in use, since names of deallocated objects are reused in preference to allocating a new directory. A typical invocation of the *Garbage Collector* on a single-cluster StarOS configuration yielded these values: 929 capabilities discovered in objects, 440 objects, 6,834 capabilities found in objects or created during the garbage collection, and 6,091 capability slots within objects. In this example, approximately 33 percent of the *Garbage Collector*'s memory references are due to capabilities created while garbage collection was in progress. This underlines the importance of having the *Garbage Collector* execute local code so that it is not overwhelmed by the rate of capability creation.

Finally, the *Garbage Collector* consumes a moderate amount of memory. The shared part of the *Garbage Collector*, the *Garbage Collector* module, consists of 11 objects, including 3 that contain code. These objects occupy 13,416 bytes of memory. For each cluster in the system, an additional 4 objects (4,052 bytes) are needed. Each *Garbage Collector* process requires 7 objects (8,866 bytes), including a copy of one code page. By contrast, the smallest StarOS task force includes 3 objects and takes up 4,258 bytes. The smallest StarOS process has 5 objects (674 bytes) without copies of code pages. Further measurements of the *Garbage Collector*'s performance may be found in Section A.17.

6.8. Summary

Our presentation of the StarOS operating system has paralleled the Medusa discussion to highlight the two systems' similarities in concept and structure. Occasionally, the symmetry is masked by the use of different terms for similar notions ("process" vs. "activity" and "nucleus" vs. "kernel," for example). This is partially due to the close communication between the two groups, which permitted fine differences in function to be identified and discussed.

StarOS is an object-oriented multiprocessor operating system using capability addressing to implement a shared global address space. Like Medusa, it has two parts: the *Nucleus*, which is resident in each Cm and Kmap; and StarOS *modules*, which can be replicated but need not be present in all processors. The StarOS *Nucleus* microcode consists of functions that the Cm* architecture forces to be there, such as address mapping and interprocessor communication, and functions with critical performance requirements, such as capability addressing and operations on representation objects. The *Nucleus* software resident in each Cm performs functions involving the management of processes and local hardware resources. A StarOS module is an entity containing all necessary information for a process to be instantiated. StarOS modules are structurally indistinguishable from user modules; they require no special privileges.

StarOS objects are composed of a capability list and a data portion. The capability list holds capabilities, while all other data resides in the data portion. A capability contains a pointer to an object and several bits that grant "rights"—the permission to perform specific operations on the object to which the capability points. Object types can be divided into *representation* and *abstract* types. Operations on representation objects are implemented in the nucleus.

There are 12 representation object types, including basic, deque, mailbox, module, and process objects. Abstract objects are implemented by a *type manager*, which is the collective name for the set of procedures that perform functions on the abstract object. An abstract object is first created as a representation object and then made abstract using the mechanism of *deamplification*. Later, before an abstract object can be manipulated, a capability for it must first be *amplified*.

All StarOS programs are composed of modules. Each module exports a set of functions. A process is created to perform a specific function of a specific module. A function in one module that wishes to call a function in another module does so by using the StarOS ***Invoke*** instruction.

Interprocess communication can be performed by means of messages. Messages in StarOS are usually sent by reference, rather than by value as in Medusa. Using a capability message, one process can send arbitrary amounts of information as well as an arbitrarily structured graph of objects.

Process scheduling in StarOS exhibits policy/mechanism separation. The mechanism is provided by the *multiplexer*, the part of the *Nucleus* responsible for selecting the next process to execute on a single Cm. The *scheduler* makes global decisions relating to scheduling policy.

Garbage collection in StarOS uses parallelism in two ways: It runs in parallel with other processes in the system, and the *Garbage Collector* is itself a parallel program. Garbage objects are objects for which no capability exists. To identify garbage, capabilities are followed to traverse the object graph. Inaccessible objects are then deleted. To prevent objects created during garbage collection from being erroneously identified as inaccessible, the Kmap microcode writes a special value into the "color" field of the descriptor for such objects.

The performance of the StarOS *Garbage Collector* has been studied in detail. Among the questions studied were these: When should a garbage collection be initiated, depending on the system load, and when should a one-cluster garbage collection be used instead of a global (intercluster) collection? Garbage collection has been stressed in our discussion of StarOS because it was an important research question. While StarOS concentrated much attention on memory management, it paid relatively less attention to robustness, which figured prominently among Medusa's goals. Hence robustness was described in greater detail in Chapter 5.

Chapters 5 and 6 have presented MEDUSA and STAROS, noting their similarities and reflecting their different emphases. Chapter 7 compares the functionality and performance of their symmetric features.

Acknowledgment. Section 6.6 was adapted from a chapter of the internal STAROS Design Manual [CMU 79] written by Robert J. Chansler, Jr.

7. Operating-System Performance

In the six years since both Cm* operating systems became operational, they have been measured in several ways. They have been compared with the "bare-bones" Smap microcode (Appendix D) and against each other. Various unique features of both systems have also been evaluated. This chapter concentrates on two specific aspects of operating-system performance: performance of firmware operations in the Smap, MEDUSA, and STAROS microcodes; performance of message operations (interprocess communication) in MEDUSA and STAROS. Our consideration of these aspects will focus on comparisons between software and firmware systems that have been implemented on Cm*. While such measurements are useful in elucidating performance issues, some care is needed in extrapolating from them to predict the performance of an application running on top of either system.

Smap, MEDUSA, and STAROS, in that order, provide increasing functional power in the addressing and protection that they provide to the software developer. For the most part, the cost of comparable functions increases in the same order. An Smap operation is generally cheaper than a comparable MEDUSA operation, which in turn is generally cheaper than a comparable STAROS operation. The cost difference in the execution of one invocation of an operation may be minute, or it may be orders of magnitude. Some of this cost difference derives from disparities in the service that the systems deliver. Another part of it is due to implementation issues.

An operating-system designer chooses what he or she believes to be suitable functional power, delivered at an acceptable cost. The desired result is a net savings in eventual, overall system usage. The three microcodes reflect different design decisions, even different attitudes about software development. For example, we have seen that Smap delivers better performance as measured in mapped references per second (see Section 3.1.3). Smap addressing is inadequate, however, for incorporation into an operating system because it allows any executing program to change addressability to gain write access to any word in all of Cm*. Both MEDUSA and STAROS provide additional protection, and they pay for it in time and space.

The design of an operating system reflects a number of trade-offs between functional power and cost. Although the operations of one system are faster, that does not mean that programs written using that system will run faster. Functionality not provided in the system may have to be implemented by the user. Indeed, it may have to be reimplemented by each user. In the case of Cm* systems, functionality not implemented in microcode may have to be implemented in the much slower medium of software.

To compare empirically two philosophies in operating-system design, one must compare two disparate implementations. The thoughtfulness of an implementation strategy, or the effort expended in optimization, can affect performance measurements in ways that distract from an evaluation of the philosophies. To appreciate the measurements that are presented in this chapter, something must be said about the degree to which the different systems have been tuned. Smap was written several years ago and has not been optimized since it was first written. Because the first implementation was coded fairly carefully and the system is reasonably simple, however, one cannot expect large improvements by making refinements.

Measurements of MEDUSA were based on the first version of its microcode. Execution efficiency was a primary goal, but critical functions have not been subsequently optimized. The initial version of the STAROS microcode, however, sacrificed execution efficiency to make it easier to write and maintain. Optimizations for speed were made in several microcode routines, and measurements in this chapter reflect some of these optimizations.

Almost all of the MEDUSA message system is microcoded, whereas approximately a third to a half of STAROS's message system is in software. This induces some rather glaring performance discrepancies, which could be largely mitigated by incorporating the remaining portions of the STAROS message system into microcode.

7.1. Microcode Measurement Techniques

Before proceeding to a presentation of the measurements themselves, some remarks about the experimental methodology are in order. To evaluate the performance of the microcodes, we have utilized two techniques, each with its own strengths and weaknesses.

The first technique for firmware (microcode) evaluation is real-time measurement, which is done by repeatedly running a program that performs a given microcode operation, usually several million times. The second is *tracing* through microcode, counting microcycles and references from the Kmap to the memory of the LSI-11s. The time required for a single microcode operation can be calculated by both methods.

There are two main advantages of real-time measurement. First, it accounts for queueing delays. Many interesting operations in both operating systems are fairly complex, requiring several passes through various queues in the hardware and firmware of the Kmap. Tracing fails to account for any queueing delays within the hardware, so trace measurements are only a lower bound. Second, real-time measurement can show the effects of contention. Because tracing cannot model hardware delays, it cannot show how the system slows down as the load on it becomes greater. It is useful to find out, for example, whether a particular operation performs acceptably when all the Cm's are making references to the same block of data. Only real-time measurement can provide the answer.

Tracing has different advantages. A real-time measurement yields only one number: the average elapsed time for the operation to be performed. A trace shows how much each microcode subroutine contributed to the total time, and thus indicates where optimization may be fruitful. It also is easier to perform a large number of traces than a large number of real-time measurements. A separate program must be written or modified to make each real-time measurement, whereas it is very simple to perform additional traces with a tracing program. Moreover, most microcode operations modify the state of the Kmap or its data RAM in some way so that consecutive operations in a real-time experiment may follow different paths through the microcode or even produce errors.

HOW REAL-TIME MEASUREMENTS ARE PERFORMED. A program that invokes the same Kmap operation repeatedly is run on one or more Cm's. It is loaded, and then controlled, using the NEST environment, which is described more fully in Section 9.1. This small executive allows the user to specify the number of iterations of the operation, to inquire about the progress of the experiment, and to compute the time consumed by an individual operation from data it displays after all the processors have finished.

HOW TRACING IS PERFORMED. Programs to perform tracing were written by Ed Gehringer in the fall of 1979. The first program, called CYCLES, performs traces of individual microsubroutines; the second then uses these traces to calculate how long a Kmap operation takes by summing the time to perform a microsubroutine and all the subroutines it causes to be called.

The first program reads in a microcode source file and builds a flow graph. Next it asks the user to select from a menu of microsubroutines to trace. It adds up the microcycles and memory references (to the memory of the Cm's) that would be performed when the selected microsubroutine was executed. At each branch point, it asks the user which branch to take.

It is often useful to perform several traces of the same routine to account for different values of parameters or different global conditions. For example, the routine that performs mapped memory references will follow different traces depending on whether the memory word is local to the cluster or in a remote cluster. The user is thus allowed to name the traces in order to distinguish them. Each time a microroutine calls another microroutine, the user is prompted for the name of the trace of the called routine.

VALIDATION OF TRACE DATA. Three factors contribute to the time consumed by a Kmap operation: Kmap microcycles, memory references from the Kmap to memory of one of the Cm's, and waiting time due to contention for resources. As noted above, tracing cannot measure contention, but it can give a load-independent lower bound on the time needed to perform an operation. In any event, we expect Kmap and memory contention to be of minor importance for most programs.

To determine how long a Kmap operation takes in the absence of contention, we must know the time consumed by a microcycle and a memory reference. A microcycle takes a constant 157 ns., as it depends on the quartz clock within the Pmap. The trace data tells us how many microcycles and memory references are encountered in a particular Kmap operation; if we know how long the Kmap operation takes, we can thereby compute how long a memory reference takes. One calculates how long a Kmap operation takes in the absence of contention by performing a real-time measurement with only one Cm in the cluster running, as there is then nothing to compete with that Cm for the Kmap or for memory.

From our initial real-time experiments, we concluded that a memory reference from a Kmap to a Cm took 4.3 μs. We then were able to use this value to predict how long other real-time experiments should take to run, based on the trace data. Five dissimilar Kmap operations were tested in this fashion. In all cases, the elapsed time was within 2.1 percent of that predicted, based on our value for a memory reference. These experiments allow us to list values for other Kmap operations without carrying out real-time measurements of those operations (many of them are not amenable to such measurement because they modify the state of the Kmap or data RAM). Microcode-measurement results helped the operating system projects choose which operations to optimize.

7.2. Performance of Similar Microcoded Operations

The previous chapter presented a comparison of mapped memory-reference performance in three of the microcode systems that were written for Cm*: Smap, MEDUSA, and STAROS. We are now ready to focus on some other equivalent, or at least similar, operations from each of the three systems. The selected operations are the ones that are executed relatively frequently by the operating system and user programs, and therefore the ones that will have the greatest impact on overall performance. This section considers the time costs of two such operations, then concludes with a survey of the space costs of various functions in the STAROS and MEDUSA microcodes.

The first operation, change addressability, allows a process to change a portion of its address space to make a new object accessible. How frequently this operation is executed in practice depends strongly on the application program, so its effect on overall performance is hard to estimate without additional data. It is, however, of concern, since the address space of an LSI-11 is only 16 bits. A typical synchronization operation was chosen for measurement because synchronization costs are important, especially for processes that share read / write data.

Most of the performance measurements in this section were obtained by real-time measurement. Those that were obtained by tracing are explicitly marked as coming from the CYCLES program. As noted above, measurements made by the two techniques are in close agreement with each other. Although the figures for operations measured on the hardware were computed by averaging over a large number of repetitions, strictly speaking, they have not been statistically validated because all the repetitions used the same hardware components. Variations between the speeds of interchangeable components could cause the figures to vary up to 5 to 10 percent for some of the operations (see Section 3.1.1, for example). For more complex operations that use a greater variety of components, variations in the individual components may average out, leading to a smaller variance than for simpler operations.

7.2.1. *Change Addressability*

The change-addressability, or ***Load Window***, operation redefines the binding between a window and the physical memory to which the window refers. Smap's version of the operation is the simplest and the fastest. It associates an arbitrary physical address with a specified window by loading the window register in the Kmap. In order to complete change of addressability, however, the LSI-11 must execute one more instruction to load the Slocal register corresponding to the window (StarOS and Medusa do this load automatically as part of the ***Load Window*** operation). To make the operations in the three systems more comparable, the 33 μs. quoted in the list below for Smap include the time to execute this extra instruction. The times for ***Load Window*** are as follows:

Smap	33 μs.	
Medusa	69 μs.	
StarOS	109 μs.	(*traced using Cycles*)

The costs listed above for both StarOS and Medusa are incurred on two separate occasions. The first portion of the cost is incurred when the operation is invoked by the LSI-11. At this time the name of the new object is bound to the window by loading the Kmap's window register and writing the name out to the process state kept in main memory. The second portion of the cost is incurred when the first memory reference is made through the window just loaded, in part due to the need to write the Slocal register. The extra time required by StarOS over Medusa is partly because of the three-level address structure of StarOS—two extra references to translate the C-list index into a capability—and partly because the entities that are read or written to memory during the operation are larger for StarOS than for Medusa. StarOS uses two-word capabilities, while Medusa uses one-word descriptor indexes.

7.2.2. *Synchronization Operations*

All three microcode systems provide indivisible operations that allow parallel programs to synchronize their operations on shared read / write data. The representative operation, ***Indivisible Increment***, has virtually the same semantics in all three systems. The operation costs are for the case in which the target of the increment is in the same cluster as the invoker of the operation. As is the case for the other operations in this chapter, any address information needed to perform the operation is assumed to be present in the Kmap's cache. The times for ***Indivisible Increment*** are as follows:

Smap (*immediate address parameter*)	25 μs.	
Medusa (*immediate address parameter*)	33 μs.	
Medusa (*virtual address parameter*)	47 μs.	
StarOS (*virtual address parameter*)	95 μs.	(*traced using Cycles*)

Smap's ***Indivisible Increment*** is faster than that of MEDUSA or STAROS because Smap makes no main-memory references to fetch the parameter for the operation; the parameter is a 16-bit immediate address that is passed up to the Kmap when the operation is invoked. MEDUSA has two forms of the operation; one uses an immediate address and the other a virtual address. The 14-μs. difference is due to the two additional main-memory references and associated microcode that reads the larger virtual address parameter. STAROS does not have an immediate address version, so its operation also pays for the memory references to read the parameter. STAROS's operation takes 48 μs. longer than MEDUSA's for two reasons. First, STAROS has a three-level address structure. Two additional references must be made during each operation to read the capability. Second, STAROS uses more high-level microcode procedures during the operation than MEDUSA. In particular, the parameter-block binding to the window is recomputed for every word that is read. STAROS permits a parameter block to be spread across two windows; MEDUSA does not.

7.2.3. Space Costs

Tables 7-1 and 7-2 show the number of microinstructions used by MEDUSA and STAROS for performing various operations. It is difficult to compare these numbers because the two microcodes vary in target architecture and structure and because different methods were used in classifying the instructions. For example, *indivisible operations* are included in STAROS's *miscellaneous user-level operations*, while *intercluster communication* is not distinguished in the MEDUSA classification scheme but is spread over the various parts of the system.

It can be seen from the two tables that the MEDUSA microcode occupies more space than STAROS. One reason for this is that trade-offs in MEDUSA generally were made in favor of speed at the expense of microcode space. The second reason is that the MEDUSA microcode also has implemented more functionality in the following ways:

Table 7-1 The Sizes of Various Portions of the MEDUSA Microcode

Function	Number of microinstructions
Messages and events	901
Common routines	750
Robustness	670
Addressing structure	610
Block transfers	367
Interrupt handling	285
Activity multiplexing	157
Indivisible operations	105
Total	3,845

Table 7-2 The Sizes of Various Portions of the StarOS Microcode

Function	Number of microinstructions	
Addressing structure		857
Capability manipulation	194	
Cached-descriptor maintenance, searching	153	
Deque / stack references, pointer manipulation	153	
Address mapping	126	
User-level capability operations	118	
Cached-window maintenance	113	
Message operations		326
Low-level operations and initialization		317
Miscellaneous user-level operations		310
Intercluster communication		241
Garbage-collection support		71
Total		2,122

Its message operations are more complex (see Section 7.4).

It performs memory reference retries on hardware errors, such as parity errors.

It has a more sophisticated exception-reporting mechanism.

Medusa implements a **Block Transfer** mechanism (Section 7.3.1), whereas StarOS does not.

The Medusa microcode provides substantially more support for activity multiplexing than the StarOS microcode.

The StarOS approach was based on the belief that a simpler, more modular system could be implemented first; additional speed improvement and functionality could be added once the system was being used and bottlenecks were identified. In that case, StarOS microcode could be extended for application experiments or performance monitoring. This would be more difficult in Medusa because it has used nearly all the control store.

7.3. Performance of Dissimilar Microcoded Operations

Medusa and StarOS each implement many operations that have no direct counterpart in the other system. In some cases, this is because of differences in their addressing structures (StarOS's capability operations, Medusa's ***Establish XDL***); in other cases, it is because Medusa has incorporated more functionality into its microcode (***Block Transfer***, for example). This section presents the performance of such operations, along with others that, although they exist on both systems, have been measured on only one system.

7.3.1. MEDUSA Operations

Block Transfer. A ***Block Transfer*** is a microcoded operation that transfers a number of words between the local memory of two Cm's. For large blocks of data, it is much more efficient than moving the words via nonlocal memory references, since a Kmap operation does not have to be reinvoked for each word transferred. Only MEDUSA incorporated ***Block Transfer***; a STAROS block transfer was designed but never implemented.

Table 7-3 gives performance statistics for MEDUSA's ***Block Transfer*** mechanism. Measurements are given in microseconds and as a multiple of the cost of a typical LSI-11 ***Load*** instruction (7.4 μs.) executed without any mapped references. All values were obtained using real-time measurement. ***Block Transfer*** is quite inefficient for transferring small amounts of data, since the overhead of setting up the ***Block Transfer*** dominates the execution time. For transferring more than 15 words, however, ***Block Transfer*** is faster than performing nonlocal memory references. The asymptotic value of the transfer time for large blocks, 10.3 μs. per word, compares favorably with the hardware-limited maximum bandwidth of 5 μs. per word.

Privileged Operations. There are several microcoded operations that may be executed only by MEDUSA utilities. Real-time measurements for three of these are given in Table 7-4. ***Establish XDL*** is used by a utility to map its external descriptor list (XDL) onto the PDL or SDL of another activity. It needs to do so in preparation for amplifying its rights to an object in one of these lists (see Section 5.1.3). ***Reawaken Sleeping Activity*** is used to reactivate a blocked activity that has just become runnable. ***Write Descriptor*** writes a descriptor indivisibly. It is a complicated operation because it must synchronize with other ***Write Descriptor***s, ***Read Descriptor***s, and Kmap contexts that may be using physical addresses derived from the target descriptor (see Section 5.1.2).

Table 7-3 Timing of MEDUSA ***Block Transfer***

Number of words	LSI-11 instr. equivalents	μs.	μs. per word
1	34	251	251.0
10	46	342	34.2
20	61	449	22.5
40	87	647	16.2
60	114	845	14.8
80	145	1,075	13.5
100	172	1,273	12.7
200	310	2,295	11.5
2,000	2,801	20,728	10.3

Table 7-4 Timing of Some Privileged MEDUSA Operations

Operation	LSI-11 instr. equivalents	μs.
Establish XDL	3	23
Reawaken Sleeping Activity	14	106
Write Descriptor	82	610

Measurements of Operations Under Load. Let us now focus attention on how the time required for certain operations is affected by the system load. The three operations that we consider—***Read Word***, ***Indivisible Increment***, and ***Conditional Receive***—are not unique to MEDUSA, but their STAROS counterparts have not been measured under load. ***Read Word*** reads a word from a specified object without the need to make it addressable first via a ***Load Window***. ***Indivisible Increment*** (Section 7.2.2) reads a word and increments it without allowing any other activity to access it in the meantime. ***Conditional Receive*** is the nonblocking message operation introduced in Section 5.2; further measurements of it will be presented later.

The elapsed time needed to perform each of these operations was measured as the number of processors performing them was increased. The activities were configured to generate maximum contention in each case. For ***Read Word***, all processors referenced the same memory location; for ***Indivisible Increment***, the same location was incremented by all the processors; and for ***Conditional Receive***, all processors sent messages to the same pipe. Each activity was executed on a separate processor. Figure 7-1 shows that all three curves have similar shapes, despite the large disparities in execution time. The time needed to perform an operation remains nearly constant until the effects of contention begin to be felt, then increases linearly in the number of activities. Because these operations are fairly representative of the microcode operations as a whole, other operations are likely to show similar performance.

7.3.2. STAROS Operations

Operations on Some Representation Types. Several of STAROS's representation object types were described in Section 6.1.1. Among these were deques and directories. A *stack* is a special type of deque whose ***Push*** and ***Pop*** operations are always done from the front. Table 7-5 gives sample timings for these types. All timings were derived by tracing, are for in-cluster references, and assume no Kmap or memory contention in the destination Cm. Timings for mailbox operations will be presented in Section 7.4.

Capability Instructions. *Capability instructions* are implemented in STAROS microcode primarily to protect capabilities from unauthorized manipulation by

Figure 7-1 Performance of Three MEDUSA Operations Under Load

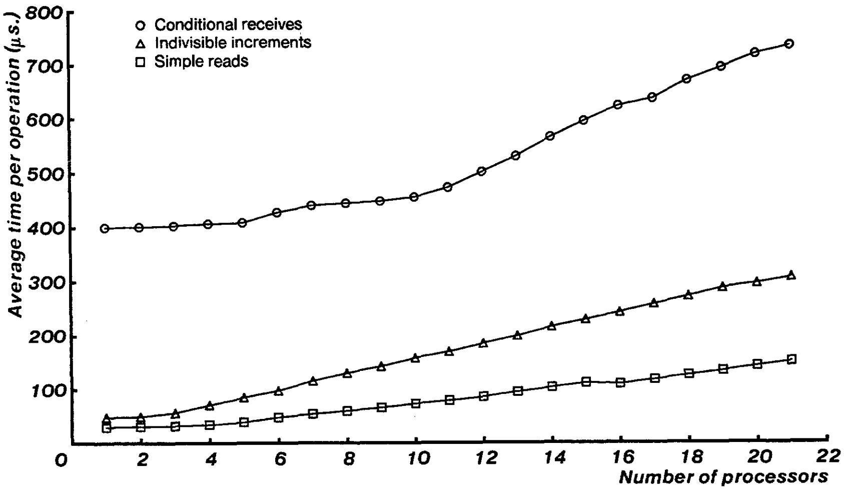


Table 7-5 Timing of Operations on STAROS Representation Objects

Object type / operation	LSI-11 instr. equivalents	μs.	# refs. to local memory of Cm's
Stack			
Push	4	32.6	4
Pop	5	39.8	5
Deque			
Push onto front	6	46.4	6
Push onto rear	6	47.1	6
Pop from front	5	39.3	5
Pop from rear	5	40.1	5
Directory			
Read	2	15.5	2
Write	3	22.1	3

software (Section 6.1). A microcoded implementation also serves to give them considerable speed, helping to make the object-oriented STAROS design more competitive with less structured memory organizations. It is instructive to consider the functionality as well as the performance of these instructions.

Create Capability (Representation type) May be invoked only by the *Object Manager*. Manufactures a capability for a newly created object of a representation type.

Create Capability (Data type) May be invoked by any user to create a data capability.

Restrict Capability "Turns off" an arbitrary set of the 13 permission bits in the rights word, diminishing the power of the capability. This operation also may be used to delete a capability by turning off all 3 bits in its type field.

Amplify Capability Converts an abstract capability into a representation capability. Requires a type token for the type of the abstract object. The most time-consuming part of this operation is comparing the type token with the object type, which is stored in the object's descriptor but is not cached.

Deamplify Capability This is the inverse of ***Amplify***. It replaces a representation capability with an abstract capability. A type token is required to prevent functions other than the type manager from creating abstract capabilities of a particular type.

Copy Capability Copies a capability from one place in the address space to another.

Transfer Capability This is a special form of ***Copy Capability*** that places a new copy of a capability somewhere in the address space, then deletes the old one.

Read Capability Copies information from a capability into the data portion of some object so that a process may "read the bits" to determine, for example, the type of a capability or the rights associated with it. This does not subvert protection because there is still no way for any process to store an arbitrary bit pattern into a capability. ***Read Capability*** also furnishes information such as the size of the object, which it gleans from the descriptor. (Often part of the descriptor is cached in the Kmap's data RAM, as described in Section 4.2. If not, it must be fetched from main memory.)

Table 7-6 presents timings for the capability instructions. All timings were derived by tracing.

Table 7-6 Timing of StarOS Capability Instructions

Operation	LSI-11 instr. equivalents	μs.	# refs. to local memory of Cm's
Amplify Capability	18	133.4	13
Copy Capability	13	99.2	10
Create Capability			
Representation	16	120.9	12
Data	12	86.2	9
Deamplify Capability	18	133.4	13
Read Capability	11	78.1	8
If descriptor is not in cache	16	120.9	12
Restrict Capability	9	67.2	7
Transfer Capability	18	130.7	13

We see that any capability operation or change of addressability can be performed in 23 average LSI-11 instructions or less. It should be noted that this mechanism provides both an expanded address space and support for program modularization at a relatively low cost. Note that unmapped references have no overhead and that the second and subsequent mapped references require no indirection through a capability. For example, the distributed partial differential equations, or PDE, application (see Section 11.5.3) incurs no performance penalty for having the benefit of capability addressing in lieu of the more restricted two-level descriptor-based addressing offered by Smap and MEDUSA.

The cost of object addressing is likely to be significant only for processes whose working set exceeds the capacity of the window registers, resulting in frequent invocations of ***Load Window***. Such processes would expect to incur substantial overhead from any mechanism that attempts to relieve the constraints of the LSI-11's small address space.

It is worthwhile to scrutinize an individual operation to assess how resources are expended in performing the operation—in particular, to account for the seemingly large number of references to the memory of Cm's during the operations listed in Table 7-6. First, observe that execution time for a STAROS microroutine is divided approximately equally between Pmap microcycles and memory references.

Consider the ***Amplify Capability*** operation, which performs 13 memory references. It takes two parameters: a capability index that tells which capability to amplify and a pointer to a type token that matches the type of the object named by the capability. These two parameters are contained in a parameter block (Section 4.2). The first memory reference by ***Amplify Capability*** reads the *processor data*, the address of the parameter block (memory reference 1). Then the capability index (2) is read from the parameter block. Using this index, the capability itself (3, 4) is read. A special value called a *plug* is written into the capability's type field (5) to flag the fact that this capability is in the process of being modified so that no other Kmap operation attempts to modify it in the meantime.

The abstract type (6) of the object named by the capability is read from the descriptor for the object because abstract-type names are not kept in the cache. The second parameter (7) names a type token, which is then read (8, 9). The amplified capability (10, 11) is written back, overwriting the plug that says the capability is being operated on. Then a zero is written into the first word of the parameter block as an indication that the operation has been successfully completed (12), and the Cm is awakened (13).

Most other capability instructions follow a similar sequence of actions, though they are somewhat cheaper because there is no need to read a token and—if the capability is not to be rewritten or deleted—to write a plug. Since the microcode performs some of the functions of a conventional executive or supervisory program, it is not surprising that a microcode function has corresponding entry and exit overheads to read parameters and write a result code. This overhead is typically four or five mapped memory references (about 35 to 45 μs. or 5 to 6 locally executed LSI-11 ***Load*** instruction equivalents). The corresponding overhead for a function implemented by the *Nucleus* process is about 750 μs., or approximately 100 instructions.

7.3.3. Interpreting the Results

The comparisons presented so far in this chapter are a good starting point for predicting the performance of an application on one of the Cm* microcode systems. Nevertheless, they do not tell the whole story because they fail to account for several factors. First, some costs of invoking an operation are not reflected in the performance figures of this section. An example is the cost of setting up a *parameter block* to perform the operation. Because the LSI-11 is considerably slower than the Kmap, setting up the parameter block can sometimes be as expensive as the operation itself, especially if the Kmap operation is a short one. A discussion of the parameter-block costs in StarOS and a comparison to an operation cost was presented in the previous section.

Another consideration in extrapolating from measurements to overall system performance is that most of them have been made under no load. The exception is mapped memory references, which have been measured under both the artificial loads (Section 3.1.3) and the applications discussed in Section 3.1.4. Some measurements of additional operations under loaded conditions are discussed in Section 7.3.1.

Finally, Medusa and StarOS have some fundamental differences. To cite one example, names in Medusa are always interpreted relative to an activity. In contrast, a StarOS capability is a global name. One implication is that in Medusa communication is virtually always "by value," whereas in StarOS communication can be either "by value" or "by reference." It is difficult to evaluate such differences between Medusa and StarOS. For example, differences in functionality will cause users to design and program their applications differently in the two systems. Because Cm* is a multiprocessor, the task decomposition itself may be influenced. In addition, the user will adapt his implementation to improve performance in the context of the operating system being used, so the implementation of an application on the two systems may look quite different. Certainly it is possible to observe two functionally equivalent task forces on the two systems and decide which one finishes faster. It is more difficult to compare the robustness of the systems and the effort expended in implementing them.

One of the lessons learned from implementing a number of different microcode systems is that careful attention must be paid to low-level microcode design to ensure that the system will perform adequately under loaded conditions. Otherwise, the system may become deadlocked over shared resources, or requests may be starved if they are not served in a fair manner. The algorithms and system structures used to avoid these problems sometimes may extract a heavy penalty if they are not designed with efficiency in mind, and the overall performance of the system may suffer drastically as a result. An example of this phenomenon has been described in Section 3.1.3.

Another observation is that simple operations such as memory references can be performed efficiently. Intelligent use of caching and of special-case microcode to handle memory references seem to make the cost of such references fairly independent of the complexity of the address structure. It is important to note, however, that a complex address structure does exact a price for the more complicated operations because it is not feasible to write special-case microcode for all such operations.

The writing of special-case microcode to improve performance is an example of a space / time-complexity trade-off that was encountered frequently in the writing of all three systems. If a particular operation is to be optimized for speed, then it often helps to write microcode that is specially adapted to the needs of that operation. Special-case microcode not only takes up more space, however, but it makes the code less structured and therefore harder to modify and debug or to adapt for the purposes of experimentation.

7.4. Message-System Performance

Next we focus our attention on the message-communication operations of MEDUSA and STAROS. Smap does not implement messages. Both operating systems rely on message communication where more conventional operating systems use procedure activation. Messages are used for general-purpose communication and synchronization of processes. In addition, both operating systems are structured so that asynchronously executing "servers" provide a substantial portion of the system functions. A request to such a server is in the form of a message that conveys the requisite parameters. Likewise, any reply is transmitted as a message. Not only do both operating systems offer a message-communication facility to user programs, they also make extensive use of it internally. Performance of the message facility is therefore crucial to the efficient operation of both systems.

7.4.1. Functional Comparison

Message communication is the least similar of the facilities provided by the microcodes of MEDUSA and STAROS. The two differ both in function and in implementation. The message mechanisms have been described in earlier chapters, but it will be helpful to review them.

Similarities. Both systems support buffering of the messages that are transmitted between processes. Both provide two kinds of operations: conditional and unconditional. Conditional operations run to completion regardless of the state of the mailbox or pipe that is the target for the operation. For example, when a process performs a ***Conditional Receive***, a message is returned from the mailbox if the mailbox is nonempty, and a status indicating an empty mailbox is returned if the mailbox is empty. In either case, the process continues processing after performing the operation. The semantics of unconditional operations are rather different in the two systems and will be discussed below.

Differences. The two message systems differ both in semantics and in implementation. It is important to keep this in mind because some large performance differences are simply an artifact of whether particular operations are microcoded and have little to do with underlying design decisions. Another difference concerns an "optional extra" of the MEDUSA message system—a feature that could be added to the STAROS message system without changing its basic design.

- Semantic differences:

Value versus reference semantics. In MEDUSA, messages are always transmitted by value; that is, data is physically moved from the address space of the sender to the address space of the receiver. No provision is made for using a message to pass a capability or descriptor that would allow the receiver to access memory belonging to the sender. Through its data mailboxes, STAROS also allows message transmission by value. Because only one 16-bit word at a time can be sent in this way, however, it would be impossibly inefficient to use data messages to convey large amounts of information. Consequently, large amounts of data are passed by sending a reference (a capability) that allows the receiver to access a portion of the sender's memory. Hence the MEDUSA message system is value based, while STAROS's is primarily reference based.

Message size. This distinction is closely related to the value versus message based dichotomy. STAROS transmits either a one-word data message or a single capability, which is the name of an arbitrary object. MEDUSA transmits variable-sized data messages ranging from 0 to 4,000 bytes by copying the message. Since mailboxes and pipes have fixed sizes, the size of messages affects how many of them can be buffered simultaneously. The buffering capacity of a STAROS mailbox ranges from 1 to 2,044 data messages or 1 to 252 capabilities. The buffering capacity of MEDUSA pipes is a function of message size; it ranges from 1,000 messages for zero-byte messages to a single message if the message is larger than 2,000 bytes.

Automatic versus voluntary blocking. The difference between the unconditional operations provided by the two systems is that when an operation cannot be performed immediately, the invoker of the operation is blocked automatically in MEDUSA; in STAROS, blocking is a separate operation that is invoked explicitly by a process.

Symmetry. The semantics of the ***Send*** and ***Receive*** operations in MEDUSA are symmetric. For example, the effect of doing an ***Unconditional Receive*** on an empty pipe is the same as the effect of doing an ***Unconditional Send*** on a full pipe—the invoker is blocked. In STAROS, ***Send*** and ***Receive*** are not symmetric because the ***Unconditional Receive*** does not have a counterpart. If a ***Send*** is done to a full mailbox, the invoker is informed that the operation cannot complete immediately.

Overlapped delivery. Because a STAROS process may have several unsatisfied unconditional requests to receive a message outstanding, delivery of several messages may be overlapped. In MEDUSA, messages arrive in the receiver's address space sequentially and while the activity is actively performing the ***Receive*** operation, or they are suspended awaiting the time when the ***Receive*** in progress can complete.

- Implementation differences:

Message delivery. The delivery of a message to an activity that has performed an ***Unconditional Receive*** on an empty pipe is handled entirely within Kmap microcode in MEDUSA. The corresponding delivery of a message to a process that has done a ***Registered Receive*** in STAROS (called ***Portal Delivery***) is implemented in operating-system software.

Blocking. STAROS blocking is performed by *Nucleus* software; MEDUSA blocking is integrated into the microcode for message operations.

Block transfer. All data transfers performed during the message operations of MEDUSA are handled by the microcoded ***Block Transfer*** mechanism. Block transfers are not used in the STAROS message mechanism, since blocks of data may be transferred, in effect, by passing a capability.

- "Optional extra":

Pause time. In MEDUSA, when an activity is blocked as a result of a message operation, the activity does not relinquish its processor immediately. Instead, it retains its processor for a period called the *pause time* (see Section 5.2), which can be specified by the activity. If the activity becomes runnable during the pause time, it is able to resume processing immediately without incurring the context-swap time of the LSI-11. In principle, pause time could be added to the ***Unconditional Receive*** of STAROS by adding a pause-time parameter to the invocation. If set to zero, this parameter would yield the current semantics of ***Unconditional Receive***.

Methods of Comparison. The performance of message systems can be compared in at least two ways. One way is to measure performance by the time it takes to perform individual operations such as ***Send*** and ***Receive***. Such a characterization provides an incomplete picture because nontrivial uses of a message mechanism require additional operations, such as setting up the message at the sender's end and reading the message at the receiver's end. In some cases, explicit synchronization is necessary beyond that provided by the message system. Message communication is fundamentally an interaction between two or more parties, so the performance measures should reflect the process-to-process interaction time.

This is not to say that raw measurements of individual operations are without value. Measurements of interactive communication are based on a particular sequence of operations performed by the sender and receiver, and while many processes may use this sequence, other processes may use a completely different paradigm. In this case, raw measurements may be a more useful estimation tool than process-to-process interaction times. We will consider message-system performance from both standpoints.

7.4.2. *Performance of Message Operations*

One major factor influencing message-operation timings is whether the message is buffered in the mailbox or pipe. Recall from Section 4.1.2 that if a ***Send*** is performed when a receiver is waiting for the message, the message need not be placed ("buffered") in the mailbox or pipe but can be copied directly to the address space of the receiver. First, consider the case in which no receiver is waiting, so buffering must occur.

When Buffering Occurs. This section discusses performance of the message operations in a situation where the operation can be completed by merely manipulating the mailbox or pipe (i.e., the invoker need not wait for some action by another process or activity). The operations chosen for measurement are ***Conditional Send*** and ***Conditional Receive***.

Timing measurements were taken of a single process or activity that first performed a ***Conditional Send*** of a message to an empty pipe or mailbox. As a result, the message is buffered. The process or activity then performs a ***Conditional Receive*** from the same pipe or mailbox. The ***Conditional Receive*** removes the message from the mailbox or pipe, leaving it empty, and returns the message to the invoker. Thus the message is copied twice. The StarOS operation was measured using both a capability message and a data message. The Medusa message consisted of a single data word.

Measurements were made using two different distributions of the various entities involved in the operations: in the *local cluster* case, the sender, the message, and the mailbox are all kept in the same cluster; in the *nonlocal cluster* case, the sender and message are in one cluster, and the mailbox is in a different cluster. The tables below show the times for ***Conditional Send*** and ***Conditional Receive*** computed from the total send-plus-receive time obtained by measurement. The cost of the send-and-receive sequence was assumed to be distributed between the ***Conditional Send*** and ***Conditional Receive*** in the same ratio as the costs of these operations that were computed from Cycles traces.

Cost for ***Conditional Send***:	*Local cluster*	*Nonlocal cluster*
Medusa	336 μs.	339 μs.
StarOS (*data*)	110 μs.	146 μs.
StarOS (*capability*)	151 μs.	196 μs.

Cost for ***Conditional Receive***:	*Local cluster*	*Nonlocal cluster*
Medusa	341 μs.	342 μs.
StarOS (*data*)	118 μs.	156 μs.
StarOS (*capability*)	180 μs.	225 μs.

The time to perform the operation in Medusa is roughly three times slower for the local cluster case and about two times slower for the nonlocal cluster case. There are three reasons for this difference in performance. First, the ***Block Transfer*** mechanism adds considerable overhead to the cost of moving small messages. Second, a deadlock-and-starvation algorithm is executed to ensure that the three contexts allocated during each operation are acquired in a safe fashion. Finally, some cost is incurred because code to handle waiting is executed, even though it is not necessary in this particular case.

When the Buffer Is Bypassed. This section presents message-system performance measurements for the case when one process (activity) has to wait for another process to act. In particular, consider a two-process message interaction that involves sending a message from one process to another process that is waiting after having done an ***Unconditional Receive*** on an empty mailbox or pipe.

Assume that the mailbox or pipe is empty when a receiver unconditionally requests a message. The message is a single data word. For StarOS the times are constant. Both Medusa operation times are a function of the size of the message. The duration of the ***Unconditional Receive*** also depends on how much of the pause time elapses before the message arrives. Duration of ***Send*** and ***Receive*** operations in the waiting case are as follows:

Medusa ***Send***	490 µs.	*(estimated)*
Medusa ***Receive***	250 µs.	+ elapsed pause time *(estimated)*
StarOS ***Send***	2,000 µs.	*(estimated)*
StarOS ***Receive***	166 µs.	*(traced using Cycles)*

The StarOS ***Send*** time is dominated by the cost of software implementation of message delivery. It should be noted that during a Medusa message operation, both the sender's and receiver's processors enter a pause state during which they may process interrupts. The sender's processor is in the pause state during the ***Block Transfer***; the receiver's processor is in the pause state for the duration of the pause interval.

7.4.3. Process-to-Process Interaction Times

One aspect of the semantic differences between the StarOS and Medusa message systems is the comparative performance of value-based and reference-based message communication. As a first attempt, let us include the time it takes to transfer the message plus the time that the receiver must spend to access the words transported. Thus StarOS measurements include the time used to make the words addressable via the capability used in the transfer.

If we confine our attention to messages that are in the form of blocks of data, we can analyze the trade-off between passing pointers (pass-by-reference) and copying data (pass-by-value). Because nonlocal accesses are significantly costlier than local accesses, it is not obvious at what point it becomes cheaper to move the data itself instead of accessing it nonlocally.

Processing Efficiency. Two parameters determine which mechanism is more efficient. The more important of the two is the number of times each data item is referenced by the receiver. We will call this parameter *frequency of use* and examine three regions: *sparse use*, where only a small fraction of the words transported are actually referenced; *moderate use*, where each word is referenced exactly once; and *heavy use*, where each word is used several times by the receiver. The second parameter is the number of words that need to be transported. We consider two cases: the time needed to communicate one word and the time to communicate 2,048 words.

SPARSE USE. Assume that only a very small fraction of the words transported are ever referenced. This would be the case, for example, if the data transferred was sorted and the receiver was doing a binary search to locate a particular value. The times for 1 word and for 2,048 words are given below. There is no break-even point because the reference mechanism is always better.

	1 word	*2,048 words*
MEDUSA	484 μs.	21,000 μs.
STAROS	228 μs.	440 μs.

MODERATE USE. Let us assume that each word transported is referenced exactly once. Under this usage pattern, the factor that determines which scheme is better is the cost of the nonlocal access. For the local cluster case, it is always better to pass a pointer, although the cost difference is small. In the nonlocal cluster case, the choice is influenced by whether the message is buffered in the MEDUSA pipe or not. If data is not buffered, it is better to move the data, as long as more than 8 words are being transported. If data is buffered, then it is better to move the data only if it is longer than 1,345 words.

One frequent use of messages is to communicate the parameters for a function invocation. Such messages are expected to be of modest size and to receive moderate use. It is cheaper to send such a message by reference, except when the message is buffered and transported to a remote cluster.

	1 word	*2,048 words*
MEDUSA (*no buffering*)	484 μs.	27,000 μs.
MEDUSA (*buffering*)	677 μs.	48,000 μs.
STAROS (*local cluster*)	228 μs.	21,000 μs.
STAROS (*nonlocal cluster*)	320 μs.	73,000 μs.

HEAVY USE. Assume that each word transported is referenced many times, so it always pays to move the words. The table below lists the costs incurred by the two systems. For STAROS, we assume that the ***Block Transfer*** is coded in software, using the most efficient implementation permitted by the LSI-11 processor. The break-even points for the intracluster and intercluster cases are 28 words and 5 words, respectively. That is, if more than 28 words must be transferred intracluster or more than 5 words intercluster, then it is cheaper to move them in MEDUSA than in STAROS.

	1 word	*2,048 words*
Medusa	484 μs.	27,000 μs.
StarOS (*local cluster*)	228 μs.	45,000 μs.
StarOS (*nonlocal cluster*)	320 μs.	90,000 μs.

Pure Message Communication. Shared memory and message passing are the two means of interprocess communication. Processes communicate via one means or the other, or via a combination of both. Let us define *pure message communication* as communication in which interactions between sender and receiver are guaranteed to occur only via messages. There is no instant in which both sender and receiver have simultaneous access to any variable, and hence there is no need for locks, critical sections, or other synchronization mechanisms to coordinate access to shared variables.

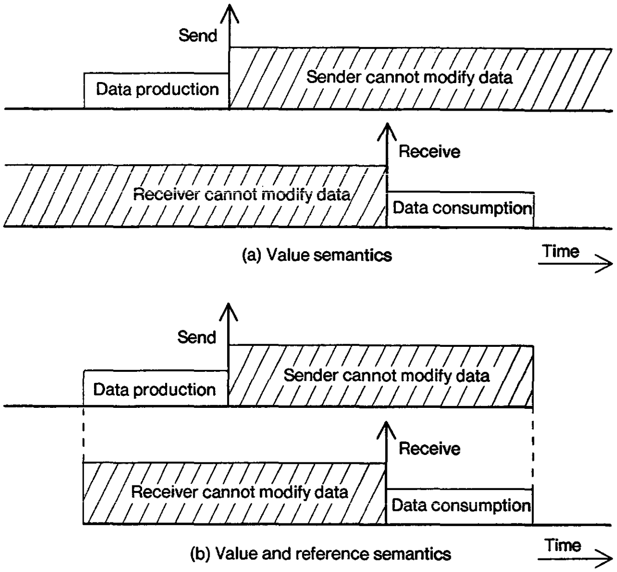
Message passing by value implements these semantics more naturally than message passing by reference. In a value-based mechanism, data is copied physically from sender to receiver. There is no time at which both processes can access the data. In a reference-based mechanism, however, what is communicated is a pointer that can be dereferenced to access the data. Since both sender and receiver can dereference the pointer, the absence of any restrictions on its use can make a reference mechanism degenerate into communication using shared memory. The receiver may inadvertently write into a message being prepared by the sender, and the sender may inadvertently write into a message being used by the receiver. There is no guarantee that interactions between sender and receiver are restricted to occur at specific instants.

To determine what restrictions must be placed on a reference-based mechanism so it satisfies the above guarantee, it is instructive to look at a value-based mechanism. Such a mechanism provides the above guarantee, although the exact conditions it satisfies are more restrictive than is really necessary. In a value-based mechanism, once a process sends data in a message, it has no way of *ever* modifying or accessing that data—the receiver has exclusive access to the data for an arbitrarily long time after receiving it. Similarly, before a process has received the data in a message, it has no way of *ever* modifying or accessing the contents of that message—the sender has exclusive access to the data for an arbitrarily long time before sending the data. The above conditions guarantee exclusive access to the data for both the sender and the receiver for a semi-infinite time interval (Figure 7-2a). It is easy to see, however, that the sender needs exclusive access to the data only while it is preparing the data, and the receiver needs exclusive access only while it is using the data (Figure 7-2b). Thus we have the following conditions that a value-based mechanism satisfies trivially and a reference-based mechanism must be made to satisfy before it can provide pure message communication:

C_1: *After a process sends a message* m, *there is no way for the process to modify the data communicated by* m, *until the receiver is done using the data.*

C_2: *Before a process receives a message* m, *there is no way for the process to*

Figure 7-2 Conditions for Pure Message Communication


Send
Data production
Sender cannot modify data
Receive
Receiver cannot modify data
Data consumption
(a) Value semantics
Time
Send
Data production
Sender cannot modify data
Receive
Receiver cannot modify data
Data consumption
(b) Value and reference semantics
Time

modify or access the data communicated by m *once the sender has decided to prepare the message* m.

This is not to say that *pure* message communication is the only useful interprocess-communication mechanism. Sometimes shared memory is the most efficient or most straightforward means of programming an algorithm. Consider the binary search that we presented as an example of sparse use. Certainly the reference-based mechanism is more efficient and for many parallel search algorithms, the fastest implementation may be to pass each process reference to the portion of the database it is to search.

Another case in point is the way scheduling is performed in StarOS (Section 6.5). Capabilities for process objects are mailed between the *Process Creator*, run queues, and schedulers as the system load changes dynamically. It would make little sense conceptually—let alone from the standpoint of efficiency—to copy (pass by value) a major portion of the process state each time a process was rescheduled. Pure message communication is important, but the efficacy of a message system does not stand or fall on how efficiently it provides pure communication.

New Measures: Latency and Throughput. In the above, our first attempt at measuring process-to-process interaction time included the time it takes to transport the message and the time for the receiver to read the message. For a more complete measure, we should include the time for the sender to produce the data for the message and, if pure message communication is desired, the synchronization operations necessary to preclude simultaneous access to the data by the sender and receiver. Only if these factors are included can measurements of message communication adequately reflect its interactive nature.

As an example of where speed of interaction is important, consider the use of a message mechanism to request the servicing of some time-critical function. What is important in this case is for the message mechanism to deliver the message rapidly from the requester to the server. The time it takes to do the ***Send*** and the time needed to do the ***Receive*** are of little importance. When response is important, the time to perform the ***Receive*** operation is completely irrelevant because the receiver presumably has already done the ***Receive*** and is waiting to service the message as soon as it arrives. The time to do the ***Send***, as viewed by the sending process, often is completely unrelated to the end-to-end transmission delay. This is the case, for example, in a message mechanism that buffers the message locally and allows the sender to proceed immediately after the message has been accepted. If the message goes through several intermediate buffering steps before arriving at the receiver, the end-to-end latency will be much larger than the time it takes to send the message. In addition, the cost of the ***Send*** alone does not include the time for a context swap that may be needed to get the receiver to start execution, although this time must be included in the measurement of end-to-end delay.

Latency L and throughput T can capture the performance of an interaction between sender and receiver. Latency and throughput have been used frequently to represent the performance of computer systems and computer communication networks [Kleinrock 75], and it has been suggested [Stone 75] that these are fundamental measures of performance evaluation. *Latency* represents the time it takes to deliver a minimal amount of information from one process to another. *Throughput* measures the speed, in bits per second, with which a large amount of data can be delivered from one process to another.

Latency can be used to predict the performance of programs in which control must be transferred frequently from one process to another. Examples of such programs are those that are set up using the requester-server paradigm, processes that handle I/O in a conventional computer system, and processes that handle time-critical functions in a real-time system. Latency is also a critical component in the time to execute a remote procedure call in a network environment [Nelson 81].

By contrast, programs that require transferring large amounts of data between processes tend to stress the data-communication component of a message mechanism, and the performance of such programs can be predicted using the throughput measure. The principal examples of this type of usage are programs for transferring files. Not all uses of a message mechanism fall into one or the other of the above categories, however, since applications such as real-time processing of speech stress both the control-transfer and the data-communication dimensions.

In order for the measures of L and T to measure their interactions, they must be measured from the point of view of a *user* of the message mechanism. The experimental setup for measuring L and T consists of two user processes, a sender and a receiver, in different Cm's, communicating with each other via messages. For both setups, communication is constrained so that it satisfies the conditions for pure message communication. The code at the sender for sending a single message looks something like Figure 7-3. At the other end, the receiver performs the set of operations in Figure 7-4 to receive a single message.

Note that both the sender and the receiver perform a number of operations in addition to the ones that send or receive messages. For the sender, the extra operations involve writing *b* bits into the message, where *b* is the size of the message, and setting up the parameters for the ***Send*** operation. For the receiver, extra operations involve setting up the parameters for the ***Receive*** and reading *b* bits from the message just received. For the purposes of this evaluation, assume that the message is read exactly once, as it was in the "moderate use" case at the beginning of this section.

The *AwaitSignal* step in the code for the *Sender* process indicates that, in the case of reference-based pure message communication, the sender must await notification that the receiver is finished using the object that holds the previous message before it begins to write a new message. *AwaitSignal* and *GiveSignal* can be implemented most efficiently as a *return* message sent from the *Receiver* process to the sender. An alternative implementation would be to use a new object to hold each message. While in principle this allows greater concurrency by permitting the sender to start writing a new message before the receiver is finished reading the previous one, the costs of dynamic memory allocation in StarOS, as in most systems, outweigh the savings.

Note that for reliable value-based message communication, a return is also desirable as an acknowledgment that the sender's message was received. The

Figure 7-3 Code for Sending a Single Message

```
process Sender =
begin
        AwaitSignal;                    { in pass-by-reference version only }
        WriteMessage(SizeOfMessage);
        SetUpSendParameters;
        Send(SendParameters);

end
```

Figure 7-4 Code for Receiving a Single Message

```
process Receiver =
begin
        SetUpReceiveParameters;
        Receive(ReceiveParameters);
        ReadMessage;
        GiveSignal;                 { in pass-by-reference version only }
end
```

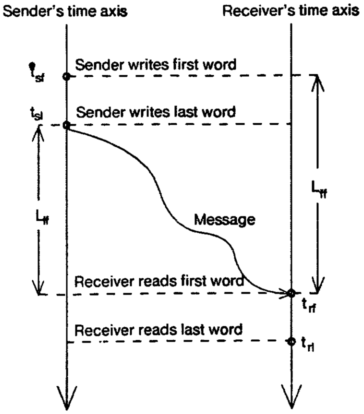
value-based measurements below do not include the cost of a return message, although there is reason to believe that the cost would be less than in the case of reference-based communication, since the receiver can send the acknowledgment immediately after receipt of the message, allowing transport of the acknowledgment to overlap its reading of the message.

***L* Measures**. A latency (*L*) measure for a message mechanism is defined to be the elapsed time from the moment a *Sender* process *decides* to send a message to the time that the receiver of the message can *meaningfully act* on it. Such a measure is useful in characterizing the performance of a message mechanism in cases where response is important or where processes interact relatively frequently using short messages. Since we expect the majority of the uses of a message mechanism to be of this type, we will consider the *L* measure to be the more important of the two measures, although both the *L* and *T* measures are required to present a complete picture of performance.

For an *L* measure to be well defined, it is necessary to specify that the receiver is waiting for the message. (If the receiver is not waiting, the latency depends on what the receiver is doing when the message arrives, and therefore it does not have a well-defined value.) This constraint on the definition of latency is also reasonable when we consider that in applications where low latency is important, a receiver (server) typically will be programmed so that it is faster than the sender (requester). In such cases, a message arriving at the receiver will find the receiver waiting.

In the above intuitive definition of latency, there are two significant instants: one along the sender's time axis when it decides to send the message, and the other along the receiver's time axis when it can first meaningfully act on the message. Figure 7-5 illustrates the range of possibilities for each of these two instants. The horizontal direction in the figure represents the separation between the sender and

Figure 7-5 Latency of Message Communication



receiver, and the vertical direction represents time. On the sender's time axis, t_{sf} is the time when the sender first writes into the message to be sent, and t_{sl} is the time when the sender last writes into the message. The time when the sender decides to send the message may lie anywhere in between t_{sf} and t_{sl}. On the receiver's time axis, t_{rf} and t_{rl} are the times when the receiver first reads the message after receiving it and last reads the message, respectively. The time when the receiver can first meaningfully act on the message may lie anywhere between t_{rf} and t_{rl}.

If we fix the time at which the receiver can first meaningfully act on the message to be t_{rf}, we can define two latency measures, L_{lf} and L_{ff}, which correspond to the extremes for the definition of when the sender decides to send the message. The last-touch to first-touch latency (L_{lf}) measures the time from the moment that the sender last touches the message to the moment when the receiver first touches the message. It is the latency of the message mechanism in isolation, since it does not include work that must be done to set up the message. This measure is useful in situations where the data to be communicated is known beforehand and therefore the setup cost does not need to be included in the measurement of latency. The first-touch to first-touch latency (L_{ff}) is useful in situations where the data to be transferred is not known in advance, and therefore the process must do all the work to set up the message *after* deciding to send the message. Consequently, the setup time becomes critical to achieve low latency.

Experimental determination of L_{lf} and L_{ff} can be somewhat difficult because it involves measuring a relatively short time interval whose starting point occurs on the sender's time axis and whose end point occurs on the receiver's time axis. This measurement is further complicated by the fact that when the sender's message arrives, the receiver must be waiting for the message. One way to make the measurement is to program both the sender and the receiver in a loop so that the sender's loop takes significantly longer than the receiver's loop. This ensures that the receiver will be waiting when the message arrives. Two hardware monitors are needed to make the actual measurement. One detects the event on the sender's time axis and the other on the receiver's time axis. The latency can then be determined by displaying the signals from the two monitors on a display device such as an oscilloscope or logic analyzer.

***T* Measures.** A throughput (*T*) measure for a message mechanism is the average rate at which the mechanism can communicate a large amount of data from one process to another. There are at least three different ways to define the amount of data communicated, and consequently three different definitions of throughput.

The first defines the amount of data communicated to be the number of bits physically transported by the message mechanism. The argument here is that it is inappropriate to credit the message mechanism with either more or less data than it actually transports. For a value-based mechanism, this would be the number of bits of data that are actually communicated; for a reference-based mechanism, it would be the number of bits used to encode a reference to the data to be communicated. The problem with this definition is that it is at too low a level. The definition of how much data is communicated depends on the semantics of the message mechanism, so we reject this definition as being inappropriate.

The second definition is based on the intent of the sending process. Here the amount of data communicated is defined to be the number of bits the sender actually writes into the message. This definition is insensitive to whether the data is communicated by value or by reference because it depends only on *how much* data the sender intends to communicate rather than on *how* this data is communicated. While this definition is independent of semantics, it fails to capture the receiver's participation in the communication.

The third definition looks at the communication from the receiver's viewpoint. Here the amount of data communicated is defined to be the number of bits, u, of data that the receiver actually reads from the message. If the receiver does not touch the messages at all ($u=0$), the throughput is zero. This definition is also independent of message semantics, but it fails to capture the sender's participation in the communication. In the remainder of this section, we will refer to the second and third definitions as the *sender's viewpoint* and the *receiver's viewpoint*, respectively, of the data to be communicated.

Since both viewpoints are equally valid, we define two measures of throughput: T_s, which is determined according to the sender's viewpoint, and T_r, which is determined according to the receiver's viewpoint. Which one of the above measures is actually chosen to represent the throughput of a message mechanism is not critical because in most applications where throughput is important, the receiver typically is programmed to read all the data in each message. In this case, the receiver's viewpoint coincides with the sender's viewpoint, and T_s and T_r are identical. Furthermore, T_s and T_r are formally related by the equation

$$T_r = T_s \frac{u}{b},$$

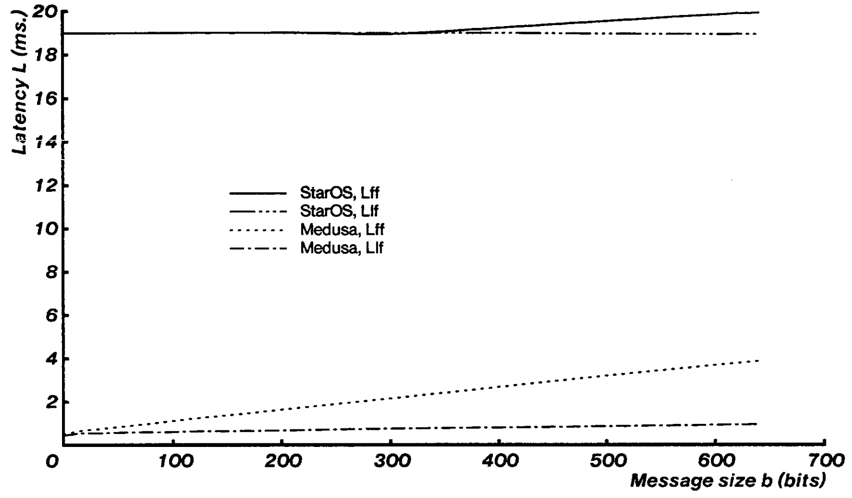
where b is the number of bits written into each message by the sender and u is the number of bits read from each message by the receiver, so that one measure is sufficient to compute the other.

The two throughput measures can be determined experimentally by programming the sender to deliver a large (potentially unbounded) number of messages to the receiver and then measuring the time it takes to send all the messages.

Latency and Throughput Measurements. The StarOS and Medusa message systems were measured under zero-load conditions; that is, the only user processes on the system at the time of measurement were the *Sender* and *Receiver* processes. Thus the measurements represent the best performance that user programs will see under actual operating conditions where there may be other programs contending for resources. All these measurements were taken on versions of Medusa and StarOS that were current on December 30, 1981.

Recall that latency is well defined only if the receiver is waiting. In general, there are two ways in which a receiver can wait: It may be blocked, in which case it is not doing any active work while waiting, or it may be busy-waiting by executing ***Conditional Receive*** operations. The blocked method of waiting usually is preferred

Figure 7-6 Latency of Blocking Operations

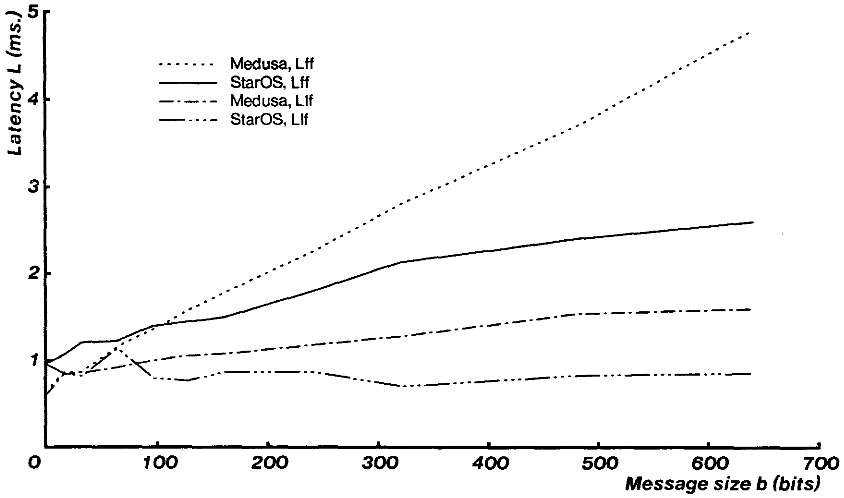


because it does not consume resources needlessly waiting for a message to arrive. For many message mechanisms, however, busy-waiting achieves a lower latency than blocking because it avoids the overhead of process switching. Since the message mechanisms of MEDUSA and STAROS provide both unconditional and conditional message operations, we present latency measurements for both forms of waiting.

Figure 7-6 shows the curves for L_{lf} and L_{ff} for MEDUSA and STAROS in the case where the receiver is blocked waiting for the message to arrive. The latency for each mechanism at a message size of zero indicates how fast the mechanism can communicate a signal from one process to another. The zero-size values of L_{lf} and L_{ff} for MEDUSA are 440 μs. and 500 μs., respectively; for STAROS both are around 19,000 μs. There are two reasons for this large discrepancy. First is the fact that MEDUSA's process blocking and reactivation is performed in microcode; STAROS's is done in software. The second reason is STAROS's voluntary blocking, in which blocking and reactivation have been decoupled from the message mechanism, making it a little more difficult to optimize latency as MEDUSA has done. The variation of L_{lf} and L_{ff} with message size is as expected for both systems. Since MEDUSA's message is value-based, both L_{lf} and L_{ff} increase with message size (although L_{lf} increases more slowly because it does not include setting up the message at the sender). L_{lf} is constant for STAROS, and L_{ff} increases slowly. All the increases are very close to linear.

Figure 7-7 shows the latency curves for the two systems when busy-waiting is

Figure 7-7 Latency of Nonblocking Operations



used. Here, the STAROS results are much better than in the blocking case. The zero-message-size latencies for MEDUSA are L_{lf}=730 μs. and L_{ff}=760 μs.; for STAROS they are L_{lf}=860 μs. and L_{ff}=910 μs. The trends with increasing size are identical to those for the blocking case. In this case, however, there is a crossover point between MEDUSA and STAROS for both types of latency. The value-based mechanism of MEDUSA is faster than the reference-based mechanism of STAROS for sizes below the crossover and slower for sizes above the crossover. For L_{lf} the crossover occurs at around 30 bits (two words) and for L_{ff} at around 100 bits (six words). The variance[1] on all of the latency measurements for the busy-waiting case was much larger than that for the blocking case. This is not very surprising, since the latency obtained in any one measurement depends on the time at which the ***Conditional Receive*** is executed relative to the corresponding ***Send***.

We have identified two measures of throughput, T_s and T_r. Because the two measures are related, we will use only one of the measures, T_s, for comparison. Just as for latency, we will present the measurements of throughput for both the blocking and nonblocking operations.

The first set of measurements (Figure 7-8) are for the case where the receiver

[1] The values plotted for each of the curves are averages over ten sample points for each message size measured. The variance for each point on the MEDUSA curves is around 50μs. and for each point on the STAROS curves around 250μs.

reads all the data communicated by the sender. For both systems, throughput increases rapidly for small message sizes and quickly reaches a saturation value. The performance of blocking and nonblocking cases for MEDUSA is virtually identical for all message sizes, and has a saturation value of around 320K bits/second. For STAROS, nonblocking operations are uniformly better than blocking operations by a small amount, but both are slower than either case for MEDUSA by a factor of around five for small messages and a factor of around two for large messages. The saturation value for the nonblocking case in STAROS is 190K bits/second and for the blocking case is 180K bits/second. The difference relates largely to the fact that message communication in MEDUSA is value-based and in STAROS is reference-based. STAROS must pay a price for the extra synchronization to guarantee the conditions for pure message communication.[2] It also must pay for making nonlocal accesses to the message from either the sender or the receiver.[3] The graph shows that in the case where the receiver references all the data communicated by a sender, a value-based mechanism has a significantly higher throughput than a reference-based mechanism.

The second set of measurements (Figure 7-9) are for the case where the receiver reads only a fraction (10 percent) of the data communicated by the sender. The curves for MEDUSA show the same general trend as before except that the throughput is a little higher and the nonblocking operations are uniformly faster than the blocking ones. For the STAROS curves there is a crossover between the blocking and nonblocking versions at a message size of around 2K bits. The blocking operations are faster for larger sizes and slower for smaller sizes. All the curves reach their saturation values at a message size of 12K bits, although this fact is not apparent from the shape of the STAROS curves. The major difference between this graph and the one for 100 percent usage is that the reference-based mechanism of STAROS performs much better, despite its extra synchronization operations to guarantee pure message communication. This is to be expected because the receiver touches only a fraction of the message and consequently does not pay very much for making expensive nonlocal references. STAROS's mechanism is slower than MEDUSA's, except for blocking operations with messages greater than around 6K bits.

It is interesting to observe that the general shape of all the throughput curves can be described by an equation of the form

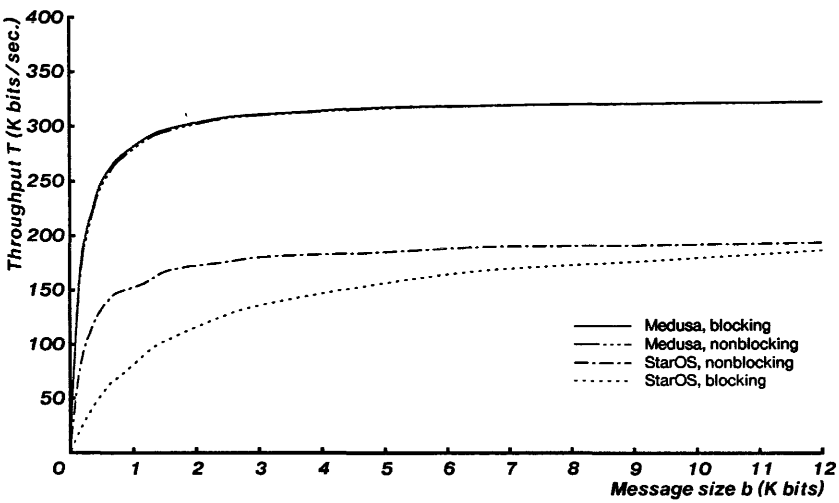
$$T_s = \frac{b}{k_1 + k_2 b},$$

where b is the message size and k_1 and k_2 are constants. Since throughput is

[2] As noted above, however, adding an acknowledgment protocol to a value-based message mechanism would offset some of this penalty.

[3] This assumes that the sender and receiver are located in different Cm's. This will often, but not always, be the case, since a STAROS function frequently invokes another function that runs on the same Cm as the invoker. In such a case, STAROS does not have to pay the overhead of nonlocal accesses.

Figure 7-8 Throughput with 100% Data Usage



defined as the ratio of the total number of bits communicated divided by the total time, the denominator of the above equation is proportional to the total time for communication. The physical interpretation of the above equation is quite straightforward: The total time to communicate *b* bits consists of a constant part and a part that increases linearly with *b*. The constant part (k_1) represents whatever overhead is present in the form of startup cost of the communication mechanism, and the linear part (k_2b) represents the incremental cost per bit for communicating the data.

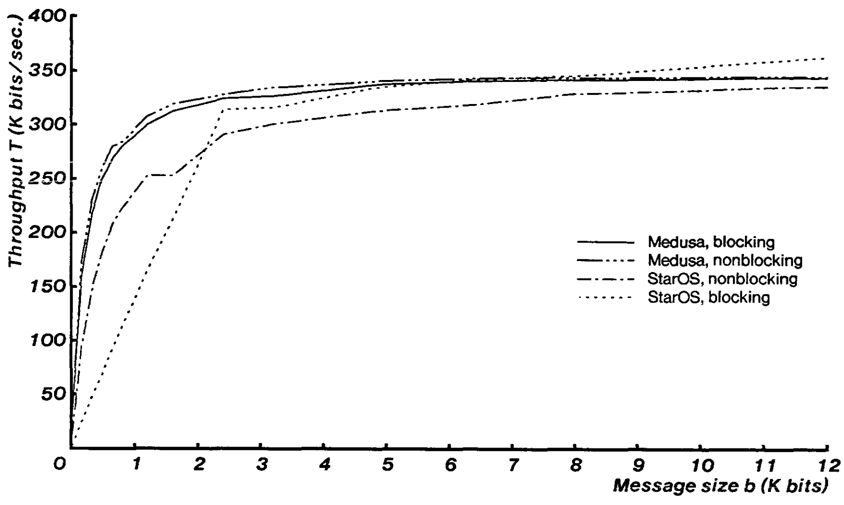
7.5. Summary

We have presented measurements of the two operating-systems implemented on Cm*, STAROS and MEDUSA. The systems are compared with the Smap microcode and against each other. Two specific aspects of their performance have been evaluated: the microcoded kernel and nucleus operations, and the message system. The disparities in functionality and the different implementation strategies are outlined, emphasizing the points where the comparison between the two operating systems is relevant.

One can view the Kmap and its microcode as a special-purpose operating-system processor. As such, the research questions are related to which operating-system functions can profitably be placed in the Kmap and the different ways that each function might be microcoded. The microcode has been measured using two techniques: tracing and end-to-end real-time measurements. The measurements for the three microcodes encompass common operations such as those concerned with addressability and synchronization, as well as MEDUSA's ***Block Transfer*** and

Figure 7-9

Throughput with Low (10%) Data Usage



privileged operations and StarOS's capability instructions and operations on representation types. The metrics that have been used include time, space, number of equivalent LSI-11 instructions, and references to the memory of a Cm.

The second aspect evaluated is the message-system performance. Both StarOS and Medusa rely on messages for requesting operating-system services as well as interprocess synchronization and communication in general. Not only do StarOS and Medusa offer a message-communication facility to user programs, they also make intensive use of it internally. Performance of the message facility is therefore essential to the efficient operation of both systems. The message systems are compared along two dimensions: functionality and performance. Along the functional dimension, similarities (message buffering, conditional and unconditional operations) and differences (value versus reference, message size, automatic versus voluntary blocking) are outlined. Along the performance dimension, two types of evaluations are provided. First, the performance of individual operations are studied. Second, the speed of process-to-process interactions through the message system is compared.

The comparison between the message mechanisms of Medusa and StarOS has centered on two differences: value semantics versus reference semantics, and integration or segregation of process control from the primitives for message communication. The overall results show that as far as latency is concerned, there is a large potential gain from integrating process blocking and unblocking with the primitives for message communication. Most of the improvements come from special optimizations that are possible as a result of this integration. One way in which the value versus reference difference manifested itself is that the latency of a reference-based mechanism is constant, whereas the latency for a value-based mechanism increases with message size. Thus if the zero-message-size latency for a reference-based mechanism is comparable to that for a value-based mechanism, a reference-based mechanism is faster for sending large messages.

In the throughput comparisons, the value versus reference difference was much more important than the integration of process control. In the measurement of throughput, a reference-based mechanism suffers because of two assumptions that frequently, but not always, hold: first, that additional synchronization is needed to make a reference-based system obey the rules for pure message communication, and second, that the communicating processes will be located on different Cm's, requiring either the sender or the receiver to make nonlocal references to the message. Under these assumptions, a reference-based mechanism is slower than a value-based mechanism except when the receiver uses only a small fraction of the data communicated by the sender.

The comparisons presented in this chapter provide a basis for predicting and evaluating the performance of an application on Cm* using one of the operating systems. Before we do so, we will first explore the programming environments and the experimentation methodology available in the Cm* research laboratory.

Acknowledgments. The material in Section 7.2 to "Pure Message Communication" was adapted from the 1980 Cm* Technical Report [Jones and Gehringer 80]. It was written by Pradeep Sindhu, Steve Vegdahl, and Anita Jones. Section 7.3 was adapted from measurements reported in the 1980 Cm* Technical Report [Jones and Gehringer 80]. The measurements were performed by Pradeep Sindhu, Ed Gehringer, and Steve Vegdahl. Section 7.4.3 was adapted from [Sindhu and Singh 83].

III. Programming Environments

8. Languages for Multiprocessing

When writing software for a uniprocessor, the programmer strives for an algorithm that is efficient in both space and time. Beyond that, he or she has little reason to be concerned about resource allocation. Among other services, the compiler automatically allocates variables to registers. The programmer does not need to be aware of how this is done, although some languages do permit the programmer to aid the compiler by supplying hints about which variables must be accessed rapidly (declaring certain variables as "register," for example). The choice of where to place particular program elements in virtual memory is completely irrelevant, since all virtual addresses can be accessed in the same amount of time. The operating system takes care of the mapping between virtual memory and physical memory. Experience has shown that, in general, automatic mapping is much more efficient than manual overlaying. Some operating systems do allow controlled prepaging, since when program behavior is well understood, performance may improve when some swap-in operations are initiated in advance.

In multiprocessing, the situation is far less settled, both because we have less experience and because new issues are raised. MEDUSA and STAROS are both object oriented in the sense that all information is contained within objects. This is especially useful for multiprocessing because it facilitates dividing the address space of a process into portions that are private to it, portions that are shared with other processes, and portions that should be replicated in other processes of the task force. Eventually it may be possible for a compiler to decide which information falls into each class, based on the characteristics of the multiprocessor and how the information is used in the program. Ideally, the programmer should be allowed to override any of the compiler's decisions that seem wrong. For the present, with little experience to go on, both MEDUSA and STAROS give the programmer the full responsibility of partitioning the address space into objects. The facilities for doing so have not been incorporated into existing languages but rather are expressed in a new language that specifies how processes in existing languages communicate with each other.

In hierarchical multiprocessors, resource-allocation decisions are more complicated because access time to memory and peripherals depends on the location of the process doing the access. Clearly, processes should be located close to the memory they will access most frequently, and device managers should be placed near the devices they manage. In addition, if too many processes are assigned to the same processor, performance will suffer. It is not easy for software to balance these conflicting objectives intelligently, especially since the access frequency of a process to a particular data item cannot be known until *after* the process is run. At present, it is best to let the programmer specify constraints on placement to improve resource usage. In the long run, the programmer can be allowed to tune the automatic placement software for better performance.

Interprocess communication and resource usage, then, are the focus of the languages that have been developed specifically for Cm*. There are three such languages: MEDLINK [Scelza *et al.* 81] and TASK [Jones and Schwans 79, Schwans 82], which are used to synthesize task forces in MEDUSA and STAROS, respectively, and AMPL [Dannenberg 81], an experimental language oriented toward the dynamic creation and control of processes. The facilities provided by MEDLINK are generally a subset of those provided by TASK; MEDLINK is best described as a linker enhanced for multiprocessing. AMPL runs on top of MEDUSA but is a complete language; entire sets of communicating processes can be coded in it. The first section of this chapter focuses on TASK and MEDLINK, while the second section presents the AMPL language. The implementation and evaluation of AMPL are described in Chapter 9, which describes additional software environments built on Cm*.

8.1. TASK and MEDLINK

Programs for Cm* are usually written in BLISS [Bliss], which is a systems-programming language developed several years ago without any special features for parallel programming. TASK and MEDLINK describe the interactions between modules programmed in BLISS. In general, a BLISS program codes the algorithm, while TASK or MEDLINK code describes how that algorithm is mapped onto Cm*.

Our description of TASK will refer to a sample program, shown in Figure 8-1, that illustrates most of the features of the language. This program describes a single STAROS task force (Section 6.3.3) that is made up of two modules. The task force solves a system of partial differential equations (PDEs), and is similar to those used for performance studies in Chapters 3, 7, and 11. We will peruse the program in top-down fashion, pausing occasionally to explain how an equivalent MEDLINK program would differ.

Ignoring, for the moment, the mailbox declaration in line 1, the TASK program appears to be divided into three distinct parts. The first, comprising lines 3 to 10, is called a *task-force template* and relates information about the task force as a whole. The remaining two parts are *module templates* and describe the two modules that make up the task force. Each STAROS module exports one or more functions. These functions are described by *function templates* which are nested inside the module templates. The keyword **function** signals the beginning of a function template. The first module defines two functions and therefore contains two function templates. The second module template (module *IOMod*) defines only one function.

In TASK, each module or function template is divided into two sections, which correspond to the two issues mentioned above: partitioning the address space into objects and allocating resources to task forces. The first section is known as the *construction description*. It declares the elements that belong to the module, function, or task force being defined. The second consists of *resource-usage directives*, which describe how objects are to be placed relative to components of the Cm* hardware and to each other. For example, in the *IOMod* module template, the construction description tells us that an *IOProcess* is to be created. It will need to

Figure 8-1 A Program in the TASK Language

```
Mailbox CommBox (MsgType = "Capability", Size = 25);

TaskForce PDEForce is
Construct (
        MastrIO: New CommBox;
        PDEM: New PDEMod(MIOMbox = MastrIO, SIOMbox = SlavIO);
        IOM: New IOMod(MIOMbox = MastrIO, SIOMbox = SlavIO);
        )
Directives ()

Module PDEMod (MIOBox: CommBox, (i=1 . . 15) SIOMbox[i]: CommBox) is
Construct (
        ModCode1: New Basic (Source = ("PDE.obj"));
        ModCode2: New Basic (Source = ("UsrLib.obj"));

        (i=0 . . 9) Grid[i]: New Basic (Size = 4K);

        MasterProcess: Process PDEMod.Master ();   ! The master process
        (j=1 . . 15) SlaveProcess: Process PDEMod.Slave
                                        (MyCommBox = SIOMbox[j]);
        )

        Function Master is
        Construct (
                MProcessStack:  New Basic (Stack);
                MCode1:  Ref ModCode1 (InitialCode ["MasterEntry"]);
                MCode2:  Ref ModCode2 (Window);
                 . . . ;
                )
        Directives()

        Function Slave (MyCommBox: CommBox) is
        Construct (
                SProcessStack:  New Basic (Stack);
                SCode1:  Use ModCode1 (InitialCode ["SlaveEntry"]);
                SCode2:  Use ModCode2 (Window);
                 . . . ;
                )
        Directives ()
Directives (
          NotDiskCm: CmStar[0] where HasDisk = false;
          Same ((i=1 . . 7) SlaveProcess[i], NotDiskCm);
          NearCm ((i=1 . . 7) SlaveProcess[i]);
          NearCm ((i=8 . . 15) SlaveProcess[i], MasterProcess);
        )
Module IOMod (MIOBox: CommBox, SIOMbox: CommBox) is
Construct (
        IOCode: New Basic (Source = ("PDEIO.obj"));
        IOProcess: Process IOMod.IO ();   ! The I/O process
        ! And now, an invocation mailbox for a present process:
        InvokeMB: New Mailbox (MsgType = "Capability");
        )
```

Figure 8-1 (continued)

```
55          Function IO (Present [InvokeMB]) is
            Construct (
                        IProcessStack: New Basic (Stack);
                        ICode: Ref IOCode(InitialCode ["ServiceLoop"]);

60                      TerminalInBuffer: Name; ! Used to receive data from the terminal.

                        TerminalOutBuffer: Name; ! Used to send data to terminal.
                        )

65          Directives (
                        DiskCm: AnyOf CmStar[0] where HasDisk = true;
                        Same (DiskCm, IOProcess);
                        )
      Directives ()
```

transfer large amounts of information to and from a disk, so it should run on a Cm that is directly attached to a disk. The resource-usage directive at line 66 indicates this. In general, resource-usage directives are used to specify that task-force elements that communicate frequently should be placed close to one another and that elements that might interfere with each other should be placed far apart. Future language compilers might be able to make automatic placement decisions, but TASK saddles the programmer with much of the responsibility for efficient allocation.

Templates that are divided into construction and resource-usage sections are called *complex templates*. Table 8-1 gives the BNF syntax for templates, while Table 8-2 defines the syntax for some other basic elements of the language. These tables and those that follow omit certain less important constructs of TASK that are not used in the example program.[1] Nonterminals are denoted by an arbitrary number of characters enclosed in angle brackets—e.g., <x>. The metalanguage constructs used are:

{ <x> | <y> } — either <x> or <y>
{ <x> }$^{\#}$ — an optional <x>
{ <x> }$^{+}$ — one or more occurrences of <x>
{ <x> }* — zero or more occurrences of <x>
{ <x> }$_{;}^{+}$ — one or more occurrences of <x>, separated by semicolons
{ <x> }$_{;}^{*}$ — zero or more occurrences of <x>, separated by semicolons
{ <x> }$_{,}^{+}$ — one or more occurrences of <x>, separated by commas
{ <x> }$_{,}^{*}$ — zero or more occurrences of <x>, separated by commas

A MEDLINK description of the PDE task force would look much like the TASK description. There are only two major differences. First, a MEDUSA task force is made up of activities, not of modules that contain functions from which processes may be

[1] For a complete list see [Schwans 82] or [Gehringer and Chansler 82].

Table 8-1 Syntax for TASK Templates

<Templates> ::= *<Template>** EOF

<Template> ::= {*<Simple Template>* | *<Complex Template>*}

<Simple Template> ::=
<Simple Object Type> *<Simple Template Name>* (*<Actual Attributes>*)$^{\#}$
Some actual attributes may occur only within functions or modules. Hence there are semantic restrictions concerning which attributes may appear in simple templates that are not bound to a particular complex template.

<Simple Object Type> ::= **Basic** | **Stack** | **Mailbox** | **Deque** | **Device**

<Complex Template> ::= *<Task-Force Description>*
| *<Module Description>*

<Module Description> ::= **Module** *<Complex Template Name>*
(*<Formal Parameters>*)$^{\#}$ **is**
<Construction Description>
<Function Description>$^{+}$
<Resource-Usage Directives>
All module attributes must be specified at declaration time. Modules can only be components of task-force templates. Only one instance of a module can be constructed from a particular template.

<Function Description>::=**Function** *<Complex Template Name>*
(*<Formal Parameters>*)$^{\#}$ **is**
<Construction Description>
<Resource-Usage Directives>
A *<Function Description>* can only be a component of a *<Module Description>*.

<Task-Force Description> ::= **TaskForce** *<Complex Template Name>* **is**
<Construction Description>
<Resource-Usage Directives>
A task force template can have no parameters. A *<Task-Force Description>* cannot be a component of another template. No more than one task force template may appear in one TASK program.
Components may have any type except those specifically excluded above.

instantiated. Hence the task force is composed of three kinds of activities: master, slave, and I/O. MEDLINK permits replication factors to accompany activity descriptions. In the PDE example, the slave activity would be replicated 15 times. Normally, objects in the SDL (Section 5.1.1) of the task force are defined before any activity. Objects within the PDL of an activity are defined within the activity definition, yielding this hierarchy:

Table 8-2 Expressions, Names, and Types in Task

<Simple Name>	::=	*<Unquoted String>*
<Simple Template Name>, *<Complex Template Name>*	::=	*<Simple Name>* Template names must be unique in their first five characters.
<Formal Parameter Name>, *<Actual Parameter Name>*, *<Identifier>*	::=	*<Simple Name>*
<Comp Name>	::=	*<Simple Name>*
<Function Name>	::=	*<Complex Template Name>* . *<Complex Template Name>*
<Entry-Point Name>	::=	*<Unquoted String>*
<Object Name>	::=	*<Simple Name>* | *<Access Expr>*
<Var Type>	::=	*String* | *Integer* | *Boolean*
<Expr>	::=	*<Arith Expr>* | *<Quoted String>* | *true* | *false*

- Task-force description
    - Shared-object definitions
    - First activity description
        - Private-object definitions for first activity
    - Second activity description
    - ⋮
    - Last activity description
        - Private-object definitions for last activity

It is also possible for objects to be defined "out of order" to allow two descriptors for the same object to reside in different descriptor lists.

Another minor difference between Medlink and Task is the ability to define constants. This feature is absent in Task, which means that the constant 15 must be repeated several times (or passed as a parameter to each module). Of course, this feature could easily be added to the language.

The second major difference between the two languages is that Medlink specifies the location of an object at the time it is defined rather than grouping location information for all a template's objects together at the end of the template. Medlink places a *location specifier* in the object definition to state where the object is to be placed. Location specifiers have much the same flavor as resource-usage directives in Task. Both will be considered later.

8.1.1. Objects and Their Attributes

Objects such as basic objects, deques, and mailboxes are described using Task *simple templates*. A simple template itself is analogous to a *type* definition in other programming languages. In the example program,

```
Mailbox CommBox (MsgType = "Capability", Size = 25);
```

is an example of a simple template. It declares a *CommBox* "type," a capability mailbox with space for up to 25 messages, from which *instances* of objects are later created in

```
MastrIO: New CommBox;
(i=1 . . 15) SlavIO[i]: New CommBox;
```

(lines 5 and 6). We will consider the *iteration* construct in the second declaration later. If we had intended to define capability mailboxes of different sizes, we could have left the size unspecified until the actual instantiation directive:

```
Mailbox CommBox (MsgType = "Capability");
                 :
New CommBox (Size = 25);
                 :
New CommBox (Size = 10);
```

It also is possible to define instances of objects by listing their attributes (Table 8-3) explicitly rather than by referencing a template:

```
ModCode1: New Basic (Source = ("PDE.obj"));
```

Table 8-3 Syntax for TASK Parameters and Attributes

```
<Formal Parameters>  ::=  <Formal Parameter>* | <Actual Attribute>*
<Formal Parameter> ::= {<Formal Parameter Name>,+ | <Iteration>} :
                                                   <Simple Template>,
                       | <Identifier>,+ : <Var Type>,

<Actual Attributes>  ::= <Actual Attribute> | <Actual Attribute>, <Actual Attributes>
<Actual Attribute>   ::= Source = ({<Filename>},+)
                       | Size = <Integer>
                       | <Special Attr>

<Actual Parameters>  ::= {<Actual Parameter> | <Actual Attribute>}*
<Actual Parameter>   ::= <Key Expr> = <Actual Expr>,

<Actual Expr>        ::= <Object Name> | <Iteration> | <Expr>
                        | <Var Expr> | <Access Expr> | <Index>

<Key Expr>           ::= <Formal Parameter Name> | <Access Expr> | <Iteration>

<Keyword Name>       ::= <Obj Att> | <Hard Att>

<Obj Att>            ::= Source | MsgType
```

This line instantiates a basic object that is to be initialized from the file *PDE.obj*. The word **New** indicates that an object is to be created anew. Another way to name an object is to reference an existing object, as in line 28:

MCode2: **Ref** *ModCode2* (**Window**);

This causes a capability for (i.e., a reference to) the object to be placed in a capability list of a task force, module, or process. (The **Window** attribute will be explained later.) A third option is to copy an existing object; the **Use** construct does this, as in line 37:

SCode2: **Use** *ModCode2* (**Window**);

This places a separate copy of the code in *ModCode2* in the address space of each process that is instantiated from the *Slave* function template.

All three of these alternatives cause a capability slot in a module object or process object to be filled with a capability for the object that is created or referenced. The fourth and final possibility is that the TASK program needs only to reserve a capability slot into which a capability for an object can be stored at some later time. The **Name** construct is used for this purpose. *TerminalInBuffer* and *TerminalOutBuffer* in the sample program are declared as **Name**s because no objects fill their slots when the task force is loaded. Rather, when the *IOProcess* begins to run, it will look up the terminal input and output buffers in a standard library and copy capabilities for them into the slots that have been reserved. The **Name** construct is needed only when two or more separately compiled BLISS modules need to reference the same capability slot. Otherwise, the names *TerminalInBuffer* and *TerminalOutBuffer* can be defined in the BLISS program that uses them; they do not have to appear in the TASK program at all.

When a complex template is instantiated, its component objects are also instantiated, either by manufacturing new objects or by acquiring the appropriate capabilities. For example, when a new *Master* function template is created in our example, new *MProcessStack* and *MCode* objects are created, too. Just as the **New** construct indicates that an object is to be fabricated, the **Process** construct indicates that a new process is to be created to execute a specific function of a specific module. (The process does not begin execution until it is scheduled, however.) The notation *PDEMod.Master*, for example, names the *Master* function of the *PDEMod* module. The syntax for construction descriptions can be found in Table 8-4.

Like simple templates, complex templates may also be parameterized. Both module templates in our example are parameterized. The parameters are mailboxes used for intermodule communication. The parameterization allows the modules to share information via the mailboxes. In TASK, the scope of object names encompasses only the template in which they are defined, so if the modules had not been parameterized, the *MasterIO* and *SlavIO* mailboxes declared in the task-force template could not be referenced from outside it.

Table 8-4 Syntax for Construction Descriptions

```
<Construction Description>  ::= Construct ( <Component> ;*)

<Component>      ::=           <Comp Name> : <Operation>
                 | <Iteration> : <Operation>

<Operation>      ::=  New { {<Object Type> { (<Actual Parameters>) }#}
                               | <Template Name> { (<Actual Parameters>) }#)}
                 | Ref <Object Name> { ( { <Special Attributes> } ) }#
                 | Use <Object Name> { (<Actual Attributes>) }#
                 | Process <Function Name> (<Actual Parameters>)
                 Note: returns a mailbox.  Semantics: Processes may be created
                 only within the body of the module defining their function
                 | Name { (<Special Attributes>) }#
                 Only the Window attribute should appear here.
```

In a TASK program, all template names are global and must be unique in their first five characters. Function names are nested, as in Algol or Pascal. If a particular name occurs twice, an instance of the name refers to the one in the smallest enclosing complex template: complex-basic, module, or task-force description. As noted above, object names are known only within the complex template in which they are defined.

8.1.2. Iteration

In the complex templates of the previous section, each component is explicitly named, but it is often useful for the number of components to vary with the size of some data structure or the number of processors used. TASK's *iteration* construct is used to specify multiple components that differ only in minor ways (Table 8-5). Iteration can be used to express the *partitioning* of data objects into multiple objects and to specify parameterized *replication* of objects.

Our example program uses replication to create a separate mailbox for the I/O process to communicate with each of the slaves (line 6), to create ten basic objects to hold the grid of coefficients for the PDE (line 17), and to tell how many slaves are to be created (line 20). In many cases, it is helpful to parameterize a module definition in order to vary the number of server processes:

```
Module PDEMod( ... NumSlaves: integer) is
Construct (
      ...
   (j=1 .. NumSlaves Max 12) SlaveProcess: Process PDEMod.Slave
                                                  (MyCommBox = SIOMbox[j]);
   )
```

The iterative syntax above will cause the creation of *NumSlaves* processes, each executing the *Slave* function. *NumSlaves* potentially varies each time the task force is instantiated but may not exceed the value 12.

Table 8-5 Syntax for Iterations

```
<Iteration>   ::=   (<IterName> = <Low Limit> .. <High Limit>) <Access Expr>

<Access Expr>   ::= <Iterated Name> [<Index>]

<Index>       ::=   <IterName> {Mod <Var Expr>}#
<Low Limit>   ::=   <Integer>
<High Limit>  ::=   <Integer>
                  | * Max <Integer>
                  | <Integer> Max <Integer>
                  | <Variable>

<IterName>    ::=   <Simple Name>
<Iterated Name>   ::= <Comp Name>
                  | <Formal Parameter Name>
                  | <Actual Parameter Name>
```

In the example program, the grid was partitioned into ten objects to circumvent the maximum basic-object size of 4,096 bytes (Section 4.2). In this case, we knew how large the grid was before running the program. This may not always be true, especially when a program takes a data structure as input or in the case of object code, where it is difficult to predict how large the compiled code will be. If the task-force author does not know how many objects will result from partitioning a source file, an asterisk (*) is used to indicate "as many as necessary to exhaust the file." Here is an example:

(*i*=0 . . * **Max** 5) *MyCode*[i]: **New Basic** (**Size** = **Source** = ("*Codfil.obj*"));

The first 4K bytes of the file will be placed in *MyCode*[0], the next 4K bytes in *MyCode*[1], and so forth until the input is exhausted. The maximum range of the iteration index must be known at compile time. Thus no more than six *MyCode* objects will be created, regardless of the size of the file *Codfil.obj*. Furthermore, the last *MyCode* object will be 4K bytes in length, regardless of whether it is completely filled.

8.1.3. Special Attributes

Several times during any TASK program it is necessary to refer to specific characteristics of the STAROS system. These characteristics are known in TASK as *special attributes* (Table 8-6). **InitialCode** (lines 27, 36, and 58 in the example program) refers to an object and means that execution of a STAROS process begins in the object at the named entry point (e.g., *MasterEntry*). The **Stack** attribute, which also occurs in the example program, identifies an object as containing the process stack of a process. TASK attempts to place **InitialCode** and **Stack** objects in the same Cm as the process that uses them. If multiple processes are created from the same

Table 8-6 The Syntax of Special Attributes

```
<Special Attr>  ::= Window [<Integer>]#
                  | PrivateMailbox
                  | InitialCode [ " <Entry-Point Name> " ]#
                  | Stack
                  | StackOwns [<Object Name>]
                  | Present [<Object Name>]
                  | Alias [ " <Function Name> " ]
```

function, it is mandatory to create separate **Stack** objects (note the **New** construct in the object definition) and usually best to create separate code objects for each (via the **Use** construct), although two processes may profitably share a code object (through the **Ref** construct) if they execute on the same Cm.

As noted in Section 6.3.1, a StarOS function may either be *present*, meaning that at least one process permanently exists to carry out the function, or be *absent*, indicating that a new process is created each time the function is invoked and terminates after servicing the invocation. If a function is present, all requests for invoking it are sent to its *invocation mailbox*. In the example program, the I/O function is present (since it would be wasteful to create a new process each time input or output must be performed). The construct **Present** *InvokeMB* in its definition states that *InvokeMB* is its invocation mailbox.

Unlike invocation mailboxes, which are shared by multiple processes executing the same function, each process that executes an absent function is invoked via its own *private mailbox*. Task allows a mailbox to be declared "private," as in

MyOwnMailbox: **New Mailbox (MsgType** = "*Capability*", **PrivateMailbox)**

which says that the private mailbox of a process created from the function that is currently being defined is to be called *MyOwnMailbox*. Since each new process receives a unique private mailbox, however, and since the **Process** construct returns whatever object is the private mailbox of the process, the **PrivateMailbox** attribute rarely needs to be used.

Since objects may be addressed only through 15 windows, a maximum of 15 objects may be addressable when a process begins execution. Among them must be the **InitialCode** and **ProcessStack** objects (or else the process cannot successfully execute its first instruction); Task takes care of placing these in windows automatically. In principle, the allocation of other windows is analogous to the allocation of registers and could be handled automatically by a language compiler. In practice, the Bliss compiler was not modified to do this, so StarOS (and Medusa) programs must explicitly load (a capability for) an object into a window before referencing it.

A Bliss program can cause a window to be loaded by executing a ***Load Window*** instruction, but certain objects, such as additional code objects or mailboxes used for interprocess communication, are likely to be used soon after a program begins execution. From the standpoint of readability, it is convenient to declare their need to be in windows in the same place that the objects themselves are instantiated. TASK provides the **Window** attribute for this purpose. In the example program, the *MCode2* (line 28) and *SCode2* (line 37) objects were marked as **Window**. Had they lacked this attribute, the *MasterEntry* or *SlaveEntry* routines would have had to load those objects into windows before any code from them was executed.

8.1.4. Resource-Usage Directives

As noted at the beginning of this chapter, resource allocation for parallel programs is still an open research question. The ideal would be to assign processes and data to processors automatically, or at least semiautomatically, allowing for tuning by the programmer. For the moment, TASK and MEDLINK give the programmer tools for making a rather complete specification of how resources should be allocated. In a task force such as the PDE, several resource-usage constraints come to mind. All processes should execute local code and use a local process stack (for justification, see Section 3.1.4)—this is one decision that TASK makes automatically. Processes that are ready to execute simultaneously should be assigned to separate processors. Page objects in MEDUSA and 4K-byte basic objects in STAROS that are used by only one process should be local to that process; other private objects should be in the same cluster. Processes that must use a certain peripheral should run on the processor that is attached to that peripheral. In the PDE task force, for example, the I/O process needed to execute on a processor that contained a disk. The TASK statements

```
DiskCm: AnyOf CmStar[0] where HasDisk = true;
Same (DiskCm, IOProcess);
```

at lines 65 and 66 in the PDE example arranged this by declaring *DiskCm* to be a processor that has a disk attached. In the next line, the *IOProcess* was constrained to run on this processor. The resource-usage section at lines 42 through 45 causes slave processes 1 to 7 to execute in one cluster on Cm's that *do not* have a disk attached (ensuring that the *IOProcess* has a Cm all to itself) and constrains the rest of the slaves to execute in the same cluster as the master. This must be a second cluster, since one cluster can contain a maximum of 14 Cm's.

In TASK, the resource-usage section of a complex template (Table 8-7) consists of *selections* and *directives*. Selections define terms that are used in the directives by associating a set of hardware resources with an identifier. Examples are the definitions of *DiskCm* and *NotDiskCm* in the sample program. Directives come in two types. The first type, called *proximity relations*, specifies constraints on the relative location of two or more software components. The second type is used to constrain the placement of software objects to hardware resources with specific attributes. Since most directives use *relative* rather than absolute locations, the TASK compiler can consider various mappings of software to hardware, in an attempt to achieve more optimal resource utilization by minimizing total communication across buses.

Table 8-7 Syntax for Resource-Usage Directives

```
<Resource-Usage Directives> ::= Directives (<Selection>;*<Directive>;*)

<Selection> ::= {<Hardware Set Name> | <Identifier>} : <Selection Expr>
<Directive> ::= <Iteration># <Proximity Degree>
                              ({<Iteration># <Object Name>)},+)
                | <Iteration># Same (<Hardware Set Name> [<Index>]#,
                              {<Iteration># <Object Name> | <Identifier>},+)

<Proximity Degree>  ::= <Integer> | SameCm | SameCluster | NearCm
                      | DifferentCm | DifferentCluster | NoCare
<Selection Expr>    ::= <Set Expr> {where <Attr Expr>}#
                      | AnyOf <Set Expr>
                      | NumberOf <Set Expr>

<Set Expr>  ::=  <Set Name> | (<Set Name> <Opr> <Set Name>)
<Set Name>  ::=  <Predefined Set Name> | <Hardware Set Name>
<Attr Expr> ::=  <Hard Att> <RelOpr>
             |   (<Hard Att><RelOpr><Hard Att Value><Opr><Hard Att>
                 <RelOpr><Hard Att Value>)#

<Opr>       ::=  and | or
<RelOpr>    ::=  < | > | = | <= | >=

<Hard Att Value> ::= <Integer> | true | false
<Hard Att>  ::=  MaxMPSize | ActualMPSize
             |   NumEther | NumLines | NumDisks
             |   MaxSize | ActualSize
             |   HasDisk | HasLine | HasEther
             |   NumCms | NumClusters

<Predefined Set Name> ::= Cm | CmStar
                        | Cluster0 | Cluster1 | ... | Cluster4
                        | Cm[0,0] | Cm[0,1] | ... | Cm[0,14]
                        | Cm[1,0] | Cm[1,1] | ... | Cm[1,14]
                        | Cm[2,0] | Cm[2,1] | ... | Cm[2,14]
                        | Cm[3,0] | Cm[3,1] | ... | Cm[3,14]
                        | Cm[4,0] | Cm[4,1] | ... | Cm[4,14]

<Hardware Set Name> ::= <Simple Name>
```

Proximity Relations. Because Cm* has a three-level hierarchical structure, there are three meaningful degrees of proximity between objects: They may be within the same Cm, in different Cm's but within the same cluster, or in different clusters. Degrees of proximity such as these can appear in resource-usage directives. If TASK were designed with a different target architecture in mind, say Cube-Connected Cycles [Preparata and Vuillemin 81] or Alpha structure [Bhuyan and Agrawal 84], different measures of proximity would be used. For these architectures, a suitable measure would be the number of nodes that a message traverses on its way to the destination. For a set of processors communicating over an Ethernet, there are only two degrees of proximity—local and nonlocal—as the transmission time over the network is independent of the relative location of the nodes.

Taking into account the three levels of hierarchy, and the need to keep different objects together or apart, six different proximity constraints are possible:

SameCm—Should be in the same computer module.
SameCluster—Should be in the same cluster and can be in the same computer module.
NearCm—Should be in different computer modules and should be in the same cluster.
DifferentCm—Should be in different computer modules and can be in different clusters.
DifferentCluster—Should be in different clusters.
NoCare—Can be anywhere.

Except for objects with the **InitialCode** or **Stack** attributes, the **NoCare** proximity degree is given to all software components that are not related by explicit directives.

Proximity relations are specified by naming the relation and listing the objects to which it applies. For example, we could have explicitly given the default directive for the location of the master process's stack:

SameCm (*MasterProcess*, *MProcessStack*);

Iteration can be used, as it was in the **NearCm** directives that constrained sets of slave processes to the same cluster.

The **SameCm** proximity relation implies a new "same-processor" relation that is transitive.[2] For example:

SameCm (*MyStack*, *Scratch*);
SameCm (*Scratch*, *MyCode*);

requires that *MyStack* and *Scratch*, *Scratch* and *MyCode*, and consequently *MyStack* and *MyCode* are placed in the *same* computer module's memory. Such implied relations can induce conflicts among proximity relations. A conflict occurs, for instance, if we add the directive

NearCm (*MyStack*, *MyCode*);

to the directives in the example above. The conflict is resolved by giving *same* preference over *different*. In the example, *MyStack* and *MyCode* would remain **SameCm**, and in the same processor. To avoid conflicts of the form

[2] The next several examples have been taken directly from the StarOS Manual [Gehringer and Chansler 82], this part of which was based on an earlier technical report by Jones and Schwans [Jones and Schwans 80]. They also appeared in [Schwans 82].

SameCm (*Code*, *Cm1*);
SameCm (*Code*, *Cm2*);

each component of a template (e.g., *Code*) can be named in exactly one **SameCm** or **SameCluster** directive.

The **NearCm** directive implies a "same cluster" relation that is also transitive. For instance, the two directives

NearCm ((*i*=1 . . 9) *ServerPM*[*i*]);
NearCm (*Scratch*, *ServerPM*[1]);

require that each server process must be **NearCm** to all other server processes and also that the object called *Scratch* be **NearCm** to the first server process. All servers and the object *Scratch* will be placed in the *same cluster*.

As noted above, the TASK compiler attempts to resolve conflicts in resource-usage directives by giving **Same** preference over **Near**. TASK also has a facility for giving the programmer a *finer grain* of control over placement decisions by using integer-valued proximity degrees. The proximity degrees run from 1 to 100, with higher values expressing a greater need for components to be near one another. All six proximity constraints are assigned values within this range:

SameCm corresponds to 90.
SameCluster corresponds to the range 31–89.
NearCm corresponds to 30.
DifferentCm corresponds to the range 11–29.
DifferentCluster corresponds to 10.
NoCare corresponds to the range 1–9.

Values above 90 express a stronger need to locate objects together than even the *SameCm* directive implies. Values within a range are used for resolving conflicts. A value of 88 would be given preference over a value of, say, 31 when two directives conflict.[3] For a final measure of control, a compile-time switch allows the programmer to specify whether the resource-usage directives are to be interpreted as *preferences* or *constraints*. If they are preferences, the compiler resolves conflicts to the best of its ability according to the priority rules just described. If they are constraints, then conflicting directives such as the **NearCm**(*MyStack*, *MyCode*) directive given above are flagged as illegal.

Naming Specific Hardware Resources. Cm's and clusters are not all created equal. Some Cm's have 64K of memory; others have 128K. Some have Ethernet boards, while others are attached to disks. Several processors have serial-line units

[3] For the purpose of resolving conflicts, the *SameCluster* and *DifferentCm* proximities are mapped to some integer value within their range. This value is empirically determined based on performance studies and is subject to change.

for connection to terminals or multiplexers. Clusters contain different numbers of these resources, even when all processors and devices are up simultaneously. One should be able to load task forces regardless of the availability of specific resource units, as long as sufficient resources are available.

The I/O process in our example program must run on a Cm attached to a disk. The programmer should not have to name a specific Cm because that demands knowledge of the Cm* configuration (which can change) and requires recompilation in the event that either the Cm or the disk fails. TASK allows *virtual* hardware resources to be named for those occasions when no other method is sufficiently expressive for the programmer's wishes. It also provides two other ways to select resources: selection from a *resource set* and selection by *hardware attributes*. We will consider all three methods of resource selection.

SELECTION BY (VIRTUAL) HARDWARE COMPONENT NAME. TASK uses Cm*'s configuration table, which can be updated before each run, to determine what hardware resources are available. It calls the first cluster it encounters **Cluster0**, and the first Cm within this cluster **Cm**[0,1]. (Alternate ways of referring to these resources are **CmStar**[0] and **Cluster0**[1].) It thus defines *virtual* cluster 0, which may or may not be physical cluster 1 of Cm*. Subsequent Cm's and clusters are numbered sequentially, as shown in Table 8-7, up to a maximum of **Cluster4** and **Cm**[4,14]. A programmer who wants to place, say, the *MasterProcess* on **Cm**[1,2] codes

Same (*MasterProcess*, **Cm**[1,2])

SELECTION FROM A RESOURCE SET. Usually it is not necessary for a program to name a particular hardware resource, such as a Cm, because any element of a set of similar resources will do. The example program, for instance, must have processes placed in particular clusters but not on specific Cm's within those clusters. The **AnyOf** construct can be used in a selection expression to choose an arbitrary element of a set. If we had been interested only in placing the *MasterProcess* on any Cm within **Cluster1**, we could have written

MyCm: **AnyOf** (**Cluster1**);
Same (*MasterProcess*, *MyCm*);

by first defining the term *MyCm* and then using it to specify the placement of *MasterProcess*.

The predefined names **CmStar** and **Cm** can also be used as resource sets. **AnyOf CmStar** names one arbitrary cluster in the system, while **AnyOf Cm** names an arbitrary Cm somewhere in the system. The **NumberOf** construct returns the cardinality of a resource set (e.g., "**NumberOf Cm**[1]" returns the number of functioning Cm's in **Cluster1**). This value may be useful as an upper bound for an iteration.

SELECTION BY HARDWARE ATTRIBUTE. The characteristics of Cm's and clusters mentioned at the beginning of this section—amount of memory, peripherals, and so forth—are called *hardware attributes* by TASK. Some attributes of individual Cm's are

MaxMpSize, the amount of memory in a Cm.
ActualMpSize, the amount of memory available to user programs (excluding that occupied by the operating system).
HasDisk, **HasLine**, and **HasEther**, which tell whether the Cm is connected to a disk, a serial-line unit, or an Ethernet board, respectively.

The corresponding attributes of a cluster are **MaxSize**, **ActualSize**, **NumDisks**, **NumLines**, and **NumEther**, obtained by summing the values of the respective attribute in each Cm.

The **where** construct expresses selection by attribute. For example,

```
AnyOf CmStar[1] where (ActualMpSize > 48K and HasEther = true)
```

selects an arbitrary Cm in virtual cluster 1 that has an Ethernet board and more than 48K of memory available for user programs.

Observe how selection by attribute is used in the example program. At line 66, *DiskCm* is defined to be a particular Cm with a disk attached. The next line constrains *IOProcess* to execute on it. The definition of *NotDiskCm* at line 42 is a bit more interesting. *NotDiskCm* is a set of resources consisting of all the Cm's in **Cluster0** that do not have disks. Slave processes 1 through 7 are constrained to execute on members of this set. Line 44 further refines the requirements to mandate that the processes execute on *different* members of this set.

The fact that the example program must use the virtual hardware component **CmStar**[0] in lines 42 and 66 points out a limitation of TASK. It would have been more elegant to define

```
PDECluster1: AnyOf CmStar where (NumDisks > 0 and NumCms > 7);
PDECluster2: AnyOf CmStar where NumCms >= 7;
DiskCm: AnyOf PDECluster1 where HasDisk = true;
NotDiskCm: PDECluster1 where HasDisk = false;
```

thus freeing the *IOProcess* from the need to run in **Cluster0** (and allowing the task force to be loaded even if there is no disk on **Cluster0**). Unfortunately, TASK's scope rules defeat us; there is no way to give two module templates access to this definition. The provision of globally defined names would resolve the problem, although it is fraught with dangers of its own. MEDLINK provides global names and also allows names defined in one activity (analogous to a template) to be selectively exported to other activities. Its **group** command is used to assign an activity to a named activity group; all activities in the group share a common name space.

MEDLINK's Location Specifiers. As we have seen, TASK templates come in two sections: After the construction description instantiates a set of objects, the resource-usage section specifies their placement relative to each other and relative to hardware components. Medlink associates a location specifier with an object —potentially with each object, although **Don't Care** is the default if no location specifier appears.

A Medlink location specifier is made up of two parts, a reference part and a relative-location part. The reference part serves as a reference point for the relative-location part, which is similar to a Task proximity constraint. The reference part can name a specific (physical, not virtual) Cm, an activity, or a task force. A Medlink location specifier might read

Cm1-5, Prefer Same Cm

Except for **Don't Care**, relative-location names are three-word phrases. The second word always is "same," while the first may be "require" or "prefer," and the third is "cluster" or "Cm." Relative-location names may be concatenated where meaningful, providing for six different relative-location parts:

Don't Care
Require Same Cm
Require Same Cluster
Prefer Same Cm
Prefer Same Cluster
Prefer Same Cm Require Same Cluster

Contrasted with Task, which treats directives as preferences or constraints on an all-or-nothing basis, Medlink provides more flexible control (although it is not clear how often the programmer might wish to mix preferences and requirements in the same program). However, Medlink lacks the **Different** proximity constraints available in Task. The programmer must name physical Cm's in order to keep task-force components apart, greatly limiting the amount of optimization that Medlink can perform. By contrast, we have seen that Task obliges a programmer to name virtual hardware resources to control the relative placement of components of different modules.

8.1.5. Running a Task Program

When a task force is instantiated using the *PDEForce* template, a StarOS task-force object is created. All its components, including the two module objects, are created and initialized. Instantiation of the module templates results in the creation of all objects specified in the modules' construction descriptions, thereby creating all 17 processes. This "chain-reaction" instantiation of objects simplifies the construction of a task force for the programmer by taking away the need to code the sequence of individual actions involved in building the task force.

It is the StarOS *Loader* that creates the task-force object. In this object, the *Loader* places a capability for each object instantiated within the task-force template. After the *Loader* finishes creating the task-force object, and anything else that it causes via the chain reaction, it gives the module capabilities in the task-force object to the user interface. This enables the user to invoke functions of these modules from command level.

8.1.6. Retrospectives

TASK and MEDLINK represent the first attempts at a linguistic approach to the problems of resource allocation in a running multiprocessor system. TASK was more ambitious than MEDLINK, but neither was considered easy to use. As noted earlier, both had a tendency to force the user to name particular Cm's in resource-usage directives. Both required the user to be familiar with the address-mapping mechanism and the organization of object-code files, as well as the processor interconnection structure.

It is not difficult to propose solutions to these problems. A judicious selection of scope rules, coupled with a generalization of the **Near** and **Different** proximity constraints of TASK, should go a long way toward overcoming the need to name physical hardware components. Perhaps some form of nested module structure can aid this effort. Beyond that, the Cm* project suffered from its attempt to make do with minimal modifications to existing compilers, as TASK and BLISS programmers constantly were confronted with the need to make objects addressable. In the future, larger address spaces will make allocation of "window registers" a matter of diminishing concern. The problem will not disappear entirely, however, because short-address operand formats will still be needed. For some systems, an associative cache may take over the work of mapping short addresses to full virtual addresses, but for other systems, software will still play a role. Window registers, like general registers, can best be managed by the compiler, however, perhaps aided by some programmer-supplied hints. At any rate, the programmer should not be bothered with routine decisions about allocating addressing registers.

Another job that neither TASK nor MEDLINK handled automatically was the assignment of object code to objects. Given the fact that objects could hold no more than 4K bytes of data, and that only 15 of them could be simultaneously addressable, it was frequently necessary to make careful choices about which object code was to be placed in which objects. TASK and MEDLINK provided ways of controlling the placement of individual CSECTs, and a bin-packing program was written to assist the programmer, but the program was not integrated into either TASK or MEDLINK. As a result, the programmer had to use the output of the bin-packing program as a guide in writing low-level directives to TASK and MEDLINK—in effect, performing manual overlaying. This is another problem that will decline in importance as address spaces grow larger. To the extent it remains, it can be combated by integrating bin-packing algorithms into the compiler.

Finally, there is the question of whether separate languages such as TASK should exist at all. Any programming language can include facilities for process and object creation (AMPL, for example, allows dynamic process creation) and could be extended to encompass resource-allocation decisions, too. This would free the programmer from the need to know *two* languages just to write a simple program, but it also would force every language to incorporate these facilities, thus increasing the complexity of languages and compilers. From the standpoint of compiling, at least, improved technology should help cope with the complexity. To avoid complicating the source code, perhaps resource-allocation directives can be separated from the body of the code in much the same way that module specifications are segregated in languages such as Ada.

8.2. AMPL

AMPL (A MultiProcessing Language) is an experimental high-level language designed by Roger Dannenberg for expressing parallel algorithms that involve many interdependent and cooperating tasks. AMPL is a strongly typed language in which all interprocess communication takes place via message passing. Like TASK and MEDLINK, AMPL is implemented on Cm*. Unlike TASK and MEDLINK, complete programs can be written in AMPL; in fact, a number of programs have been written to perform numeric and symbolic computation. The AMPL implementation includes its own run-time system, which runs on top of MEDUSA and provides interesting contrasts to STAROS's garbage collection and MEDUSA and STAROS's message systems. This section concentrates on the language itself; Section 9.2 explores the design and performance of the language and its run-time system.

Motivation for the design of AMPL originated in the consideration of dataflow languages and experience with hardware design. Since the inception of digital electronics, hardware designers have devised interpreters, real-time control systems, and data-processing systems that utilize thousands of computing elements, all operating in parallel.

Dataflow computers are an approach to achieving this level of parallelism in a programmable machine. Dataflow programs are normally restricted in that only functional or "stateless" programs are allowed. The value of a "variable" cannot be changed because values are used to synchronize computation. In AMPL, networks of computing elements can be interconnected to perform computation like the nodes of dataflow machines, but like hardware elements, nodes can have state. Not only does this allow operations with side effects, but storage can be distributed and maintained at the site of computation rather than in a "structure processor" [Ackerman 78] or other special storage unit, as required by a dataflow machine.

AMPL is not designed for the implementation of systems that must survive hardware and software errors to provide reliable service over extended periods of time. As in early algorithmic languages for uniprocessors, little attention is paid to issues of exception handling and error recovery. AMPL is machine independent and could be implemented on a variety of multiprocessor and computer network structures. Because AMPL is an experimental language, it has intentionally been kept simple and small. A richer syntax and additional features would be expected of a production language.

8.2.1. AMPL Programs and Modules

AMPL is based on the use of message passing for both communication and synchronization. Shared memory is not available. The language provides for the dynamic creation of processes, the ability to pass references to processes through messages, and garbage collection of processes. AMPL is a strongly typed language, and borrows heavily from Modula and Pascal, using a similar syntax and also restricting the use of dynamic structures to simplify storage allocation [Wirth 77]. Its parallel processing facilities are descendants of CSP [Hoare 78], although substantial changes have been made, for example, to allow dynamic process creation.

A short example of an AMPL program is given in Figure 8-2. The program adds ten pairs of numbers and prints the results. Two processes are used in addition to an output process predefined by the language. One process generates numbers to be added and formats the output. The other process actually performs the additions, returning results to the formatting process. Figure 8-3 illustrates the run-time structure and communication paths of this program.

As can be seen from Table 8-8, an AMPL program is made up of a series of module definitions, potentially preceded by global constant and type definitions. The adder program consists of two module definitions, one global type, *RefPortInteger* (its mnemonic significance will be explained later), and no global constants. If we wished, however, we could easily use a global constant to control how many numbers are added by inserting the declaration

```
const NumAddends = 10;
```

and changing the 10 in line 12 to *NumAddends*.

A module header includes the module name, which has global scope, and a parameter list. The module name is passed as an argument to the *create* function, and causes a new process to be instantiated to execute the module's code. The variables in the parameter list are initialized to the values supplied in the *create* function call, as described below. All processes must be explicitly created in this way at run time with two exceptions: (1) system-defined modules for I/O are created automatically, and (2) an instance of the program-defined module *main* is created automatically.

8.2.2. Basic Program Elements

Within a module definition, one can define constants, types, variables, and ports (analogous to pipes in MEDUSA or mailboxes in STAROS) associated with the module. Constants and types declared within a module are local to that module; if declared outside, they have global scope. No variables can be declared outside modules because AMPL does not allow shared memory. Special rules apply to port declarations, as will be explained in Section 8.2.3. Table 8-9 gives syntax for these and other basic program elements. Note the strong resemblance to the corresponding elements of Pascal. The similarity carries over to comments, which may be bracketed by { and } or (* and *).

Figure 8-2 A Simple AMPL Program to Add Ten Pairs of Numbers

```
type RefPortInteger = refport integer;

module main;                                    {this process created automatically}
port
     SumPort: integer;
var
     sum, i: integer;
     adder: refmod AdderMod;
begin {add some numbers using an instance of AdderMod}
     adder := create (AdderMod, self.SumPort);       {parameter tells . . .}
     i := 1;                                         { . . . where to send results}
     while i < 10 do
               send i to adder.APort;
               send 100 to adder.BPort;
               send i to WrInt;                 {print i}
               send ' + 100 = ' to WrStr;       {print a string}
               accept SumPort(sum);             {get sum of i+100}
               send sum to WrInt;               {print sum}
               send 1 to WrLn;                  {print 1 newline}
               i := i + 1
               end;
     end;


{the following module adds two integers and sends the sum to result:}

module AdderMod (result: RefPortInteger);
port
     APort: integer(1);                         {operands arrive one at a time}
     BPort: integer(1);                         { so buffer size is set to 1}
var
     a, b: integer;
begin
     while true do
               accept APort(a);                 {get operands . . . }
               accept BPort(b);
               send a + b to result;            { . . . return result}
               end
     end
```

AMPL uses a standard expression syntax (Table 8-10). The operators, listed in order of decreasing precedence (with operators of equal precedence on the same line) are as follows:

```
  - (unary minus)
  * / div mod
      + -
< <= >= = > <>
     not
     and
      or
```

Figure 8-3 Run-time Structure of the Program Presented in Figure 8-2

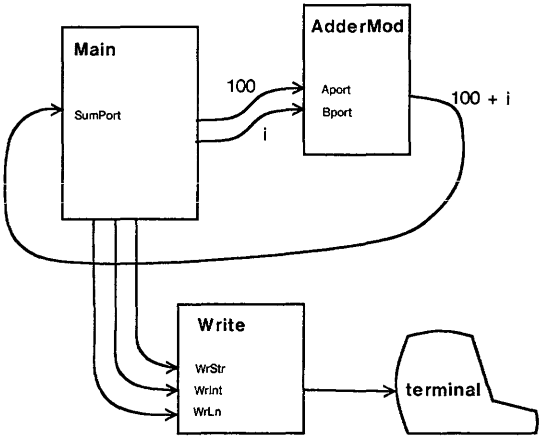


Table 8-8 Syntax for AMPL Programs and Modules

```
<program> ::= { <constant declarations> }#
              { <type declarations> }#
              <module declarations>

<module declarations> ::= { <module definition> };+

<module definition> ::= module <identifier>
                        { <parameter list> }# ;
                        <block>

<block> ::= { <constant declarations> }#
            { <type declarations> }#
            { <port declarations> }#
            { <variable declarations> }#
            begin { { <statement> }# };* end
```

The **if** statement is reminiscent of its counterpart in PL/I. The expression must be of type *Boolean*. Notice that the statement is terminated by the symbol **end**. Statement lists may follow the symbols **then** and **else**. Either or both statement lists may be empty. The entire **else** clause is optional. AMPL's only iterative construct is a standard **while** statement.

Table 8-9 AMPL Basic Program Elements

<*letter*>	::=	A | B | C | . . . | Z | a | b | c | . . . | z
<*digit*>	::=	0 | 1 | 2 | 3 | . . . | 9
<*stringchar*>	::=	" " | ! | " " " | # | $ | % | & | (|) | * | + | , | – | . | / | : | ; | "<" | ">" | ? | @ | [| \ |] | ↑ | – | ' | "{" | "|" | "}" | ˜ | <*digit*> | <*letter*>
<*identifier*>	::=	<*letter*> { <*letter*> | <*digit*> }*
<*integer*>	::=	{ <*digit*> }$^+$
<*string*>	::=	' { <*stringchar*> | " }* '
<*constant declarations*>	::=	**const** { <*constant definition*> }$^+$
<*constant definition*>	::=	<*identifier*> = <*constant*> ;
<*constant*>	::=	{ { + | – }$^\#$ { <*identifier*> | <*integer*> } | <*string*>
<*variable declarations*>	::=	**var** { <*variable definition*> }$^+$
<*variable definition*>	::=	<*identifier list*> : <*type*> ;
<*variable*>	::=	{ <*identifier*> | <*variable*> [<*expression*>] | <*variable*> . <*identifier*> }

8.2.3. *Ports and Messages*

AMPL processes communicate via *ports*. One process *sends* a message to a port, from which it may later be *accepted* by a receiving process. Several examples of **send** and **accept** statements appear in the adder program. A port belongs to the receiving process. Only one process can receive messages from a given port, and that right cannot be transferred. This decision simplifies the implementation because it eliminates a level of indirection (mapping from ports to receivers).

Three ports are defined in the adder program, two for the *AdderMod* to accept the two addends and one in the *main* module where the result is to be returned. All three ports are of type integer, because both addends and the sum are integers. The addend ports have a buffer size of one (in parentheses); they can buffer only one message at a time. Thus only one set of inputs can be waiting for service simultaneously; processes that attempt to send to a full buffer will be suspended, just as in MEDUSA. No buffer size is specified in the *SumPort* declaration, so the default buffer size is used. The syntax for port declarations, along with **send** and **accept** statements, is given in Table 8-11.

All three ports in the adder program are defined using the first form of a <*port definition*>. The second form is a list of port specifications in parentheses.

Table 8-10 Syntax for AMPL Expressions, Iterations, and Conditionals

<*expression*>	::=	{ <*conjunction*> | <*expression*> **or** <*conjunction*> }
<*conjunction*>	::=	{ <*negation*> | <*conjunction*> **and** <*negation*> }
<*negation*>	::=	{ <*comparison*> | **not** <*comparison*> }
<*comparison*>	::=	{ <*sum*> | <*sum*> <*compare op*> <*sum*> }
<*compare op*>	::=	"<" | "<=" | ">=" | = | ">" | "<>"
<*sum*>	::=	{ <*product*> | <*sum*> <*add op*> <*product*> }
<*add op*>	::=	{ + | − }
<*product*>	::=	{ <*factor*> | <*product*> <*mult op*> <*factor*> }
<*mult op*>	::=	{ * | **div** | **mod** }
<*factor*>	::=	{ <*term*> | − <*term*> }

<*term*>	::=	{ <*integer*> | <*string*> | <*identifier*> <*actual list*> }
		| (<*expression*>) | <*variable*> }

<*if statement*>	::=	**if** <*expression*> **then** { { <*statement*> }$^{\#}$ };$^{+}$ { **else** { { <*statement*> }$^{\#}$};$^{+}$ }$^{\#}$ **end**

<*while statement*>	::=	**while** <*expression*>
		do { { <*statement*> }$^{\#}$ };$^{+}$ **end**

Table 8-11 Syntax for AMPL Ports and **Send** and **Accept** Statements

<*port declarations*>	::=	**port** { <*port definition*> }$^{+}$
<*port definition*>	::=	{ <*port specification*> ; | ({ <*port specification*> };$^{+}$) ; }
<*port specification*>	::=	<*identifier*> : <*type*> { (<*constant*>) }$^{\#}$
<*send statement*>	::=	**send** <*expression*> **to** <*expression*>
<*accept statement*>	::=	**accept** <*accept list*>
<*accept list*>	::=	{ <*accept item*> },$^{+}$
<*accept item*>	::=	<*identifier*> (<*variable*>)

Messages sent to any of the listed ports will always be received in the order of their arrival times. All messages in the list logically share a single queue. (The software writer is free to use separate queues and time stamps, for example.) The size of the queue is the maximum of any constants given in the port specifications. If no constants are specified, a reasonable default value is computed. The use of this second form of specification is explained below.

All identifiers in port specifications are known globally so that given a reference to a module, all ports of that module can also be referenced. The *main* module is able to *send* to *APort* and *BPort* without having the port names explicitly declared as parameters. By contrast, all other identifiers declared inside a module are local to that module.

Declaration of a port within a module indicates that the port is to be instantiated whenever the module itself is instantiated. Therefore, if the module is instantiated twice, then separate ports are created for each process. Processes can accept messages only from their own ports. Any process with a reference to a port can send messages to it.

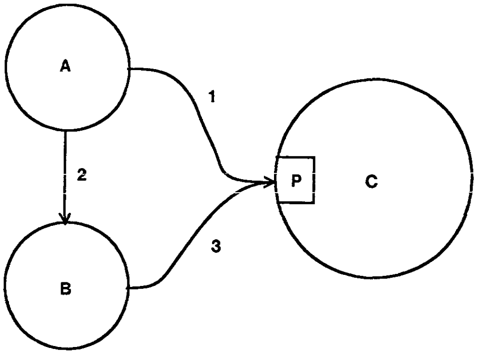
Send Statements. The syntax for the **send** statement includes two expressions. The first expression is evaluated to yield a value that becomes the content of the message. The second expression is evaluated to yield a port reference. A port reference is a variable of a *reference type*. Its format is *M.p*, where *M* is a module that has a port *p*. If the first expression is of type *T*, then the second expression must be of type **refport** *T*. For example, the **send** statement at line 13,

send *i* **to** *adder.APort*;

sends the integer *i* to *adder.APort*, which is of type **refport** *integer*, since *Aport* is defined as an integer port. It is an error to send to a null port descriptor or to a process that has halted. In the event that the destination port buffer is full, the sender remains suspended until the message can be delivered.

When an AMPL process performs a **send**, it is suspended until the message arrives at the port (not until the receiver *accept*s it from the port, unless the port is full). This guarantees that messages arrive at a port in the same order in which they were sent. More precisely, the following property is obtained: *If a* **send** *to a port completes before another begins, the first message sent will be the first one received.* In the case where the sends are from the same process, more efficient protocols can maintain a chronological ordering, but consider the case illustrated in Figure 8-4. Suppose process *A* wants to deliver a message to port *P*, after which process *B* will deliver a message, and these messages must arrive in order. Also suppose that **send** statements do not wait for delivery. After process *A* sends to port *P* (message 1 in the figure), *A* instructs process *B* to send a message by sending message 2. *B* responds by sending message 3. But wait! Message 1 may not have arrived at port *P* because process *A* never waited for its delivery. In AMPL, Process *A* cannot proceed until message 1 is delivered, so no race condition can arise. This sort of synchronization is used only occasionally in AMPL programs, but failure to wait for delivery could result in unreliable and very mysterious program behavior.

Figure 8-4 Two Processes Sending Ordered Messages to a Single Port



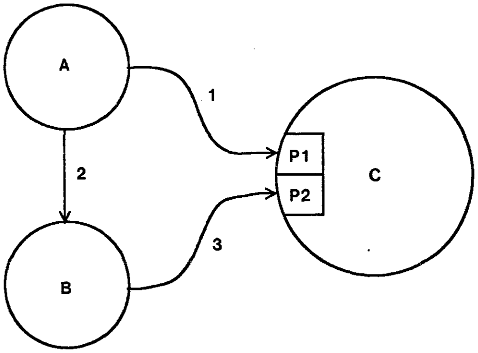
Waiting for delivery also facilitates flow control. If the buffer at the receiver is full, the message can be discarded because the sender is suspended with a copy of the message. A complete description of the message-passing implementation is given in Section 9.2.1. A process can have only one undelivered message outstanding, so this decision potentially reduces parallelism by blocking processes unnecessarily.

Sending Messages of More Than One Type. AMPL is a strongly typed language. The AMPL restriction that a port can receive messages of only one data type is essential to its strong typing. Safe garbage collection in AMPL is made possible by strong typing. It is sometimes necessary, however, to send more than one type of a message to a port.

To illustrate the problem, refer to Figure 8-4 and suppose that message 3, sent by process *B*, has a different type from that of message 1 but that process *C*, the receiver, must receive both messages in order. For example, message 1 may be the last data message of a stream, and message 2 may be an end-of-stream message. Because the messages have different types, two ports must be used, say *P1* and *P2* (see Figure 8-5). With the mechanisms presented so far, there is no way to guarantee that messages are received in order, unless extra synchronization messages are sent. The problem is that messages are, in effect, sorted first by ports and then by arrival time. We want to receive messages from *P1* and *P2* sorted first by arrival time and then by ports. This can be accomplished by using the second form of port declaration given in Table 8-11. Messages are *accept*ed in the order in which they were sent, regardless of what port in the list they were sent to. An alternative solution is to provide union types so that the data message (message 1) and end-of-stream message (message 2) can be sent to the same port. In a larger language, this probably would be the best choice because union types have other uses and are therefore more general.

Accept and Select Statements. Two AMPL statements can be used to receive

Figure 8-5 Synchronization of Messages Arriving at Two Ports



messages. The simpler of the two is the **accept** statement. The messages sent to a port can be received only by the process in which the port is declared. If only one <*accept item*> is named in an **accept** statement, a message is removed from the indicated port's message queue and is assigned to the indicated variable. If the queue is empty, the process suspends until a message arrives.

If more than one accept item is present in the accept list, each port identifier must be distinct. The process suspends until a message is present at each port's message queue. Then one message is removed from each queue and assigned to each variable in the order accept items are listed. An **accept** statement with a list of accept items is identical to a sequence of **accept** statements with one accept item each, except that with a sequence of **accept** statements, some messages may be accepted before all messages are present.

AMPL's **accept** statements are fine for receiving messages from a single port or from *all* ports in a set, but they are not useful when a process wants to receive a message from *any* of a set of ports or when a list of ports is declared to allow messages of different types to be queued in a first-come first-served fashion. In these cases, a **select** statement must be used.

The **select** statement is a generalized nondeterministic form of the **accept** statement. It allows a process to act on any of several message arrivals and to place constraints on which message is accepted next. One and only one of the select alternatives is executed. An alternative is said to be *enabled* when its conditional expression is true and a message is waiting in the queue for each port specified in that alternative's accept list. Again, ports in the accept list must be distinct.

If no alternatives are enabled, the process suspends until an alternative is enabled. If one alternative is enabled, messages are accepted as in the simple **accept** statement, and the corresponding list of statements is then executed. If more than one alternative is enabled, the one least recently executed is selected. This provides a fair choice in the sense that if a given clause in a given alternative is enabled repeatedly when the select is executed, then the alternative eventually will be chosen. If no enabled alternative is the least recently executed (i.e., none has been executed at all), then the choice is made arbitrarily.

Figures 8-6 and 8-7 are two examples of **select** statements. The first statement (Figure 8-6), when executed inside a loop, will implement mutual exclusion on a shared piece of data. The second example (Figure 8-7) uses the built-in function *ready*. The *ready* function takes a port identifier (not a port reference) as its argument and returns *true* if the port has a nonempty message queue. The *ready* function is useful for indicating priority in **select** statements and for avoiding suspension when a message queue is empty. The code in Figure 8-7 grants requests to read or write to shared data. Writers have priority, and multiple readers are allowed.

Figure 8-6

```
select
    accept read(reader) then
      send data to reader end;
    accept write(data) then end
    end
```

Figure 8-7

```
select
     when ReadCount = 0 accept WriteRequest(writer) then
          send OK to writer;
          accept WriteDone(writer)
          end;
     when not ready(WriteRequest) accept ReadRequest(reader) then
          send OK to reader;
          ReadCount := ReadCount + 1
          end;
     accept ReadDone(reader) then
          ReadCount := ReadCount - 1
          end
     end
```

8.2.4. Types

Except for the addition of reference types, AMPL types are similar to Pascal or Modula types. As in Modula, there are no types that must be dynamically allocated within module instances. Unlike Modula and Pascal, types need not be declared in any particular order. Forward references and recursive types are handled properly. Types are equivalent if they are structurally the same. For arrays, the number of components must be identical, and the component types must be equivalent. For records, the types of corresponding fields must be equivalent, and the number of fields must match. All subrange types are equivalent to type *Integer*, but the

Table 8-12 Syntax for AMPL Types

<type declarations> ::= **type** { *<type definition>* }$^+$

<type definition> ::= *<identifier>* = *<type>* ;

<type> ::= { *<simple type>* | *<array type>* | *<record type>* | *<ref type>* }

<simple type> ::= { *<constant>* .. *<constant>* | *<identifier>* }

<array type> ::= **array** [*<simple type>*] **of** *<type>*

<record type> ::= **record** *<field list>* **end**

<field list> ::= { *<field>* };$^+$

<field> ::= *<identifier list>* : *<type>*

<identifier list> ::= { *<identifier>* },$^+$

<ref type> ::= { **refport** | **refmod** } *<type>*

software writer may assume that the value of a variable of a subrange type is bounded by the constants specified in the type declaration. Reference types are equivalent if the referenced types are equivalent. No two modules are considered equivalent; this is the only case where structural equivalence is not applied. The syntax of AMPL types is given in Table 8-12.

Since floating-point numbers are not permitted, *integer* and *Boolean* are the only predefined types. Arrays are one dimensional, but multidimensional arrays can be simulated easily using arrays of arrays. The syntax for referencing a structure is like that of Pascal (see Table 8-9). The field selection (dot) notation also references a port within a module as explained in Section 8.2.3.

Section 8.2.3 introduced the **refport** data type. Variables of type **refmod** are also permitted. The *create* function returns **refmod** values; only in this way may **refmod** values be created. A **refmod** value is essentially a pointer to a process. In the adder program, the *adder* variable is of type **refmod** *AdderMod*. If another process were created to execute the *adder* module, another variable of type **refmod** *AdderMod* would be needed to store the reference to the process. That is, the code

```
var adder, adder2: refmod AdderMod;
          :
adder := create(AdderMod, self.SumPort);
adder2 := create(AdderMod, self.SumPort);
```

instantiates two *AdderMod* processes, which are capable of performing two additions in parallel. Notice the use of *self.SumPort* as a parameter of the *create* function. Every process has an implicitly declared variable called *self*, which is initialized with a reference to the process itself. In a process that is an instantiation of module *M*, the type of *self* is **refmod** *M*. If module *M* declares a port *p*, then *self.p* refers to the name of port *p* in that process. Consequently, the *main* process passes both *AdderMod* processes a reference to its *SumPort*. Both the adder processes return values to the same port, and these values may arrive out of order. Nonetheless, the parallelism is useful if the adder processes are slow and a long sum is being computed. The order in which the numbers are added does not matter, as addition is associative and commutative.

8.2.5. Functions

Functions cannot be defined by the AMPL programmer, but there are two system-defined functions, *create* (Section 8.2.1) and *ready* (Section 8.2.3). An alternative syntax for process creation, which is more consistent with AMPL's message-passing facilities, would be to use a language-defined port called *create*. Messages would be sent to the *create* port to specify the process to be created, the parameters for the process, and a *reply* port to which a reference to the created process would be sent. This alternative approach would allow the creator to continue executing in parallel with the creation of the new process, but the overall time to create a process would be slightly longer.

8.2.6. Language-Design Issues

Dannenberg [Dannenberg 81] studied several programming-language issues in the design of AMPL. Many of them are too specific to delve into here, but a couple are of particular interest because they represent different design philosophies from other programming systems developed on Cm*.

Storage Allocation. Several aspects of typing in the AMPL design simplify the storage-allocation problem. The typing system is restricted so that the compiler knows the size of all objects. AMPL does not provide procedures (although procedures may be simulated with other mechanisms). The storage-allocation strategies, in turn, affect the design of garbage collection.

The absence of procedures allows the compiler to determine the maximum stack size of a process. The maximum size for a given process is bounded and depends only on static properties of the code. Heap storage is required only for processes, since variables cannot be allocated dynamically and there are no "pointer" types in AMPL.

A more elaborate and flexible set of types, as illustrated by modern programming languages (e.g., Ada), is desirable for many applications. The typing system of AMPL was chosen primarily to keep the implementation simple; it provides enough power to allow interesting parallel programs to be written without overly complicating the implementation.

Processor Allocation. AMPL has no facility for specifying physical process locations. Processor allocation is performed by the run-time system. Because processes do not share memory and interact via messages, it is particularly easy to move processes from one processor to another. Thus allocation can be varied dynamically. Where there are large numbers of processors, it is difficult to know how and where to locate processes, especially in programs that create processes dynamically. The philosophy behind the design of AMPL is that programs should specify a high degree of parallelism, with far more processes than processors. This helps ensure that the available parallelism will be utilized even though the allocation of processors is less than optimal. The absence of shared memory means that the set of variables used by a process is always known, so it is easy to arrange to keep variables physically near the processor that accesses them. Very little research has been done relating to processor allocation in this sort of system. The AMPL implementation has hooks to allow process migration, but none has been performed.

8.2.7. Retrospectives

AMPL demonstrates how message passing can be used to express interprocess synchronization and communication in a high-level language. The expressive power of the primitives in AMPL allowed the development of programs with more complex control structures than previous Cm* application programs. While many synchronization errors were found in the run-time system as it was tested, the synchronization in all the AMPL programs was correct from the start (which is not to say there were no other bugs). The high level of correctness can be attributed to several factors. First, the absence of shared memory forces the programmer to consider synchronization whenever processes interact. Second, type checking helps ensure that data is interpreted correctly and that interfaces between processes are correctly implemented. Finally, because AMPL has built-in mechanisms for process creation and message passing, the programmer can avoid the implementation of tedious operating-system interfaces and concentrate on the problem at hand. In addition, the compiler can apply some checking to these operations.

A pattern or style of programming was observed in the test programs. Nearly all modules in existing AMPL programs can be viewed as implementations of abstract types whose operations are invoked by sending a message to a port. If we were to design another language for parallel processing, we would attempt to use this as a basis for the language rather than message passing. The invocation of an abstract operation normally would be implemented by sending a message, but the actual construction of the message and its receipt would be hidden from the programmer. The compiler might then be able to detect special cases where simple mechanisms could be substituted for the full-blown message-passing scheme of AMPL.

8.3. Summary

This chapter has presented two dissimilar approaches to programming environments. The first approach, represented by TASK and MEDLINK, provides a set of primitives external to a sequential programming language language to support operations specific to parallel programming. These include resource allocation, creation of independent processes, interprocess coordination and communication, object management, and scheduling.

Both MEDUSA and STAROS embody the notion of a task force. TASK and MEDLINK allow task forces to be synthesized by means of programming templates —"blueprints" for a parallel computation. These templates represent the macrostatic parallel program structure in terms of modules, their communication patterns, and resource-allocation directives. The templates are separate from the sequential code that makes up the modules, which usually are written in BLISS. Although this approach generally is useful for any multiprocessor, TASK and MEDLINK solve resource-allocation problems specific to Cm*. Both tend to require the user to be familiar with the Cm* architecture and object organization. Also, both require the user to know two languages (BLISS and TASK or MEDLINK). Aside from these drawbacks, TASK and MEDLINK have proven extremely useful in contrasting and understanding the logical-to-physical resource-mapping problem.

The second approach is to design a new language and compiler to support programming for a multiprocessor. AMPL is an attempt to do just that. It emphasizes a model of computation where sequential modules cooperate through messages. AMPL constructs describe three types of basic elements: modules, ports, and messages. Within a module definition, one can define constants, types, variables, and ports. AMPL is a strongly typed language. Except for the addition of reference types, AMPL types are similar to Pascal or Modula types.

The discussion of AMPL continues in Chapter 9, which contrasts its software environment with another very different programming environment on Cm*.

Acknowledgment. The material on AMPL in this chapter has been adapted from [Dannenberg 81].

9. Other Software Environments

Almost all the processes that run on an ordinary computer system are run with the support of an operating system, which provides utilities such as I/O routines to simplify the task of the programmer. Indeed, most of the experiments performed on Cm* have utilized one of its two operating systems. But Cm* is no ordinary computer system, both because of its experimental nature and because of the extraordinary freedom to change its virtual machine, even down to its addressing structure, by modifying Kmap microcode. Throughout the history of Cm*, three software environments other than operating systems have been used to run processes. One of these was developed to run large benchmarks before the operating systems were fully available. The other two were motivated by a desire to make the construction of large task forces cheap by simplifying the creation and management of processes.

This chapter begins with a brief description of NEST, which was brought up quickly to run experiments and compare different Cm* microcode systems before the software portions of STAROS and MEDUSA were completed. It progresses to a description and evaluation of the AMPL run-time system, which provides many operating-system-like facilities in an effort to circumvent some of the overhead of MEDUSA process creation and message passing. The chapter concludes by exploring the ECHOES experiment, whose goal was also inexpensive process creation but which went about it by a diametrically opposite approach: maximizing, rather than minimizing, the sharing of information. It is a testimony to the flexibility of the Kmaps and the Cm* architecture that both AMPL and ECHOES were able to achieve their goals.

Specifically excluded from this chapter are other systems and tools that more properly fall in the realm of firmware than software. Among these are Smap (the simple microcode), which is described in Appendix D, and several microcode-development tools. Among the latter are the microassembler CMIC, the register-allocation program PMIC, James Gosling's MUMBLE (Most Unlikely MicroassemBLEr), and John Ousterhout's KDP (Kmap Debugging Package). All these were described in [Jones and Gehringer 80].

9.1. NEST

NEST is the work of Jarek Deminet who was the first to run extensive multicluster benchmarks on Cm*. Previous benchmarks had been run on a ten-processor system with serial lines attached to all Cm's so that all software could be loaded directly through the Cm* Host. The 50-processor system still had only 10 serial lines, so a more sophisticated environment had to be developed. The operating systems also provide such facilities, but they were not yet operational.

The first approach was to identify a particular Cm as the *master* and load all code into memory through its serial line. The master then transferred parts of the code to other *slave* Cm's and initialized the global data, which resided in its memory. Note the similarity between this organization and the sample task force presented in Section 4.1.1. After starting the slaves, the master joined them in running the parallel algorithm. Simultaneously, however, it had to handle clock interrupts and monitor the other processors to report final results. This version of a benchmark was called the *standalone* experiment.

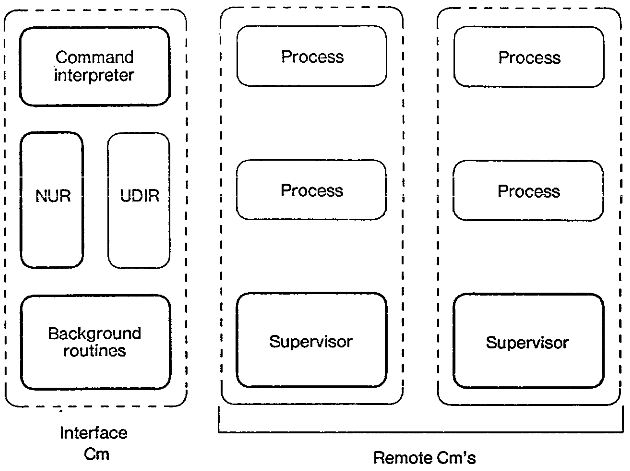
Standalone experiments had several shortcomings, which provided the impetus for a more sophisticated environment. First, the user could not keep track of the progress of the experiment because the master and the slaves were busy running the benchmark and could not communicate with a terminal without perturbing the measurements. This was a serious inconvenience for longer experiments, which often lasted several hours. Even without performing terminal I/O, the master was slowed down by its other duties, such as fielding clock interrupts and implementing the counter used to time the experiment. Furthermore, the master needed to contain the master and slave code and the I/O packages, which seriously constrained the amount of address space left over for global data and thus limited the size of the experiments that could be run.

The NEST (Nuclear Environment for Software Tests) environment was developed in response to these problems. It dedicated a single Cm, known as the *interface module*, to communicating with the terminal and monitoring the other Cm's. Essentially this meant divorcing the responsibility for communication from the processors running the experiment (Figure 9-1). It permitted the user to interrogate the task-force status at any time with minimal perturbation of the results. Each of the other Cm's, called *remote Cm's*, contained a small supervisor to handle interrupts, communicate with the interface module, and multiplex the processes that were running the experiment. In many cases, however, only one process was allocated per processor, rendering multiplexing unnecessary.

In addition to the supervisor, each Cm also contained a set of background routines, which provided medium-level memory-management operations. Different versions of these routines were created to serve as interfaces to all three microcodes—Smap, MEDUSA, and STAROS. Since NEST ran without the operating systems themselves, descriptor lists and other data structures that were needed by the microcodes were created by NEST itself. Not all the microcode operations could be used by programs running under NEST; because the microcodes were dissimilar in design, some of the functions of one would have been difficult or impossible to simulate in others. Only the address-space, synchronizing, and initializing operations were used.

In its time, NEST greatly facilitated running experiments and comparing the performance of the microcodes. It was used by Deminet in all his experiments (Sections A.7, A.8, and A.9) and by Carey's power-systems simulation. It demonstrated that simple software support was adequate for running rather large benchmarks. Its interface facilities were later used as the basis for the MEDUSA user interface.

Figure 9-1 The NEST Structure



UDIR--User-defined Initializing Routines
NUR--Nest Utility Routines
—— User-defined objects
—— Nest-defined objects

9.2. The AMPL Run-Time System

Chapter 8 contained a description of the AMPL language. The implementation also raises a number of interesting issues because it encountered many of the same problems as other Cm* systems but chose to solve some of them in different ways. This section surveys several aspects of the AMPL implementation and reports performance measurements on it.

9.2.1. Implementing AMPL on Cm / MEDUSA*

While the Cm* hardware is almost ideal for AMPL, the process and message-passing abstractions provided by MEDUSA are poorly matched to its requirements. As a consequence, the AMPL run-time system implements its own processes, ports, scheduling, and storage allocation. MEDUSA activities exist in separate address spaces, and the smallest grain of protection provided by hardware is a 4,096-byte page. The smallest activity in MEDUSA is at least this large. At most 16 processes can be created in any computer module. The designers of MEDUSA intended that activities be rather static entities. It was decided to use BLISS-11 coroutines to implement AMPL processes. This allows processes to share an address space and allows the run-time system to choose a process representation that is optimized for AMPL processes. Because several AMPL processes can share a single address space, there is no lower bound on process size due to hardware restrictions. Note the similarity of this solution to the strategy adopted for MEDUSA utilities (Section

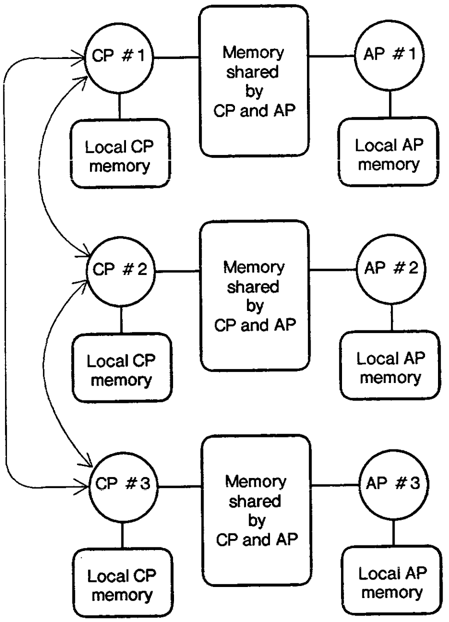
5.3), where coroutines were used to avoid the overhead of dedicated activities to infrequently invoked utility service classes.

Ports. MEDUSA has a fairly elaborate message-passing system. At first sight, MEDUSA seems like an ideal base for AMPL because it emphasizes the use of message-based communication over shared memory. Upon closer inspection, however, it is found that MEDUSA pipes are only remotely similar to AMPL ports. First, pipe creation is a time-consuming operation due to protection, addressing, and memory-management problems handled by the operating system. One way to circumvent this problem is to preallocate a large number of pipes in a commonly accessible place (MEDUSA's shared descriptor list) and assign pipes to processes dynamically. Pipes would be reused rather than destroyed. The SDL allows a maximum of only 512 pipes, which would be too few for many programs. Another problem with pipes is that if a pipe is full when a message is sent, the sender is blocked. Assuming that AMPL processes are implemented as coroutines, what we desire is the suspension of one coroutine, not the suspension of the MEDUSA process that is responsible for executing many coroutines (AMPL processes). For these reasons, MEDUSA messages cannot be used directly as AMPL messages. MEDUSA pipes are used, however, in the implementation of AMPL ports, in the manner described below.

Process Pairs. An ideal implementation of AMPL would use at least two types of processors. One would be optimized to perform message-passing and resource-allocation functions, and the other would be designed to execute AMPL programs efficiently. Earlier it was mentioned that the Kmaps could perform the communication functions. In our implementation, computer modules are used in pairs. One processor in each pair is called the *communication processor*, or CP, which creates processes, delivers messages, and performs garbage collection. The other processor is called the *application processor*, or AP, and it actually executes AMPL programs.

The CP and AP interact closely and share the memory used for AMPL processes. The run-time system is composed of many logically identical CP/AP pairs (see Figure 9-2). These pairs could be physically implemented on one computer module, but there are at least three reasons for not doing so. First, it is desirable to service incoming messages promptly to avoid letting a pipe become full. A full pipe might block some other process and perhaps lead to deadlock. One solution is to use interrupts to notify the receiver immediately, but MEDUSA does not provide a means of interrupting a process when a message arrives. Second, while the total memory of Cm* is large, individual computer modules have a limited amount of memory; typically about 40K bytes are available for programs and data. Separating the CP and AP allows the code for each to reside on separate machines. Third, using two physical processors can provide a large degree of additional parallelism, since message-passing operations overlap other computations. The decision to use processors in pairs (rather than, say, one communication processor for every two application processors) was somewhat arbitrary. It turned out that some programs

Figure 9-2 Basic Structure of the AMPL Run-Time System



saturate the CP and others saturate the AP, so the right mixture is defined to a large extent by the nature of the AMPL program. Dynamic optimization of processor assignment would greatly complicate the implementation.

While there is very close interaction and a high degree of communication within a CP/AP pair, the interaction of one pair with another is much more limited. Communication between pairs is almost entirely conducted via messages. While not always the most efficient organization, this approach has the advantage of reducing complexity to a manageable level. Message passing introduces some overhead. More data is moved and copied to pack, send, receive, and unpack messages. Messages also take substantial amounts of Kmap processing time. A shared-memory approach would require additional code for synchronization. The possibility for errors is large, and the errors would not necessarily be reproducible because of timing considerations. Moreover, a fair number of the memory references would be intercluster. By not splitting CP/AP pairs across clusters, we can avoid intercluster references with the message-passing approach. Because there are so many opposing factors, it is unclear how a shared-memory approach would compare to the present implementation, but it would certainly be less understandable.

All processes on a given CP/AP pair share a single address space. Subscripts must be checked to prevent inadvertent memory accesses, which might corrupt run-time system structures or other processes. The shared address space makes context switching more efficient and, as noted above, allows processes to be smaller than a single page object (4K bytes).

Send Statements. A number of steps are performed to implement **send** statements. Execution begins when an AP reaches code for a **send** in a process. Execution of the **send** involves the use of several MEDUSA messages and several processors. At the completion of the **send**, the AMPL message must reside in the message queue of the destination port, and the AP resumes execution of the next AMPL statement.

The AP that initiates the **send** constructs a MEDUSA message in a *send buffer* allocated by the compiler. All MEDUSA messages sent by the AMPL run-time system have, as the first word, a function code that identifies the type of the message. (These function codes will be written in SMALL CAPITALS in the text.) A MEDUSA message with a function code such as SEND will also be referred to as a SEND message or simply as a SEND. The MEDUSA message contains the following:

1. A function code (SEND).
2. The destination port name.
3. The name of the sender.
4. The actual AMPL message.

The AP directly sends this message to the CP indicated by the destination port name. At this point, the AP suspends the current process. The next process on the ready-to-run queue is executed while the message is delivered. The time to perform this context switch is much less than the time required to deliver the message.

When the CP receives a message, it first reads the function code and then calls a handler to perform the requested function. Figure 9-3 outlines the structure of the CP program. In this case, a *receive handler* is called. The receive handler reads the process index from the destination port name and locates the *process frame* in the *process descriptor table* that resides in each CP / AP pair. Each process frame contains storage for message queues. The frame also contains the address of a *module template*, which is a storage map used to locate message queues and variables in the process frame. Once the CP finds its way to the appropriate queue, three cases arise:

The queue is empty. The message is copied into the queue. If the receiving process is suspended, it is moved to the ready-to-run queue because the process may be waiting for the message.

The queue is neither empty nor full. The message is copied into the queue, and no rescheduling is attempted.

The queue is full. The name of the sender is placed on a special *waiting* list associated with the buffer, and the message is discarded.

In the first two cases, the message is delivered, so the sender must be rescheduled. The receive handler sends a MEDUSA message with the function code

Figure 9-3 The Structure of the CP Program

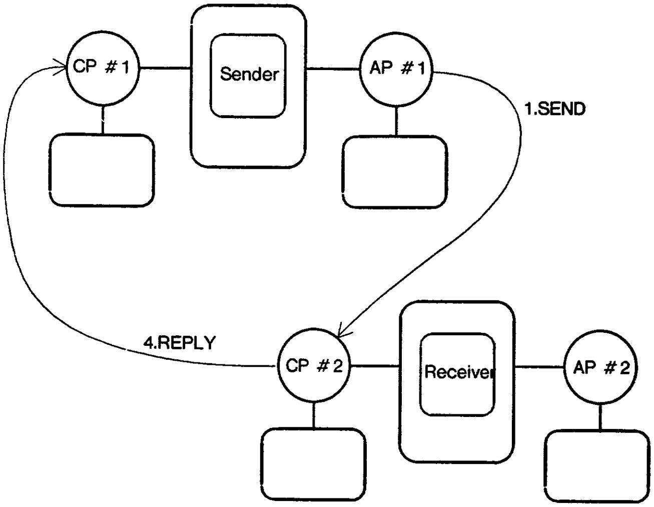
```
initialize data structures
loop
    if message in high-priority pipe then
        receive message
        case message function code of
            SEND:       call receive handler
            REPLY:      call reply handler
            CREATE:     call create handler
                        :
            end case
    elsif message in low-priority pipe then
        receive message
        case message function code of
            GCMARK:     call gcmark handler
            GCSCAN:     call gcscan handler
                        :
            end case
        end if
    end loop
```

REPLY to the CP associated with the sender (see Figure 9-4). This completes the processing at the receiver's end. When the sender's CP receives the REPLY message, a *reply handler* is called. This routine simply reschedules the sending process by placing it on the AP's ready-to-run queue. Finally, the AP removes the process from the queue and resumes execution.

In the third case (a full message queue), no REPLY message is returned, and the sending process remains suspended (see Figure 9-5). The name of the suspended process is saved on a waiting list. Whenever an AMPL message is removed from a queue, the waiting list is checked to see whether any senders are suspended because of a previous attempt to send an AMPL message. If the waiting list is not empty, a REQUEST message is sent to the CP associated with the name at the head of the waiting list. When the CP receives the REQUEST, it finds the suspended process and locates its send buffer, which still contains a copy of the original SEND message. The function code is changed to REQUEST-REPLY, and the (MEDUSA) message is again sent to the destination CP. This time, it is known that there is room in the destination port, so the suspended process is rescheduled without waiting for a reply. When the destination CP receives the REQUEST-REPLY message, the correct message queue is again located, and the message is placed in the queue. As before, if the queue changes from empty to nonempty and the process is suspended, it is rescheduled.

Messages are delivered in order, even when senders are blocked waiting for a queue to become nonfull. The actual procedure used by the receive handler is a little more complicated than described. The waiting list is a FIFO queue of process

Figure 9-4 Sending a Message to a Nonfull Port



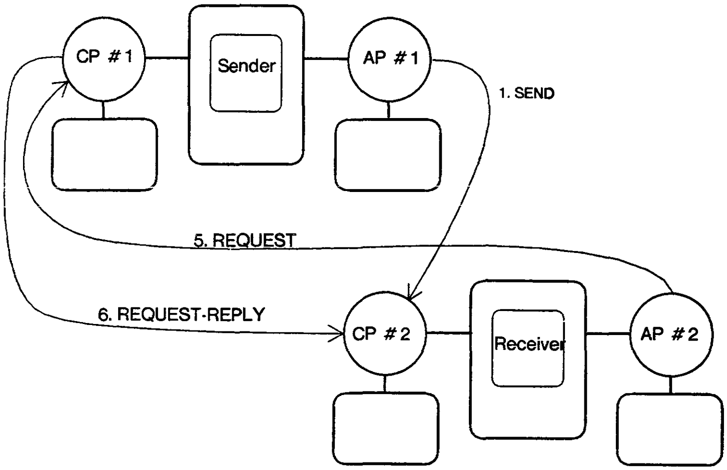
1. AP #1 sends message to CP #2.
2. AP #1 suspends sender.
3. CP #2 copies message into receiver's message queue.
4. CP #2 sends reply to CP #1.
5. CP #1 reschedules sender.
6. AP #1 resumes execution of sender.

names. A message can be placed directly into the message queue only if the waiting list is empty. Otherwise, even if space is available for the message, the sender's name is added to the waiting list and the message is discarded. This prevents a message from being placed ahead of one sent previously and preserves space for messages that have been requested. A name is removed from the waiting list only after the REQUEST-REPLY arrives to fill the reserved slot in the queue.

Garbage Collection. This section describes the parallel garbage collector used in the AMPL run-time system. Like the STAROS *Garbage Collector* (Section 6.6), it consists of cooperating run-time processes. This contrasts with most parallel garbage collectors [Dijkstra 78, Kung and Song 77], which employ only a single garbage-collection process. The AMPL garbage collector was operational before the STAROS *Garbage Collector*, and, to our knowledge, it was the first working garbage collector with multiple tasks creating garbage and multiple tasks collecting garbage. We will consider AMPL garbage collection in detail and note the similarities and differences between the AMPL and STAROS garbage collectors.

All garbage collectors are based on the following simple "mark-scan" algorithm.[1]

Figure 9-5 Sending a Message to a Port that is Initially Full



1. AP #1 sends message to CP #2.
2. AP #1 suspends sender.
3. CP #2 puts sender's name in receiving port's waiting list.
4. Receiver accepts a message, making room for a new one.
5. AP #2 sends request to CP #1.
6. CP #1 resends sender's original message.
7. CP #1 reschedules sender.
8. CP #2 delivers message.
9. AP #1 resumes execution of sender.

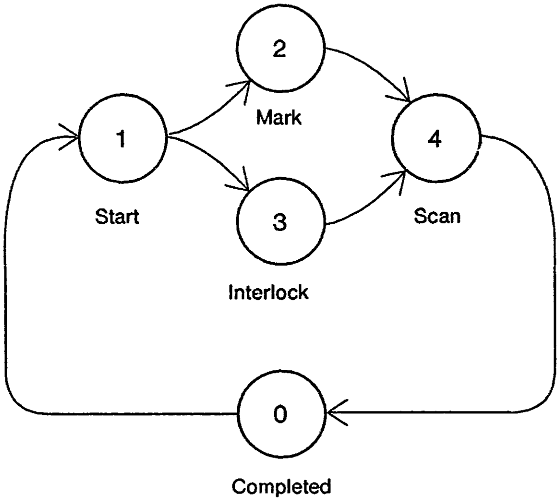
1. Mark the set of objects known not to be garbage (called the *root objects*).
2. Mark all objects that can be reached by following pointers from the marked objects. Repeat this step until no more unmarked objects can be found.
3. Reclaim the storage used by any object not marked; these objects are garbage because there is no way to reach them by following pointers from the set of root objects. The remaining nongarbage items will be referred to below as *reachable* objects.

The AMPL garbage collector consists of four phases, two of which run in parallel, as illustrated by Figure 9-6. The algorithm still corresponds to the classic mark-scan algorithm, and the extra phases are for synchronization only.

Garbage collection is performed by the communication processors. Recall the structure of the CP (Figure 9-3). Each CP has two MEDUSA pipes; one is for high-

[1] Even the semispace copying algorithm [Baker 78] fits this description. Rather than setting a "mark bit," objects are marked by moving them to another block of memory. Thus an address bit is used as the mark bit [Hibbard 80]. Compare this with the STAROS algorithm in Figure 6-5.

Figure 9-6 Garbage collection phases



priority messages such as SEND messages, and the other is for low-priority garbage-collection messages. The CP continually polls its two pipes for messages. The low-priority pipe is examined only if the high-priority pipe is empty. Every message received has a function code, which is used to specify a handler for that message. Each handler performs a small task, sometimes sending new messages. Garbage collection runs as a background task because garbage-collection messages are sent to the low-priority pipe. This organization effectively multiplexes the CP to perform many tasks while using only one process. Since CPs interact almost exclusively through messages, a large amount of overhead for synchronization within each message handler is avoided. For globally synchronizing garbage-collection phases, global counters and MEDUSA-provided atomic increment and decrement operations also are used. These operations are much faster than MEDUSA messages, but their use is purely an optimization of a message-based implementation.

WHAT TO COLLECT. The only dynamically allocated objects in AMPL are processes and messages. Messages take up storage only until they are delivered, so garbage collection is concerned only with reclaiming process names (descriptor-table entries) and process frames. The initial (root) set of processes known not to be garbage is composed of the following:

1. The processes being executed by APs.
2. Processes in the ready-to-run queues.
3. Processes whose names are on the create waiting queue of some CP. (These processes are blocked waiting to create a process.)
4. Process names in messages waiting to be delivered.

Any process reachable from this set is marked; the rest are garbage. If a name is identified as garbage, and the name refers to a process frame, then the process frame is not only suspended, but it can never be rescheduled (no other process can send it a message). Therefore, the frame is returned to the free storage pool.

PHASE 1. Any CP can start garbage collection. To do so, a memory location global to all CPs is locked with a MEDUSA "test and set" operation to prevent two CPs from starting garbage collection simultaneously. Then, GCSTART messages are sent to each CP. The CP that starts garbage collection is called the *garbage-collection master*.

When a CP receives a GCSTART message, it finds all the root processes and sends GCMARK messages to the CP associated with each name. (Actually, an optimization could eliminate the need to send messages from a CP back to itself; for details, see [Dannenberg 81].) The purpose of GCMARK messages is explained below. The CP then sets a local state variable to *GCstarted* and returns a GCSTARTREPLY message to the garbage-collection master, which is identified by a field in the GCSTART message.

Some root names cannot be accessed by any CP when garbage collection is started. These are names that are in MEDUSA pipes. Consider a process suspended waiting for a REPLY message. The reply could have the only reference to this process, so if garbage collection completed before this message arrived, the suspended process would be erroneously collected. To prevent this, the garbage-collection master sends an INTERLOCK message to each high-priority port after a GCSTARTREPLY message is received from each CP. Receipt of the INTERLOCK message is described below.

PHASE 2. The second phase is concerned mainly with receiving and handling GCMARK messages. Each GCMARK message contains one process name. The handler for these messages starts by setting the mark bit for that name in the process descriptor table. If the name was previously unmarked, and there is a process frame for the name (the process has not terminated), then the frame is scanned using the module descriptor to locate all process and port names. For each nonnull name found, a GCMARK message containing the name is sent to the CP indicated in the name. (Just as a STAROS object name contains the number of the cluster where it is located, an AMPL process name contains the number of the CP / AP pair on which it runs.)

While a frame is being scanned, a lock is set on the frame to prevent the AP from copying any process or port names. Otherwise, the garbage collector might miss a reference as it is being moved, or it might read a name in a corrupt state (i.e., the first word of one name and the second word of another). In STAROS, objects are not locked as they are scanned. Instead, the microcoded ***Copy Capability*** instruction must be used to copy a name, and the microcode notifies the garbage collector (by placing its name in the garbage-collector deque; see Section 6.6). The corresponding action in AMPL would be either to require the CP to copy references or to have the AP inform the CP whenever a name is copied. In the latter case, locks would still be needed to prevent the CP from reading names in a corrupt state. The CP would release the lock after reading each name. The trade-offs are summarized below. For the method implemented,

- The CP performs only one lock operation per process scanned.
- The AP may be blocked for the time it takes to scan the frame for names.

For the alternate scheme,

- The CP performs one lock operation for each name scanned.
- The AP is at most blocked only for the time it takes to read one name.
- The AP must make a test, and possibly send a message, to the CP whenever a name is moved.

While marking is taking place, processes can be sending messages and creating new processes. The following invariant is maintained: *If a process is marked by the garbage collector, then a* GCMARK *message has been sent for each name contained in that process frame.* To guarantee this property, special precautions must be taken when sending messages and creating processes. Every SEND, REQUEST-REPLY, CREATE, and CREATE-REPLY message has a tag that is used to mark the message if the sender is marked. If a message is marked, then it is known that a GCMARK message has been sent for each name in the message because the names in the message are copied from variables in a marked process frame.

When an AMPL message is delivered to a port, the tag is checked. If it is marked, then no action is taken. If the message is unmarked, and the receiving process is marked, then a GCMARK message is sent for every name in the message before the message is placed in the receiver's message queue. This must be done to maintain the invariant without unmarking process frames.

In the case of create messages, the created process is marked to prevent its garbage collection. If the create message tag is unmarked, then a GCMARK message is sent for each name supplied in the actual parameter list.

Phase 2 completes when there are no more GCMARK messages and all reachable processes have been marked. Determining when this condition is reached is not simple because messages are sent and received asynchronously by all CPs. Even if all pipes are empty, there could be a message in transit. Completion is detected by maintaining a global counter (initially zero), which is incremented before each GCMARK message is sent. After a GCMARK message is received and the process frame is scanned, the global counter is decremented. The counter returns to zero when all GCMARK messages have been processed.

In STAROS, *Garbage Collectors* rely on real-time constraints to determine when the mark phase is complete. When it *appears* that marking is done, the *Garbage Collectors* wait for a predetermined time to make sure no new names are discovered. Since marking is assisted by Kmaps, processing is fast, and the time delay is short. A third method could be based on sending replies to each GCMARK message.

The mark phase is guaranteed to terminate in the following sense. Assume that unlimited memory and name space are available, so processing is never held up because space is unavailable. Garbage-collection messages are handled at a finite rate in each CP, but the high-priority messages—that is, non-garbage-collection messages that may contain names—can be handled at an arbitrarily fast but finite rate. The mark phase will terminate in a finite time. The proof is based on the invariant stated above, the fact that created processes are marked, and the fact that processes are never unmarked during the mark phase. The mark phase begins with a finite and nonnegative number of processes. It remains to show that the number of unmarked processes is decreasing.

Consider the set of reachable unmarked processes. There must be at least one reference from a marked process to one of these unmarked processes. By the invariant, a GCMARK message has been sent with that reference. The queues of GCMARK messages can get arbitrarily large, depending on the relative amount of processing time devoted to garbage collection, but the queues are always finite. Because garbage collection messages are handled at a finite rate, this message will eventually be received, the process will be marked, and the number of unmarked processes will decrease. Thus we see that the number of unmarked but reachable processes must go to zero. Eventually, all reachable process frames become marked, and no more GCMARK messages are sent, so the queues become empty. At this point the mark phase terminates.

In STAROS, the garbage-collector deque is used to prevent the *Garbage Collector* from missing a reference to a reachable object. If assignment of references occurs arbitrarily more frequently than garbage-collection operations, then the marking phase will not terminate. (This can occur even when the total number of names and objects is constant.) This property is not a problem in practice for STAROS.

PHASE 3. Recall that in phase 1, an INTERLOCK message was sent to the high-priority pipe of each CP. Phase 3 is the period between the end of phase 1 and the time all the interlock messages are received. Phase 3 overlaps phase 2, as indicated in Figure 9-6.

Some rather intricate synchronization involving another global counter is used to detect when both phases 2 and 3 have completed in all CPs. The STAROS analogue of phase 3 is the time delay used to check that all marking is complete. In STAROS, the time during which a name can be hidden (in Kmap registers) is bounded by a short interval, so waiting is preferable to the use of interlock messages.

PHASE 4. The CP detecting the end of phases 2 and 3 in all CPs sends a GCSCAN message to each CP. When this message is received, the process descriptor table is scanned by each CP, and unmarked processes are collected. As each CP completes its scan, it increments yet another global counter. When this counter reaches the number of CPs, the "garbage collection in progress" lock is reset to allow another garbage collection to be started. The period between garbage collections is referred to as phase 0.

9.2.2. *Performance Measurements*

A number of small programs have been coded in AMPL and executed on Cm*, with the goal of learning how the language influences algorithm design and affects performance. The measurements reveal run-time characteristics of parallel algorithms and the AMPL implementation. Those experiments that disclose more about the run-time system are reported in this chapter; those that relate more closely to parallel-algorithm performance are reported in Appendix A.

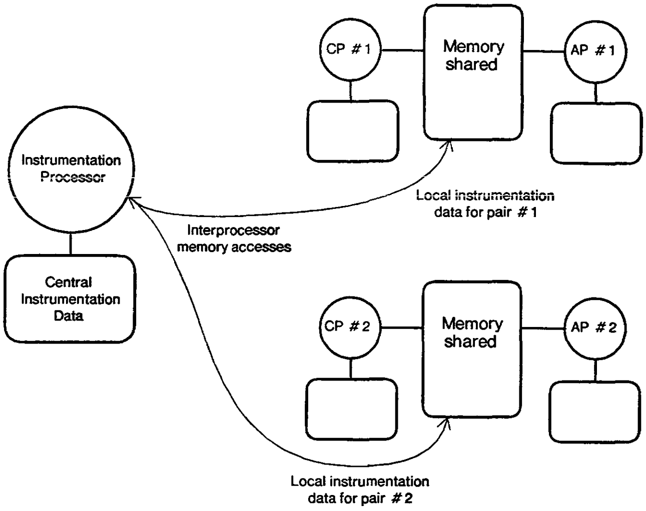
Some of the AMPL programs to be described exhibit a negligible communication cost, and the total execution time is determined almost entirely by the AP processing time. Other programs are dominated by the communication cost, and APs are idle much of the time. Changing the relative execution speeds of the AP and CP by optimizing the run-time system or increasing the quality of the code generator could radically change some of the results. For this reason, measures of program performance that are independent of relative execution speed are sought. One such measure is the ratio of information passed in messages to the information stored in or fetched from local variables. A low ratio indicates that most of the execution time is spent accessing local memory and performing local computation. A high ratio indicates that most of the execution time is spent communicating with other processes, and little local computation is performed.

Obtaining the measurements was greatly simplified by the fact that the AMPL compiler generates BLISS code rather than assembly code. Complex actions, such as AMPL **sends**, are generated as BLISS macros. These macros can easily be modified to collect a wide variety of performance data, such as counts of different message sources and destinations. It is also easy to eliminate instrumentation overhead when necessary simply by redefining all instrumentation macros to the null string.

A dedicated processor is used to gather this data at run time, so as to cause minimal interference to the run-time system. One advantage of this approach is that values such as the number of bytes sent in messages can be accumulated using single-precision arithmetic without danger of overflow. The instrumentation process periodically clears these accumulators and transfers their contents to a central, extended-precision accumulator. Figure 9-7 illustrates the instrumentation organization. Another use of the instrumentation process is to take samples of the state of the run-time system. AP and CP utilization are computed by sampling flags that tell when the AP and CP are idle. Using a separate process to collect data also allows each AP and CP to avoid time-consuming intercluster memory updates and reduces the code required for the AP and CP processors.

A few general comments must be made about the measurements. First, execution times of AMPL programs vary slightly from one run to the next. The execution speeds of the computer modules on Cm* vary (see Section 3.1.1). Many objects, including MEDUSA pipes and some memory pages, are dynamically allocated. The choice of object locations can also affect timings. Special efforts have been made to specify locations where placement is critical, but all measurements given in the following sections should be considered accurate only to within 5 percent. Con-

Figure 9-7 Instrumentation of the AMPL Run-Time System


CP #1
Memory shared
AP #1
Instrumentation Processor
Local instrumentation data for pair #1
Interprocessor memory accesses
Central Instrumentation Data
CP #2
Memory shared
AP #2
Local instrumentation data for pair #2

sequently, most of the measurements have been rounded to two decimal places. Unless otherwise indicated, measurements are made with all bounds-checking and debugging facilities enabled. The debugging facilities slow the run-time system by approximately 10 percent.

Static Measurements. The run-time system has a total of 3,424 lines of code, of which 28 percent are comments. This does not include any of the MEDUSA operating system, nor does it include the rather large file that contains macros and definitions used to interface BLISS-11 programs to MEDUSA. The compiler has 3,364 lines of code. Of this, 3,058 lines are in the code generator (15 percent are comments), and 306 lines define the syntax for the parser generator.

The code sizes for the run-time system are listed in Table 9-1. Typically, 16K bytes are allocated by each AP for the process-frame heap. About 15K bytes are available for compiled AMPL programs, limiting program size to about 350 lines of code. Since little attention was paid to conserving memory, there is considerable room for improvement in this area.

Send Operations. A program was written to measure the time it takes to send and accept a minimum-length AMPL message. The program was run on a single CP/AP pair. (The same code is executed regardless of the destination of a message.) The program is written so that all **sends** are to an empty port and all **accepts** are from a full port, so this represents the fastest time to deliver a message. It takes 96 seconds to send 10,000 messages; each message takes 9.6 ms., or about 1,370 equivalent LSI-11 instruction times.

Table 9-1 AMPL Run-Time System Code Size

Description of code	Size (bytes)
Memory manager	792
Garbage collector	2,646
Debugging and display routines	2,638
Input / output routines	1,408
BLISS-11 run-time routines	240
Other AP routines and initialization	2,830
Other CP routines and initialization	5,336
AMPL input / output modules	4,334
Total	20,224

Some simple analysis was performed to see where the time is spent. Table 9-2 provides a breakdown of the total time in terms of instruction counts. Instruction counts are used rather than instruction *times* to simplify the analysis. An average instruction time of 7 μs. is assumed. Actual instruction times are a function of the type of instruction, the source addressing mode, the destination addressing mode, and the location of all memory references (local, intracluster, or intercluster). Where MEDUSA operations are invoked, the equivalent number of LSI-11 instructions is taken from Chapter 5 or Chapter 7. The instruction counts indicate that the computation time is spread fairly uniformly among the logical subtasks required to send a message. There is little overlap between the CP and AP in this program. The measured time is therefore somewhat misleading because if two processes each sent a message, the CP and AP execution would overlap. Although the time to deliver a message would increase, the measured throughput would be significantly greater. Furthermore, if multiple CP / AP pairs were used, even more parallelism could be obtained.

The total estimated execution time is shorter than the measured time. Several factors have not been included in the estimate. First, nonlocal memory references have not been considered. The AMPL process frame is local to the AP. All references from the CP to the process frame take about 8.3 μs., so each reference costs approximately one extra instruction time. Second, our estimate of 7 μs. per instruction may be incorrect, particularly since our implementation uses a large amount of indirect addressing to locate variables and port structures in the process frame. Third, estimates are based on inspection of the code, not on instruction traces. In particular, it is assumed that when a message arrives at the CP, the CP is at a random point in its cycle of polling the high- and low-priority pipes. A complete cycle (two conditional receives) takes just over 1 ms.

Table 9-3 contains an alternate breakdown of the total execution time. As indicated, a significant fraction of the total execution time is consumed by microcoded Kmap operations, including MEDUSA message operations as well as ***Block Move***

Table 9-2 Time to **Send** and **Accept** an AMPL Message

	Instructions	Percent of total
AP instructions for **send** statement:		
Evaluate arguments and call send routine	9	1%
Build SEND message	123	11
Send MEDUSA message	71	6
Context swap	78	7
Subtotal	281	25
CP instructions to handle SEND message:		
Poll pipes for SEND message	136	12
SEND handler	205	18
send REPLY message	96	9
reschedule receiver	46	4
Subtotal	483	43
AP instructions for **accept** statement: *Subtotal*	141	12
CP instructions to handle REPLY message:		
Poll pipes for REPLY message	136	12
REPLY handler	41	4
Reschedule sending process	46	4
Subtotal	223	20
Total	1,128	100%
Total estimated time	7.9 ms.	

Table 9-3 Alternate Analysis of the Cost of an AMPL **Send** Statement

	Instructions	Percent of total
Save registers and coroutine call	96	9%
Debugging	103	9
Kmap operations	418	37
Other	511	45
Total	1,128	100%
Total estimated time	7.9 ms.	

operations to build the SEND message, to copy the AMPL message value into the destination message queue, and to copy from the queue into the variable specified by the **accept** statement.

Create Operations. Another experiment was run to create an arbitrary number of "empty" processes, which contain no parameters, variables, ports, or statements. This program was run on a single CP/AP pair; it creates and destroys 400 processes in 8 seconds, or 1 process in about 20 ms. The process descriptor table is large enough that no garbage collection is invoked during these CREATE operations. During this time, the AP is idle 87 percent of the time; the CP is never idle.

Tables 9-4 and 9-5 provide cost analyses similar to those given for **send** statements. Most of the time (58 percent) is taken by the CP to allocate, initialize, and schedule the new process frame. The CP executes an estimated 480 instructions, or 18 percent of the total, to fill the frame's stack area with zeros. This initialization is performed only to facilitate debugging and could be eliminated. A significant amount of time (24 percent) is spent allocating and deallocating process frames. This is an estimate based on the assumption that a memory block is split in two once during each allocation (the "buddy system" is used) and that two blocks are reunited once per deallocation. More efficient code probably could cut the memory-manager execution time in half.

Obtaining further improvements in the run-time system would be more difficult. The measured time for creation and termination of a process (20 ms.) is somewhat longer than the estimated time (18 ms.). The preceding section on **send** operations offers possible reasons. The process creation and termination time compares favorably with the message *send* and *accept* time. Note that these measurements include time to terminate a process and to reclaim the process frame. An estimated 14 ms. would be needed just to create a process.

Garbage Collection. A benchmark program was written to create an arbitrary number of processes. This program is identical to the process-creation benchmark except that the created processes each declare a port and a variable and attempt to execute an **accept** statement. No message is ever sent to the created processes, so all of them remain suspended. The process frames fill the available memory, causing the garbage collector to be invoked. On a single CP/AP pair, the execution time for 2,000 **create** operations is 53 seconds, or about 27 ms. per create. This includes time for garbage collection, which is in progress 61 percent of the time. A total of 1138 GCMARK messages are sent to accomplish 95 garbage-collection cycles. The AP is idle 84 percent of the time, and the CP is idle 8 percent of the time.

9.2.3. Conclusions

Building AMPL on an existing operating system provided some insight into what features an operating system should possess to support similar languages. Two approaches can be taken. In one approach, the operating system provides a basic set of abstractions without hiding the underlying physical machine from the user. An

Table 9-4 Cost of Process Creation and Termination

	Instructions	Percent of total
AP instructions for **create**:		
Evaluate arguments and call create routine	9	0.3%
Build CREATE message	76	2.9
Send MEDUSA message	71	2.7
Context swap	78	2.9
Subtotal	234	8.9
CP instructions to handle CREATE message:		
Poll pipes for CREATE message	136	5.2
Allocate process frame and table entry	362	13.7
Initialize process frame	917	34.7
Schedule new process	46	1.7
Send CREATE-REPLY message	71	2.7
Subtotal	1,532	58.0
CP instructions to handle CREATE-REPLY message:		
Poll pipes for CREATE-REPLY message	136	5.2
CREATE-REPLY handler	35	1.3
Reschedule creating process	46	1.7
Subtotal	217	8.2
AP instructions to run created process:		
Build UNCREATE message and context swap	134	5.1
Send UNCREATE message	71	2.7
Subtotal	205	7.7
CP instructions to handle UNCREATE message:		
Poll pipes for UNCREATE message	136	5.2
UNCREATE handler	25	1.0
Deallocate process frame	291	11.0
Subtotal	452	17.1
Total	2,640	100%
Total estimated time	18 ms.	

example of such an abstraction is a process. The power of the physical processor should not be restricted by the process abstraction [Parnas 72]. If the user decides to implement his own abstractions, he will have the power and efficiency of the physical machine at his disposal. The lack of interrupts in MEDUSA is an example of a violation of this principle. If interrupts were available, the CP and AP could have been implemented together on a single processor. Without interrupts, polling is necessary to detect when work is available for the CP. Whenever the low-level machine is hidden, there is a danger of providing the "wrong" abstraction.

A second approach is to provide everything the user might need. In the case of

Table 9-5 Alternate Analysis of the Cost of AMPL Process Creation and Termination

	Instructions	Percent of total
Save registers and coroutine calls	211	8%
Debugging	119	5
Kmap operations	583	22
Other	1,727	65
Total	2,640	100%
Total estimated time	18 ms.	

AMPL, this would require a number of abstractions not commonly found in operating systems. First, we would want to manipulate protected address spaces and processes separately to avoid the expense of allocating and deallocating a protected address space when processes are created and destroyed. (This problem was tackled by ECHOES, as reported below.) The process descriptors should be accessible to the user so that a garbage collector can be implemented. Alternatively, the system could provide its own garbage collection. Small, dynamically created processes are necessary for an efficient AMPL implementation. There must be a flexible message-passing system with an arbitrary number of ports per process and the ability to wait selectively on sets of ports. Finally, this all must be as efficient as can be achieved by writing customized run-time routines, or the language implementor probably will reimplement the necessary facilities to achieve better performance.

The problems of storage allocation have been avoided to a large extent in the design of AMPL. An efficient implementation of a production language would require further investigation of the storage allocation problem. In particular, there may be special cases where modules can be represented with little memory overhead. In AMPL, message queues are allocated statically within process frames. Dynamic allocation might conserve memory and allow the **accept** statement to be implemented by changing pointers rather than by copying the message. Storage allocation is another area where a reduced overhead may allow finer-grain processes and a higher degree of parallelism. Methods for efficiently allocating process frames are described in [Lampson 80]. Using microcode support, process and procedure frames can be allocated from a heap with a very small overhead.

As one of the first working implementations of a parallel language on a multiprocessor, AMPL was one of the first to face the problems of debugging in a parallel environment. Dannenberg offers these insights:

> Although we had no problems in constructing programs with correct synchronization, none of our larger test programs were free of errors. Debugging a parallel AMPL program is considerably more difficult than debugging a sequential one. Since no memory is shared, it is difficult for any one process to determine much about the global state of the program. For example, when debugging the PDE program, we wanted to print snapshots of the grid to

help locate a bug. Unfortunately, the grid is spread across several slaves, so the cooperation of several processes would be required to access it. New ports would have to be added to each slave to request access to the grid. Another problem is caused by the parallelism. For example, it is sometimes helpful to trace program execution by printing values of variables at run time. Because of parallelism in AMPL, if several processes begin printing at the same time, output become hopelessly mixed up. Ordinarily, a collection of related debugging information must be packaged into a single message and sent to a printer process which formats and prints the data in a readable fashion. Obviously, more debugging aids are necessary. A symbolic debugger which could access all processes and monitor all messages would be a great help. There is also a need for methodologies which enable programmers to transform sequential programs into parallel ones. Most of the test programs we wrote are essentially sequential programs with a few simple modifications to subdivide and distribute the program. It would be nice to debug the sequential parts of the programs on sequential machines and to be able to take existing sequential algorithms and transform them to parallel programs.[2]

9.3. ECHOES

ECHOES [Jones and Gehringer 80] was an experiment by Mike Kazar in designing operating-system primitives for executing procedure calls and forks rapidly. Although somewhat earlier (1979) than AMPL, it was motivated by the observation, underlined in the AMPL measurements reported above, that efficient multiprocessing requires fast process creation. Its approach was to write microcode to perform protected procedure calls quickly and then to provide a ***Fork*** operation (for process creation) as an extension to the ***Call*** mechanism. In effect, it provided the addressing structure for a *procedure-oriented* operating system, whereas the AMPL run-time system is completely process oriented (as user-defined procedures do not even exist).

ECHOES is not a complete operating system; it contains little code for resource management and almost no utilities. Rather, it is a set of microcoded Kmap operations that can be invoked from BLISS-11 programs. A BLISS program executes as it would on a normal LSI-11 until it needs to call a new module, create a new process, or gain addressability for some additional data, at which time it invokes the ECHOES microcode, which was written in MUMBLE.

It has been observed [Lauer and Needham 78, Keedy 79] that procedure-oriented and process-oriented operating systems are at opposite poles of the operating-system design spectrum. Procedure-oriented systems tend to use procedure calls to communicate with the operating system and other modules, and thus they require well-developed within-process protection mechanisms. Process-oriented systems tend to use message passing to communicate with the operating system and processes instantiated from other modules; simple interprocess protection suffices, but a well-developed message system is a necessity. Among the Cm* software environments, AMPL is totally process oriented; STAROS and MEDUSA are largely process oriented, STAROS a bit more so than MEDUSA; and ECHOES is largely procedure oriented.

In another respect, AMPL and ECHOES exhibit nearly antithetical approaches to the

[2] [Dannenberg 81], pp. 68–69.

issue of shared memory. AMPL processes share no memory at all, except possibly their object code. ECHOES is designed to encourage the use of shared memory. The ECHOES run-time environment consists of a set of reusable *address spaces*. These address spaces are large enough (2^{28} bytes) to accommodate programs of substantial size. They are addressed through 256-item descriptor segments, where all but 10 descriptors are shared by multiple processes. The philosophy behind this approach is that the cost of creating an address space is high and should be paid only once per module. Except for the objects pointed to by those descriptors, the address space is sharable by multiple processes. It survives the process or processes that execute in it, and after reinitialization of those 10 descriptors, it is available to other processes that execute code from the same module.

To see the importance of shared address spaces in fast process creation, consider the AMPL data from Table 9-4. Allocating a process frame consumes 13.7 percent of the time it takes to create and then destroy a process, and initializing the process frame takes 34.7 percent of the time. Creating an address space, then, takes up roughly half the time for a process creation and termination. Counting the 11 percent of the time devoted to deallocating a process frame, address-space manipulations account for about 60 percent of process creation and termination. These numbers take on added impact when it is noted that five-sixths of the remaining time (34 percent overall) is devoted to message operations, which could be optimized in a custom implementation. Clearly, sharing address spaces offers great potential for improving the performance of process creation.

9.3.1. Addressing Structure

An ECHOES process can address 256 segments (Table 9-6) at any one time. The descriptors for these 256 segments reside in segment 7, which is called the *descriptor segment*. The majority of these segments, numbered 16 through 255, depend only on the address space in which the process is executing. There is only one address space per module; all functions of that module—regardless of which process invoked them—execute within that address space and can access segments 16 through 255.

Table 9-6 The Segments in an ECHOES Address Space

Segment number	Type	How shared
0–3	(unused)	
4	Gate segment	Per address space
5	Protected call stack	Per process
6	Process stack	Per process
7	Descriptor segment	Per address space
8–15	Argument segments	Per call
16–255	Code and data	Per address space

Processes executing the same code must not be able to access each other's process stacks, so processes executing within the same address space must not share stack segments. In fact, an ECHOES process has two stack segments. There is a different copy of both these segments for each process executing within an address space. Segment 6 is an ordinary process stack and is used in the same fashion as any other PDP-11 stack. Segment 5 is a *protected call stack*, which is accessible only to the ECHOES microcode. The microcode uses it to maintain return addresses for intermodule calls. A new frame is pushed on this stack each time an ECHOES ***Call*** operation occurs.

Of the remaining segments, numbers 8 through 15 are *argument segments*. After a procedure call has taken place, segment *8 + i* contains a descriptor for the *i*th argument of the call. (The limitation of eight arguments per call has not proven to be a problem.) These slots can be occupied by "refined" descriptors, whose base and length fields grant access to only a *portion* of the segment being passed. Subsegments as small as one word in length can be passed in this fashion. Argument descriptors also may have fewer rights than the caller possesses; for example, a procedure with read/write access to a particular segment might pass it as a read-only argument. Each time a different call takes place, these segments can, in principle, be different. That is, if the same process makes two calls to a procedure of the same module, the arguments may be different each time.

Segment 4 is known as the *gate segment*, used to mediate protected calls, and is analogous to gate segments in Multics [Organick 72]. Segments 0 through 3 are unused and generate faults if indirected through. This helps catch uninitialized-pointer errors.

When a procedure call occurs, a check is made to see whether an address space already exists for the module that is the target of the call. If not, an address space must be created, requiring building the descriptor list, allocating memory for each segment, and initializing the segments as required. When the address space exists, the following actions are performed:

- A new frame is placed on the protected call stack.
- Arguments are transmitted, possibly with refinements.
- A protected call takes place, which usually[3] causes the process to begin executing in a new address space. Thus, it gains access to a whole new set of segments (segments 16 through 255 of the called module) and loses access to another set of segments (number 16 through 255 of the calling module).

A ***Fork*** (process-creation) operation is not very different from a procedure call. If there is a free processor available, a new process will be created to execute code from the destination module. Otherwise, the forking process will be suspended while the forked process executes. Creating a new process is quite simple. The new

[3] Protected calls to the current module are also permitted. In this case, the address space of the called module is the same as the address space of the caller.

process inherits an address space from the called module. At system initialization time, ECHOES *preallocates* a number of segments that can be used for process stacks and protected call stacks. [4] These segments are placed on a free list. The system merely gives two segments from this list to each new process that is created.

If enough processes are created, of course, it is possible to run out of preallocated stack segments. It would be easy to remedy this problem without slowing down process creation appreciably. When the system noticed that the supply of preallocated segments was running low, it could assign an unused processor to create more. This processor would run slowly, since it would need to make many mapped references, and would add to memory contention, thus slowing the other processors somewhat; but these effects would be minor compared to waiting for segments to be allocated at process-creation time.

9.3.2. Measurements

The ***Call***, ***Return***, and ***Fork*** operations were measured in real time. It was immediately noticed that the timings were unexpectedly large. This problem was due to scheduler overhead; the simple scheduler was interrupting processes 60 times per second, and the resulting context swaps caused the processor to be effectively slowed down to 60 percent of its nominal speed. This was compensated for by multiplying all measurements by 0.6. The numbers presented below have the scheduler overhead factored out in this way.

Another factor that decreased the observed execution speed was the inability of the Kmap to set the program counter (PC) on an LSI-11. This is a general characteristic of Cm*; the Kmap cannot directly access any LSI-11 processor registers. Yet the Kmap must be able to set the PC during the execution of a protected procedure call. ECHOES arranged this in the following way: The Kmap forces the LSI-11 to take a nonexistent memory reference fault—called an NXM—which the Kmap *can* do by writing a special Slocal register. The LSI-11 then checks the reason for the interrupt and finds that it was to reset the PC. The NXM handler then returns to the proper place. This method of resetting the PC takes about 193 μs.

The time to perform a ***Call*** and a ***Return*** (with a null procedure body) is 846 μs., of which 386 μs. can be written off to the overhead of setting the PC. This leaves 460 μs., the equivalent of about 66 average LSI-11 instructions, or twenty-three 16-bit ECHOES mapped memory references.[5] In fact, a ***Call*** requires the following mapped references:

[4] This identical mechanism was later used on the Intel 432 [Organick 83] to save time in protected procedure calls.

[5] Note that ECHOES mapped references (20 μs.) are more expensive than intracluster references in STAROS or MEDUSA. This is because they must handle 20-bit offsets rather than the 16-bit offsets for which the Kmap was designed.

- The ***Call*** instruction takes three memory references to fetch and another two to locate the branch address.
- Six 16-bit memory references push data onto the ECHOES protective call stack: a 32-bit return address, a 16-bit "domain field," the number of arguments (in a 16-bit field), and the stack limit (a 32-bit quantity).

The total memory-reference time is thus about 220 μs. The ***Return*** operation should take two 16-bit mapped memory references *less* than the ***Call***, since it pops the same information back off the stack and restores the processor state from it but does not fetch a branch address (which it gets instead from the call stack). Thus the amount of memory-reference time for a ***Return*** is about 180 μs., and the total for ***Call*** plus ***Return*** amounts to 400 μs. In addition, the Kmap microcode to perform a ***Call*** and ***Return*** uses 382 microcycles, which at 157 ns. per microcycle comes to 59.97 μs., yielding a total that is very close to the observed 460 μs.

These times are for a procedure call with no arguments. Each argument takes somewhat more than three memory-reference times, or 60 μs. extra, to pass. The argument segment registers (segments 8 through 15) are cached in the Kmap. To pass an argument, the descriptor for the argument segment is copied to the Kmap (requiring three mapped reads and a comparatively insignificant amount of Kmap microcycles), and refined if necessary. It remains cached in the Kmap unless the Kmap's cache overflows, in which case it is written to memory. Note that cache overflow occurs only in the case of deeply nested calls with many arguments. The Kmap cache, used in this fashion, speeds up procedure calls in the same way as the multiple register sets of the RISC I [Patterson and Sequin 81], except that the ECHOES cache holds only addressing information, not data.

A ***Fork*** that fails to create a new process (because no processor is free) takes 172 μs. more than a ***Call***. Although it makes six fewer memory references, these savings are more than offset by more complex microcode plus the cost of a system-supplied synchronization operation to test for completion of the *fork*ed process. If the ***Fork*** does create a new process, it takes 6 μs. less than an unsuccessful ***Fork***, the difference apparently being due to overlapped execution of the two processes. As shown in Table 9-7, adding further processors increases the execution time, undoubtedly because of the extra contention.

9.4. Summary

AMPL and ECHOES reflect two different philosophies and a number of trade-offs. AMPL is process based, while ECHOES is procedure based. ECHOES endeavors to provide fast procedure calls, but AMPL disallows procedures. AMPL prohibits shared memory; ECHOES emphasizes it. AMPL runs on top of an operating system, while ECHOES does not even provide all of an operating system's functionality. AMPL has concentrated on issues of language design (it provides a high-level facility for interprocess communication) whereas ECHOES makes do with primitive ***Fork*** and ***Join*** operations. ECHOES, much more than AMPL, has concentrated on optimizing its run-time system.

Table 9-7 Timings of Echoes ***Call***, ***Fork***, and ***Return***

Operation	Microseconds
Standard procedure ***Call*** and ***Return***	846
Failing ***Fork*** (no free processor), ***Return***, and re-***Synchronize***	1,018
Successful ***Fork***, ***Return***, and re-***Synchronize*** (2-Cm system)	1,012
Successful ***Fork***, ***Return***, and re-***Synchronize*** (3-Cm system)	1,278
Successful ***Fork***, ***Return***, and re-***Synchronize*** (4-Cm system)	1,573

The performance of the two systems can be compared in one important way: the speed with which processes can be created and deleted. Echoes, which has been optimized for this purpose, seems to be nearly 18 times as fast as Ampl. In practical terms, the difference would not be quite this great. Ampl can distribute processes widely throughout Cm* with little or no performance penalty. Echoes processes would suffer severe degradation if they ran remote from major portions of their address space. For performance reasons, address spaces might have to be partially or wholly duplicated in several Cm's, obviating some of the advantage of sharing. Furthermore, if a process began executing (due to a protected call) in an address space on a remote Cm, its stack would have to be moved to that Cm to obtain good performance. This would impose an additional overhead for which the measurements do not account. These problems are not insoluble, but much research would be needed to determine effective strategies for dealing with them.

These experiments demonstrate the flexibility of the Cm* architecture, especially of its programmable distributed switch. It has been shown that even with limited effort, it is possible to gain an order of magnitude improvement in the performance of a critical function (here, process creation). The efforts of other researchers have proven that other functions (message communication, for example) can be similarly speeded up. The fact that Cm* has been effectively exploited in such different ways confirms its utility as a flexible testbed. A testbed, however, is only as good as the quality of the experimentation support. The next chapter describes the prototype experimentation environment of Cm*.

Acknowledgment. The description of the Ampl run-time system has been adapted from [Dannenberg 81].

10. Integrated Instrumentation Environment

Commercial operating systems are designed for the graceful support of applications software. In an experimental operating system, the applications consist largely of the experiments that are run. It is not the results produced by the applications that are of primary interest, but rather the performance of the computer system while producing them.

Traditional operating systems provide little help to the researcher attempting to develop experiments or classify and interpret their results. Consequently, it is up to the researcher to write benchmarks that differ only in detail, to run them, and to develop a uniform scheme for keeping track of the results. Data analysis programs are usually written later to translate the data into a readable form. The work involved is considerable, and much of it is clerical in nature, only tangentially related to scientific inquiry. The Cm* project thus developed an instrumentation environment that runs on top of both operating systems to provide assistance in performing these chores.

10.1. Functionality of the IIE

An *Integrated Instrumentation Environment* (IIE) consists of a set of tools that cooperate closely with each other and present the user with a single uniform interface to assist and partially automate the process of experimentation. The general objective of an experiment is to inquire about performance, reliability, or any of a number of interesting properties of a computation. In the context of a computer system, an experiment is the execution of an instrumented program in a controlled environment that allows measurement, collection, and analysis. An experiment may involve multiple executions of the instrumented program with different input parameters or within different environments.

The IIE supports the notion of an *experiment schema* as the high-level unit of experimentation management. Each schema specifies a related collection of *runs*—that is, executions of an instrumented program. Intuitively, a schema can be seen as a parameterized experiment script that describes the experimentation process. A schema specifies the instrumented program, the monitoring directives, the specifications of the run-time environment, and the input parameters for each run.

The result of executing a schema is captured in a *schema instance*, containing measurements, values of schema parameters, and environmental information. This is a data structure representing the unit of management for the experiment results. Schema instances are archived in a database for later analysis.

By using the generic notions of schema and schema instance, the experimentation process can be expressed as in Figure 10-1.

Each phase of the experimentation process will be discussed in detail in the following sections.

Figure 10-1 Experimentation Process in the IIE

```
Schema = design(experiment)
while (not end-of-experiment) do
    begin
        execute(Schema)
        create(Schema Instances)
    end
analyze(Schema Instances)
```

An IIE requires software to support the several phases of experimentation, including:

- Translation of collections of user-defined modules and predefined synthetic actions into instrumented parallel programs.
- Creation of the schema by merging the instrumented parallel stimulus, the monitoring directives, and the environment information.
- Schema interpretation and run-time control.
- Creation of schema instances.
- Analysis of schema instances.

To illustrate further the experimentation process described above, we will follow an example through in some detail. This example shows how the IIE, at each stage, interacts with the user, performs the required actions, and generates its outputs. The example stimulus, simply called a multiprocessor experiment, or MPX, involves a single *initiator* and multiple *servers* communicating through a shared buffer or mailbox. The initiator repeatedly sends requests through the buffer to one or more servers, which operate on those requests concurrently. When the buffer is empty, the servers wait for further requests; when the buffer is full, the initiator waits for a request to be removed by a server. The servers perform identical functions, so a request can be satisfied by any server. Additionally, the servers communicate with each other via shared memory.

The goals of the proposed experiment are to investigate the interaction between the request rate (expressed as the average number of requests per unit time) and the number of servers, and to investigate the effect of the request rate and the number of servers on the average buffer queue length and the average waiting time in the buffer.

Two interesting steady-state behaviors have different average queue lengths and service rates. In the first case, the request rate exceeds the aggregate processing rate of the servers, so the buffer will always be full. In the other case, the buffer will never contain more than one request. The aggregate service rate will be approximately constant, yet radically different, in both cases. This analysis assumes a constant individual service rate by independent servers. In Cm*, however, accessing shared data perturbs the performance of both the servers and the buffer insert / remove operations in subtle ways, greatly complicating analytical modeling of the queue length and waiting time. As was shown above, the boundary between the two cases is quite distinct if contention is ignored. The experiment will investigate the boundary in the presence of contention.

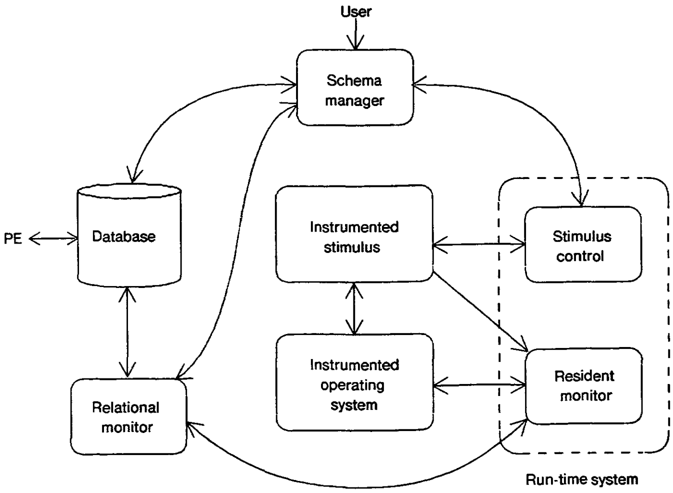
To summarize the approach, experiments are described as schemata, and the result of executing a schema is a schema instance. The primary functions of the IIE are the creation of schemata, schema management execution, and control of schemata, along with the creation, management, and analysis of schema instances. The next section presents the design of an IIE supporting these functions.

10.2. Design of the IIE

The IIE contains several components: a schema manager, a run-time environment, an instrumented stimulus and operating system, a database, and a monitor (see Figure 10-2). The monitor consists of a resident monitor, which gathers the data from the system under test, and a relational monitor, which aggregates and correlates the data into a high-level form. The user interacts directly with the schema manager, which communicates with the run-time environment and the monitor, which in turn interacts with the instrumented program (the *stimulus*) and the database. The IIE interacts with the programming environment (PE) through the database.

The schema manager is responsible for supporting the schema and schema-instance abstractions. The monitor initializes the schema instance with information

Figure 10-2 IIE Components



The arcs indicate transfer of data or control.

specifying this environment (including details of the hardware configuration sion of the operating system, support software, and stimulus) and the valu parameters that will remain constant for this execution of the schema. The manager then cycles through the runs as indicated in the schema, i parameters that vary on a per-run basis, starting the stimulus, and colle monitoring data. Finally, data concerning the runs as a whole is collected puted and is stored in the schema instance for later study. Note that not a components should necessarily reside and execute in the same machine. In Cm* IIE implementation spans several computer systems. The run-time sy the stimulus are resident in Cm*, whereas the schema manager, the datab the relational monitor are remotely located in a VAX 11/780. The two systems are connected by an Ethernet link.

One motivation for partitioning the components of the IIE into a run-time ment and a remote environment is that only the run-time environment is co to any particular hardware or software configuration. Care has been taken the remote components as system independent as possible.

Two preliminary implementations were built for the run-time environmen different operating systems, while only one implementation of the rem ponents is necessary.

The stimulus-controller component provides a well-defined interface to t mented stimulus. The functions it supports include modifying parameters stimulus, both before and during the run, generating initial control even stimulus, reporting errors back to the schema manager, and controlling t Similarly, the resident monitor provides a uniform interface for the relationa The resident monitor is responsible for enabling and disabling *sensors* and ing the information back to the relational monitor in a format convenient f processing. The sensors are embedded in the stimulus, in the stimulus co the operating system, and in the resident monitor itself. The relational mo trols the resident monitor and computes derived information, which is then a schema instance in the database.

The database serves an important role in the IIE because the informa tained in the database is the end result of the entire experimentation Additionally, the interaction between the IIE and the PE occurs via the da having one environment create objects in the database for the other envir use. For instance, schemata are initially created in the PE, to be interpret schema manager. Schema instances, created by the IIE, are managed version-control facilities of the PE. By using a common database, it is p use the functionality provided by the PE. This approach allows the design IIE to concentrate on those operations unique to experiment management.

10.3. The Instrumented Stimulus: Representation and Specification

The stimulus is an arbitrary set of processes executing in parallel. The stim may incorporate sensors; in addition, sensors reside in the operating syst the resident monitor. Tools were developed to aid in the rapid develop stimulus. One of them is a workload generator. A user specifies the behavior of his or her parallel program in a special high-level behavior-description language, the *B-language*. This behavior is specified as a directed dataflow graph, similar to a complex bigraph [Cerf 72, Gostelow 71]. This graph is known as a *task graph*. Its nodes represent subtasks, or processes, that execute in parallel with other subtasks. Each subtask is composed of *actions* (parameterized program fragments that may be predefined or user defined) that are repeated at certain rates. Associated with each arc of the graph is a buffer that may hold data variables or control tokens flowing from one subtask to another. Each subtask has an associated control tuple (*i*, *o*), where *i* corresponds to the in-firing rule for the subtask and *o* corresponds to the out-firing rule. This set of firing rules characterizes the precedence relationship between the subtasks of the graph. A B-language program is compiled into an executable version as illustrated in Figure 10-3. This section gives a brief overview of the B-language; a more detailed discussion can be found in [Singh 81].

The B-language thus represents the interaction of parallel processes via the graph model of computation. A typical example is shown in Figure 10-4. Subtask *A1* is fired by the arrival of a token in buffer *B1*, which corresponds to the entry arc of the graph. Upon completion, subtask *A1* fires either subtask *A2* or *A3* by placing control tokens in buffer *B2* or *B3*, respectively. There is a certain probability associated with the OR-output logic of subtask *A1* (designated by the +). Finally, subtask *A4* fires if it receives a token from either *A2* or *A3*. Upon completion, it places a token in buffer *B6*, which corresponds to the exit arc of the graph and represents the end of a single execution of the parallel synthetic program. The B-language subtask declarations for this example are as follows:

subtask *A1* {**inlogic**: *B1*; **outlogic**: %40 (*B2*) **or** %60 (*B3*)}

subtask *A2* {**inlogic**: *B2*; **outlogic**: *B4*}

subtask *A3* {**inlogic**: *B3*; **outlogic**: *B5*}

subtask *A4* {**inlogic**: *B4* **or** *B5*; **outlogic**: *B6*}

Figure 10-3 Steps in Stimulus Generation

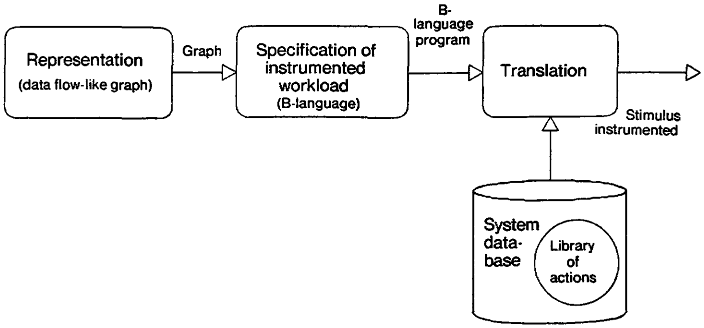
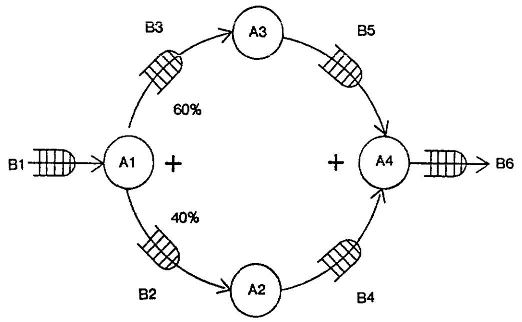


Figure 10-4 A Parallel Synthetic Program—Graph Representation



Notice that buffers *B1* to *B6* correspond to the arcs of the parallel synthetic program. The delimiter % is used to specify the branching probabilities for the arcs of an OR-output.

The specification of parallel synthetic programs in the B-language is based on the object model supported by both operating systems on Cm*. The objects represented directly in the B-language include the following:

The task-force object. The task force abstraction, a collection of processes that cooperate to achieve a single logical task, is represented by a set of *subtasks.*

The subtask object. This is the sequential computation unit that cooperates with similar user-defined objects to compute the overall stipulated multiprocess task.

The buffer object. The buffer object is a conventional queue of messages and is used by the subtasks to communicate with each other.

The semaphore object. Semaphores synchronize requests for shared resources.

The file object. Files represent a sequence of bytes.

The shared data object. Variables specified in the shared data object are globally shared by all the subtasks of the task force. This allows communication of data and control through shared memory.

The table object. Tables implement functions varying with time.

Within a subtask, the basic building block is an action. To capture the cyclic nature of synthetic workloads, an action a_j itself is described by an action-repetition tuple (specified as $<a_i, r_i>$). This tuple specifies that the action a_i is repeated sequentially r_i times, constituting action a_j. An action may be arbitrarily complex and may be further composed of action-repetition tuples. Also, both the *a* and the *r* can be parameterized. Other control constructs within a subtask include composition and conditional and probabilistic branching.

The library of actions consists of a collection of predefined and user-defined program fragments programmed in the systems programming language and stored as part of the system database. Examples of predefined actions include sending or receiving messages via a buffer, inputting or outputting to a file, referencing local memory, blocking on a semaphore, and accessing a shared resource. The user gains flexibility by being able to include his own special program fragment among the actions in the library. An example of a user-programmed action is the code for a disk process in a database application running on a specific multiprocessor. Hence the library of actions is specific for a particular multiprocessor system. The B-language should be viewed as a portable framework into which system-specific actions are inserted from a library of actions.

Special control constructs are included in the B-language so that the schema manager may control the user's workload at run time as specified in the schema. The control commands initiated by the schema manager are executed by the stimulus controller component of the run-time system. The **vary** construct in the language permits the stimulus controller to vary parameters on a per-run basis. The language also allows one to specify that the parameters are to vary in real time. This is accomplished by binding a real-time function to a run-time variable on a per-run basis. The real-time function is defined by a table object and an associated interval of time. The stimulus controller forces the run-time variable to take on successive values from the table during successive time intervals.

Using the **MsgEvent** construct, the language permits the stimulus controller to initiate variable time-driven events in the stimulus on a per-run basis. This construct requests the stimulus controller to deliver messages to a buffer with intermessage time periods, as specified by successive entries of a table. The stimulus controller can associate a different table object, or a constant time period, with the **MsgEvent** variable on a per-run basis.

To allow measurement of the generated workload, a special **sensor** construct permits a user to embed sensors into his program. Sensors allow specified information as well as a time stamp to be sent to the monitor as event records. In addition to user-defined sensors, the B-language program has some built-in sensors. For example, the start time and end time for each execution of a subtask are automatically recorded in the event record. Furthermore, instrumentation available in the operating system and the IIE run-time system allows the schema manager to access information not explicitly specified in the B-language program. An example is information regarding the interaction of the stimulus and the operating system.

The B-language translator constructs special data structures, allowing the stimulus controller to exercise external control over the experiment as specified in the B-language program. The translator also generates sensor descriptions (see Section 10.6) for all programmed and predefined sensors in the B-language program. These descriptions are used by the relational monitor to sort out event records flowing from the resident monitor.

As an example, consider the B-language program (Figure 10-5) for the single-requester, multiple-server experiment discussed in Section 10.2. The task force consists of an array of five identical server subtasks that wait on the *RequestBuffer*

Figure 10-5 B-Language Program for the MPX

```
TaskForce MPXperiment;
buffer
            RequestBuffer{size: 512};
semaphore
            GDSemaphore{initial: 1};
shared
            GlobalData[512];
vary
            RequestPeriod;
MsgEvent
            RequestService = RequestBuffer @ RequestPeriod;
sensor
            StartGlobalPhase;

subtask Servers[1..5]
                  {inlogic: RequestBuffer}
vary
            SharedDataAccess;
begin
            <$DoLocalWork: 10>,
             StartGlobalPhase,
            <$AccessSharedData (GDSemaphore, GlobalData): SharedDataAccess>
end
```

for queued service requests. The *RequestBuffer* is associated with the message-event generator via the **MsgEvent** construct. This allows an experimenter to vary the request rate by changing the time (*RequestPeriod*) between successive firing of servers on a per-run basis. The **begin** and **end** constructs mark the service loop of each subtask that is executed each time its in-firing rule is satisfied. In this example, each server does ten units' worth of work local to its processor and then does some variable number of accesses to global data, which is arbitrated by a semaphore. A sensor, *StartGlobalPhase*, is embedded in each subtask and sends an event record to demarcate the transition from local work to global work. Built-in sensors record the beginning and ending of each subtask and the firing of request tokens by the message-event generator. The variable parameters of the experiment are the number of active servers, the request rate, and the amount of global work done by each server. This program will be specified in the schema as the stimulus for the MPX.

10.4. Relational Monitor

In the IIE, each time the experiment schema is interpreted and the stimulus executed one or more times, various monitoring information is collected and stored in the database in a schema instance.

The model of the monitoring data adopted in the IIE is a variant of the *relational model* used in conventional relational databases [Ullman 80]. Information is recorded as a collection of two-dimensional tables, called *relations*. Each row, called a *tuple*, records a particular relationship between entities named in the columns, called *domains*, of the tuple. For example, the relation *Running* (*Process, Processor*), with two domains, may contain the tuple (*MyProcess, ProcessorA*) indicating that the process called *MyProcess* is running on the processor called *ProcessorA*. Relations used in monitoring are temporal in that each tuple records relationships that are true at an instance of time or over some interval of time. A relation involving instances of time is called an *event relation*; each tuple records the occurrence of a particular event. A *period relation* records a relationship that exists for an interval of time. Periods are delimited by events; each tuple (period) in the *Running* relation is associated with two other event tuples, one in the *Start* relation and one in the *Stop* relation. Time is included in an implicit domain manipulated by the monitor.

The *Running* relation is an example of a *primitive relation* because the information it contains is a direct translation of a set of recorded events. Primitive relations may be divided into three categories: operating system, stimulus control, and user defined. The first category is concerned with information involving the operations and data structures supported by the operating system. The *Running* relation is in this category. The second category involves the actions performed by the run-time portion of the IIE. Examples of event relations from the MPX include:

RequestService(*TokenID*). The sending of a **MsgEvent** token to the *RequestBuffer*; the *TokenID* identifies the token.

ServersStart(*Index, TokenID*). The in-firing of a sensor's subtask; the *Index* identifies the server; the *TokenID* identifies the token causing the firing.

ServersEnd(*Index, TokenID*). The out-firing of a sensor's subtask.

The one user-defined primitive relation specified in the MPX, *StartGlobalPhase*, is also an event relation and contains only the implicit time domain. This relation was declared as a **sensor** in the B-language program for the MPX (see Figure 10-5) and records the time at which the server subtask finished its local work and started the shared data access.

Given a collection of primitive relations, new relations can be defined as a result of operations performed on existing relations. These *derived relations* are specified using a relational query language. The query language used in the IIE is a version of Quel [Stonebraker 76] augmented with additional temporal constructs; it is discussed elsewhere [Snodgrass 82]. Figure 10-6 illustrates the definition of the derived relations *AverageQLength* and *ServiceRate* used in the MPX. The former relation has one domain, *AvQL*, with the tuples specifying this value for the various time intervals. Similarly, the *ServiceRate* relation will have one domain, *SRate*, containing values varying over time. These queries will be referred to by the schema for the MPX and will specify both the primitive relations to be monitored and the calculations to be performed on the data in the event records.

Figure 10-6 Queries for the MPX

```
range of R is RequestService              ; references to R will indicate the
                                          ; RequestService relation
range of S is StartServers
range of Sp is StopServers

define WaitingInQueue (R.TokenID)          ; one domain, the request's TokenID
          where R.TokenID = S.TokenID      ; request is being serviced by a server
             start R                      ; the waiting begins when the request
             stop S                       ; is made, and ends when the server starts
range of W is WaitingInQueue

define QLength(L = Count(W))              ; count the number of outstanding
                                          ; requests in the buffer
range of Q is QLength

define AverageQLength(AvQL = Average(Q))  ; instantaneous average

define TotalWaiting(W.TokenID)
             where Sp.TokenID = W.TokenID
             start W                      ; total waiting time begins when the request
             stop SP                      ; was made, and ends when the server stops
range of TW is TotalWaiting

define ServiceRate(SRate = 1 / Average(Duration(TW)))
```

10.5. Stimulus Controller

The stimulus controller component of the run-time system is a set of utilities that permit control of the stimulus as specified in the schema. While the schema manager provides experiment *management* through the management of the schema abstraction, the stimulus controller provides low-level experiment *control* through the management of a single run. The motivation was to separate the low-level control functions from the experiment-management functions so that different management strategies could be carried out using common control primitives. The functions exported by the stimulus controller are therefore geared toward the initialization and execution of a single run.

One responsibility of the stimulus controller is to ensure the repeatable behavior of a run by eliminating side effects from one run that might perturb the next run. An example of a side effect is the presence of tokens left over in the edges (buffers) as a result of the previous run. The stimulus controller ensures that all data structures are in a well-defined state at the beginning of a run. For example, buffers are emptied and all semaphores are initialized as specified in the B-language program.

The stimulus controller is also responsible for the variation of parameters both on a per-run basis and in real time during a run. The variation of parameters on a per-run basis involves the **vary** parameters of the B-language program (see Section 10.3) and the variation of the graph-structure representation of the program. A typical modification of the graph structure involves changing the number of active subtasks for a specific run. This is particularly useful in real-time experimentation, where one wants to determine the number of subtasks necessary to meet real-time constraints. The variation of parameters in real time during a run involves the variation of the run-time variables of the B-language program, according to some function of time expressed as a table object and an associated interval of time.

The stimulus controller must provide a well-defined mechanism to start the run. In the graphic representation of the program, this corresponds to firing the entry node—that is, placing a token on the entry arc of the graph. To start a run, the stimulus controller delivers a specified number of control tokens into a system-defined buffer, called the *IgnitionBuffer*, which corresponds to the entry arc of the dataflow graph. The user may use this source of tokens to start any desired subtask by specifying the *IgnitionBuffer* appropriately in the in-firing rule of that subtask. Similarly, to detect the end of a run the stimulus controller watches a system-defined *TerminationBuffer* for a specified number of tokens.

The stimulus controller has four major subcomponents. The first subcomponent executes basic control functions, including *initialize*, to initialize the instrumented program before each run; *fire*, to fire a specified number of tokens into the *IgnitionBuffer*; *vary*, to permit the variation of **vary** parameters on a per-run basis; *display*, to display the value of a **vary** parameter; *enable / disable*, to enable or disable subtasks on a per-run basis; and *status*, to return the status of the program. Observe that functions such as *display* and *status* are interactive in nature and can be used during the interactive creation of a schema (see Section 10.7).

The second subcomponent is a message-event generator, which delivers token messages to prespecified buffers according to pre-specified functions of time. Control functions performed by this subcomponent include *start generator*, to start the message-event generator for a particular run; *stop generator*; and *set message event*, to allow the association of either a table object or a constant with a buffer.

The third subcomponent is a run-time variable driver, which ensures that all run-time variables vary in real time, as specified by their associated table objects and time intervals. The main control function of this module is to allow the association of different table objects and time intervals with a run-time variable on a per-run basis.

The fourth subcomponent is a clock module, which permits access to a set of clocks distributed over the system. This module is used by the message-event generator, the sensors, and the run-time variable driver.

Additional functionality in the instrumented program may be added by augmenting the stimulus controller. For example, a set of components used for experimentation related to reliability has been designed and partially implemented. This includes software-implemented voters and accelerated fault-insertion and configuration-control modules.

10.6. The Resident Monitor

The monitoring information is collected as *event records*, which are generated by *sensors* in the instrumented stimulus, the run-time system, the operating system, or the hardware. Each event record contains an indication of the operation being monitored, the name of the component performing the operation, and the name of the object on which the operation is being performed. The event record may optionally contain a time stamp and other information germane to the event. For instance, a sensor located in a file-system process might generate event records for file reads. In this case, the event record would include the name of this process, the name of the file being read, an indication that this is a file-read event, the time stamp, and perhaps the block number being read.

Highly selective filtering of the event records is necessary to constrain their flow into the monitor. Enabling and filtering directives are encapsulated in data structures called *receptacles*, which are associated with either active components, such as a file-system process, or passive objects, such as a file. Each receptacle contains event-enable switches and a buffer for temporarily storing event records. The resident monitor (and thus, indirectly, the relational monitor) has the ability to enable switches in each receptacle. The flexibility in associating receptacles with either processes or objects provides a mechanism for filtering the event records. For example, if the receptacle was associated with the file, and the file-read event was enabled, event records for all file reads performed on the file would be written into the receptacle. Alternatively, if the receptacle was associated with a file-system process, event records for all file reads performed by the process on any file would be written into the receptacle.

A task force is instrumented by specifying the sensors, events, and object types in a file called a *sensor description*. The operating system and stimulus controller, being task forces themselves, also are associated with sensor descriptions. A sensor description is generated automatically when a B-language program is processed. Users can also write their own sensor descriptions. Figure 10-7 illustrates the sensor description generated from the B-language program for the MPX given in Figure 10-5. This description includes a sensor-process definition for each subtask and for the stimulus controller. It also includes events for the start and end of each subtask's execution and the start of each run. Another program takes the sensor description and produces optimized code for each software-implemented sensor, based on the specifications in the sensor description. Sensor descriptions thus allow users to specify their own sensors, which will utilize the same mechanisms for event record and generation as the sensors embedded in the run-time and operating systems.

It is important to note that the user never needs to be concerned about receptacles or event records. Instead, the IIE (through the monitor component) presents the view of a database composed of temporal relations. New relations can be derived using the query language (identified in Section 10.4). As a result of executing a query, the appropriate operations (locating and enabling receptacles, processing event records, and generating the schema instances) are performed automatically.

Figure 10-7 Sensor Description for the MPX

```
(TaskForce (name MPExperiment)              ; Standard prelude
    ... )
(SensorProcess (name StimulusControl)
    ... )
(Event (name PerRun)
       (Domains (Domain (name RunNumber)
                        (type Integer))
                (Domain (name RequestPeriod)
                        (type Integer))
                (Domain (name ServerCount)
                        (type Integer)))
       (TimeStamp yes)
    ... )
(Event (name RequestService)                ; MsgEvents
       (Location StimulusControl)
       (Domains (Domain (name TokenID)
                        (Type Integer)))
       (TimeStamp yes)
    ... )
(SensorProcess (name Servers)               ; SubTasks
    ... )
(Event (name ServersStart)
       (Location Servers)
       (Domains (Domain (name Index)
                        (type Integer))
                (Domain (name TokenID)
                        (type Integer)))
       (TimeStamp yes)
    ... )
(Event (name ServersEnd)
    ... )
(Event (name StartGlobalPhase)              ; User-defined sensors
    ... )
...
```

The use of receptacles and sensors may extend from sensors implemented in hardware to sensors embedded in the operating system to sensors placed in the user's program. It is the resident monitor's responsibility to extract the event records from the receptacle and send them to the relational monitor. By the time the relational monitor receives the event records, they are in an identical format, regardless of how they were generated.

10.7. Schema Management

The central management and control of the schema and the schema instances is performed by the schema manager. Functions of the schema manager fall into two broad categories: the creation, manipulation, and execution of the schema, and the creation, archiving, and cross-analysis of schema instances. The schema manager is organized into three main functional parts:

1. A user interface provides a uniform view of the various components of the IIE. Schemata can be created using conventional text editors or incrementally by directing the IIE to perform a series of runs. In the latter case, the corresponding schema and schema instance are automatically generated and archived. This incremental mode is particularly helpful in the tuning of experiments. The user interface also directly supports monitoring queries and database queries, thereby allowing a user to manipulate and analyze schema instances.
2. A schema interpreter scans the schema and sends control directives to the run-time system, including global initialization commands for the entire experiment, along with commands to set up, start, and terminate each run.
3. A schema-instance generator interacts with the relational monitor to ensure that an instance is created and placed in the database. Both predefined and user-defined relations are created and stored in the schema instance as a result of interpreting the schema.

The schema contains all the information necessary to perform a complete experiment. It consists of five major components: the system configuration, the stimulus, monitoring directives, initial experiment conditions, and experiment directives (see Figure 10-8). The system configuration completely defines the environment in which the experiment is to be performed. The stimulus is in the form of a translated B-language program, containing controlling parameters and data-collection sensors as described in Section 10.3. The monitoring directives are in the form of a collection of queries as described in Section 10.4. The initial experiment conditions consist of a set of invocation parameters and the required resources (i.e., hardware and operating-system configuration and instrumentation, stimulus version, etc.). Invocation parameters can be used to initialize parameter values for experiments and typically are specified at schema-interpretation time. The experiment directives are interpreted by the schema manager and specify how the stimulus should be executed. Specifications are provided for the iteration of the stimulus over the experiment runs and for the variation of parameters for each run.

During schema execution, the relational monitor creates a schema instance to hold the results of the experiment. The monitor collects all the resulting event records together with the schema identification and environment information and creates an object to be managed by the programming environment. By using standard relational database queries, the user can then perform analyses across

Figure 10-8 High-Level Organization of a Schema

```
schema (<invocation parameters>)
        <system configuration>
        <stimulus>
        <monitoring directives>
        <initial conditions>
        <experiment directives>
end schema
```

schema instances. The data in the instance, which is collected automatically, provides the user with enough information to replicate any particular execution of the schema to verify the results.

To illustrate the use of schema and schema instance, consider the schema describing the MPX, shown in Figure 10-9. The schema has two invocation parameters, *RequestPeriod* and *SDA*. The configuration data specifies the resources requested by this experiment, including the versions of the operating system and IIE components, the hardware components, data files to be read by the stimulus, and initial tests to be used later to calibrate the results. The experiment directives are in the form of a loop that generates the execution of five runs. Each run will have its own value for the *NoOfServers* parameter. The execution of this schema will terminate when 30 seconds have passed for each run. During execution, the sensors implanted in the B-language program will generate data that is collected according to the monitor queries.

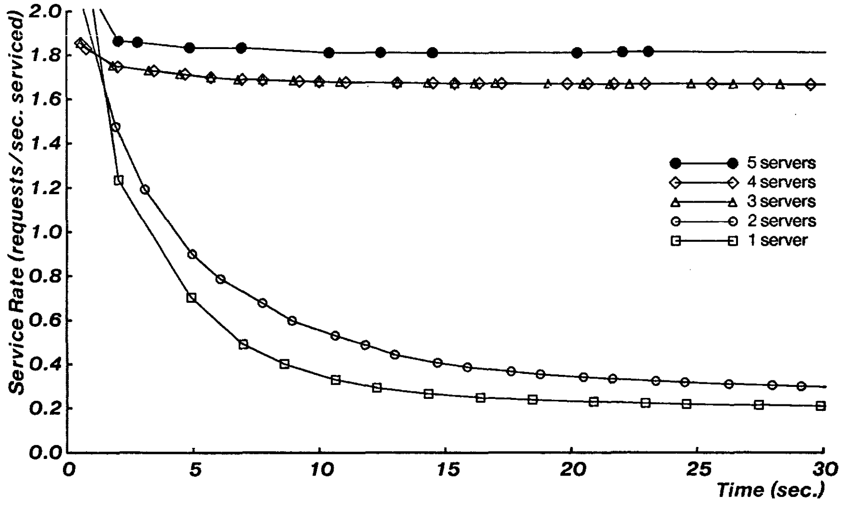
Each time this schema is interpreted, a schema instance will be created automatically in the database by the IIE. Each instance will have the following components:

- The date, time, and user identification.
- The values of the invocation parameters.
- Exact version numbers of all software used in the experiment.
- A detailed description of the hardware configuration.
- Results of the initial tests as specified in the system configuration.
- The system- and user-defined relations (in this case, the *PerRun*, *AverageQLength*, and *ServiceRate* relations).

Figure 10-9 The Schema for the MPX

```
Schema mpx (RequestPeriod, SDA)

SystemConfiguration <configuration data>;

TaskForce <B-language program>;

MonitorQueries <relational queries>;

ResultRelations AverageQLength, ServiceRate;

vary SharedDataAccess[i] = SDA where i from 1 to 5;

vary NoOfServers from 1 to 5
do
          BeginExperiment
          enable Server[i] where i from 1 to NoOfServers;
          terminate after 30 seconds
          EndExperiment
od
EndSchema
```

Figure 10-10 Service Rate vs. Time for a Variable Number of Servers



Once the instances have been created, additional analysis can be performed on the instances individually or as a group. Figure 10-10 shows the relationship between average service rate and time for a *RequestPeriod* of 200 ms. and a value of *SharedDataAccess* of 400 accesses per request. Initially, the service rate is high because the buffer is empty. For five servers, the buffer never contains many requests, so the average service rate remains high. For fewer than three servers, however, the buffer fills up quickly, causing the average service rate to plummet. The behavior with three or four servers is more involved, and further analysis is necessary using different values for the request period and the SDA.

10.8. Summary

The IIE constitutes a systematic approach to experimentation on multiprocessors. This approach emphasizes the integration of experimentation tools and the development of techniques for experiment management. The tools integrated in the IIE deal primarily with performance measurements on the components of a multiprocessor application. The emphasis of the IIE on performance-evaluation tools is important because the efficient implementation of parallel applications is one of the main research issues in parallel processing.

As the IIE constitutes the infrastructure for experimentation, it is appropriate at this point to discuss the experiments performed on Cm*. Part IV will emphasize performance aspects of parallel algorithms as reflected in the experiments performed on Cm*.

Acknowledgment. This chapter has been adapted from [Segall 83].

IV. Experiments

11. Performance of Parallel Algorithms

The execution of parallel algorithms is the *raison d'être* of multiprocessors. Such computations are characterized by substantial *interdependence* of their components—relatively frequent communication among simultaneous processes, either by shared memory or explicit message passing. Given the ease of linking independent processors together in a network, only the need for a high communication bandwidth can justify the elaborate interconnection structures that characterize more closely coupled parallel processors.

How should we measure the performance of a parallel algorithm on a multiprocessor? The traditional measures for uniprocessor algorithms are elapsed time (sometimes called completion time) and execution time.[1] Both measures depend to some extent on the system load at the time of the measurement—elapsed time because it is affected by contention for shared resources such as peripherals or locks on data, and execution time because its value usually includes part of the overhead of processing asynchronous interrupts generated by other processes and context swaps due to time slicing by the scheduler. For this reason, the measurements are usually made under some standard condition, such as an otherwise idle system.

Multiprocessor algorithms, too, usually are measured on an otherwise idle system. Elapsed time is defined as the completion time for the entire algorithm—that is, the instant when the last processor finishes. Execution time usually has been seen as a less important measure because if the system is otherwise idle, the goal is to finish the computation fastest, not to use the least processing power. As multiprocessors move out of the experimental phase and into service, it will be increasingly important to execute multiple task forces simultaneously (see Section 5.5), and thus execution time per processor will become more important. This measure has not, however, been stressed in these experiments. Instead, we have concentrated on *speedup*, which measures how well an algorithm benefits from additional processors.

11.1. Speedup

At first glance, it might seem that most problems could be solved faster on a multiprocessor. Few algorithms, after all, are strictly sequential by nature. But offset against the potential parallelism is the overhead of creating, synchronizing, and communicating with additional processes. These are pitfalls on which many algorithms falter.

Sometimes these considerations are serious enough to warrant major changes in

[1] These measures should be contrasted with raw instruction-execution speed, throughput, and response time, which endeavor to measure the performance of a computer over an entire workload rather than a particular algorithm.

an algorithm when it is decomposed for a multiprocessor. It may be worthwhile to redo a small set of calculations in each parallel process rather than pay a penalty to access a single copy of the results. Occasionally, benefits can be realized by completely changing the algorithm. For example, *randomized* algorithms become attractive for some problems as the degree of parallelism increases [Mehrotra and Gehringer 85]. Even though some processors may redo the work of others (by following the same path in a global search, for example), the decrease in communication costs may more than offset the redundant effort.

Consequently, a multiprocessor algorithm may do more total work than its uniprocessor counterpart. If run on one or two processors, it might actually run longer than the serial algorithm, but as the number of processors is increased, the parallelism begins to pay off, and the algorithm begins to show "speedup."

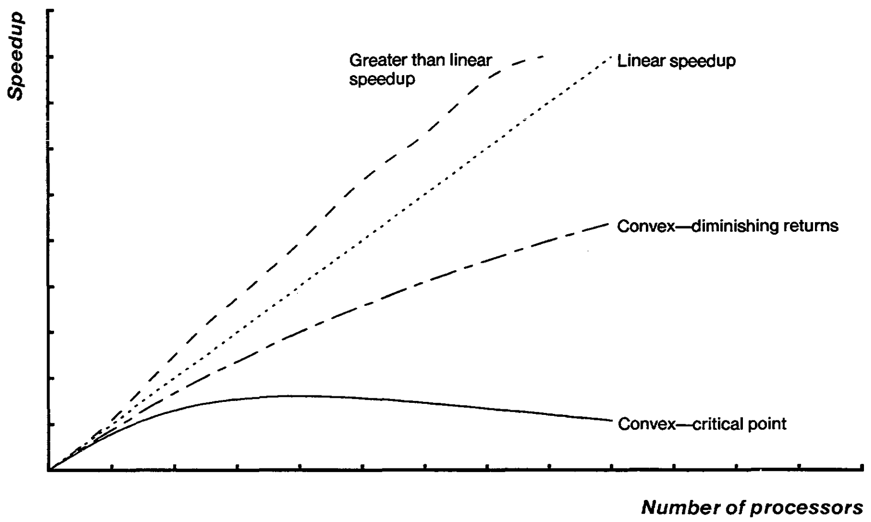
Speedup is a measure of whether an algorithm succeeds in harnessing the potential parallelism of a multiprocessor like Cm*. Intuitively, it is a measure of how much faster a computation finishes on a multiprocessor than on a uniprocessor. An algorithm that finishes twice as fast shows a speedup of two—but a speedup relative to what? Should the execution time of the multiprocessor algorithm on *N* processors be compared to its execution time on one processor or to the execution time of the best known uniprocessor algorithm—which may be a totally different algorithm?

The first definition is more easily applied, because it frees us from the need to determine the "best known" uniprocessor algorithm. Fortunately, it also works well in practice because we rarely want to compare the performance of two different algorithms. Hence we define speedup as E_u / E_N, the ratio of the elapsed time required by a one-processor version of an algorithm to the elapsed time taken by its *N*-processor counterpart, assuming that all processors are equally powerful. Speedup is almost always between 1 and *N*. Occasionally, one encounters a speedup of more than *N*. This topic is considered in Section 11.2. Often the speedup curve is convex in the number of processors; speedup rises as processors are added, up to a certain point, known as the *critical point*. Then the speedup begins to fall, meaning that adding more processors actually slows down the computation. Thus another interesting question is "What is the optimal number of processors for executing the algorithm?" Figure 11-1 illustrates how speedup may behave as processors are added.

It is important not to confuse speedup with execution speed. An inefficient uniprocessor algorithm may have very high speedup despite its long execution time. Synchronization and communication impose a rather small overhead. As processors are added, the execution time decreases almost proportionately. Now consider a more efficient algorithm requiring less computation time but the same synchronization and communication overhead. If enough processors are used, execution time decreases to the extent that the overhead begins to dominate the computation. Speedup is lower, but this algorithm finishes more quickly than the first. In this chapter, we will encounter several such algorithms.

During the past few years, the speedup of several algorithms has been measured on Cm* under widely varying conditions. Early experiments were performed with only a ten-processor Cm* system, precluding realistic multicluster investigation. In

Figure 11-1 Several Shapes of Speedup Curves



later experiments, the number of Cm's per cluster has changed from time to time. Some experiments have enjoyed the full support of an operating system; others have been built "from scratch," using only rudimentary Kmap microcode. The details of these experiments can be found in Appendix A. We will refer to them to illustrate our points.

11.2. Greater Than Linear Speedup?

An algorithm is said to exhibit *linear* speedup if the time taken by an *N*-processor version is one-*N*th the time used by the single-processor version—in other words, if its speedup is *N*. Is it ever possible for a multiprocessor algorithm to exhibit *superlinear* (greater than linear) speedup? Suppose such an algorithm does exist. Then for some value of *N*, the *N*-processor implementation must do less total work than its uniprocessor cousin. But suppose we simulate the multiprocessor solution by time slicing a single processor among the *N* processes (which, in effect, become coroutines). A faster uniprocessor implementation results. Its run time is not more than *N* times that of the original *N*-processor solution; compared to it, the multiprocessor solution has not been "sped up" by more than *N*. Does this mean that superlinear speedup is impossible?

Given our definition of speedup, the answer is no for the following reasons:

The new uniprocessor solution has a different control structure from the original one. Thus it cannot properly be called the same algorithm, and it is incorrect to measure speedup of the multiprocessor implementation with respect to it.

Depending on the architecture and operating system, it may be time-consuming to switch between coroutines, especially because it may have to be done arbitrarily often to simulate parallelism effectively. This cost may offset the advantage of restructuring the algorithm.

There are some real-time algorithms whose total *work* increases as their execution time grows longer. If a consumer falls behind a producer, for example, it may have to do extra bookkeeping to keep its agenda straight.

Let us now survey a few cases of superlinear speedup.

11.2.1. Search Problems

Search algorithms may occasionally obtain superlinear speedup. The simplest case is where the input data just happens to be configured in a way that favors a particular processor when the data is parceled out. The data is really more responsible for this than the algorithm; for other sets of input data, speedup might be less than linear.

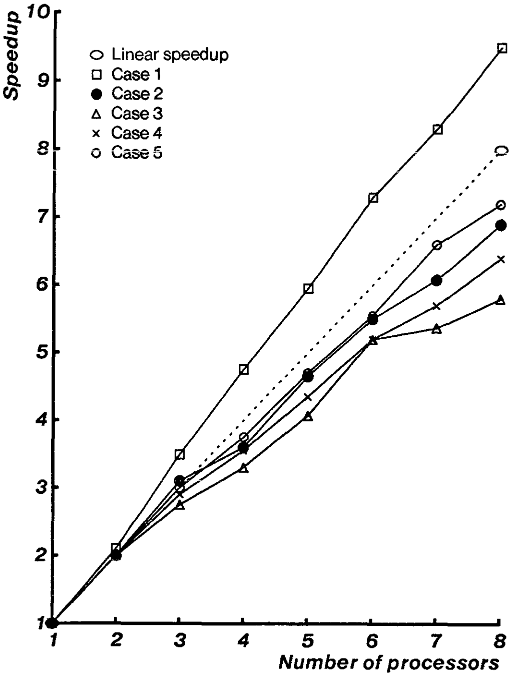
This phenomenon was observed with Raskin's integer-programming algorithm, described in Section A.3. This is a search algorithm. Its initialization phase puts a large number of possible solutions in a global stack, from which all the processors choose their work. As the search proceeds, a global variable holds the cost of the best solution found so far by any processor. All processors compare their current cost value to it and begin to backtrack in the search when the global cost is lower.

It is possible for the multiprocessor version to be "lucky." If one of its processors encounters a near-optimal solution at the outset, none of the processors will have very much work to do. The uniprocessor version, which does not encounter the near-optimal solution until later, has the disadvantage of having done a more complete search over the earlier possible solutions. But the opposite also can happen. Suppose the near-optimal solution turned up, say, first of all. The uniprocessor and multiprocessor versions would then encounter it at the same time. But before its cost could be determined, the other processors in the multiprocessor version would have wasted processing time on their initial solutions.

Hence the multiprocessor version cannot be lucky all the time. A search program can occasionally exhibit superlinear speedup, but it cannot *always* show greater than linear speedup over all possible sets of input data. The results shown in Figure 11-2 illustrate this. Only one of the five integer-programming runs managed to surpass linear speedup.

Next we might ask whether it is possible for a search algorithm to have superlinear speedup on average—that is, whether the expected execution time of an N-processor implementation can be less than 1/Nth the expected execution time of the uniprocessor version. Again, the answer is yes, but only if the execution time of the algorithm obeys a certain kind of probability distribution.

Figure 11-2 Speedup of Integer-Programming Computation



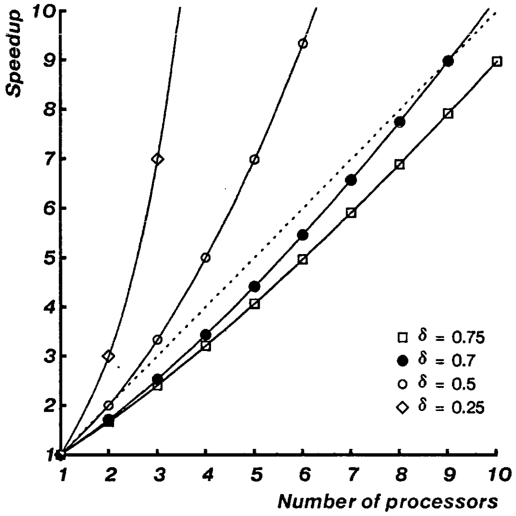
Weide [Weide 78] originally analyzed the characteristics of algorithms with superlinear speedup. Such an algorithm usually has a rather short execution time, but it executes for a long time often enough to raise its mean execution time well above the median. Given a random input, the uniprocessor version of this algorithm has an expected execution time equal to the mean of the execution-time probability distribution function. The expected run time of the N-processor version is the *expected minimum of* N *draws from this distribution*. If small values are sufficiently likely, the expected run time with N processors will be less than $1/N$th of the mean.

A specific example is an algorithm whose run-time probability distribution function is $F(x) = x^{\delta}$, $0 \leq x \leq 1$, $\delta > 0$. Weide showed that, if X is a random variable having the distribution F, and $X_{1:k}$ is the smallest of k such random variables, then

$$E(x) = \frac{\delta}{1+\delta}$$

$$E(kX_{1:k}) = (k \cdot k!) \Big/ \prod_{j=1}^{k} (j+1/\delta)$$

Figure 11-3 Hypothetical Superlinear Speedup



When δ < ½, then $E(kX_{1:k}) < E(X)$ for all $k > 2$. Also, if δ < 1 then $E(kX_{1:k}) < E(X)$ for sufficiently large k.

Figure 11-3 shows the speedup of such an algorithm for various values of δ. Note that these algorithms could be used to derive better single-processor algorithms via the coroutine method mentioned above. Thus algorithms that show better than linear speedup are not optimal, except perhaps when the overhead of coroutine-switching is taken into account. One can conjecture about the characteristics of other search algorithms whose speedup also would be greater than linear. Wilkes [Wilkes 77] cites two hypothetical examples. Mehrotra and Gehringer [Mehrotra and Gehringer 85] have shown an algorithm for finding a leaf node of a search tree that does in fact achieve superlinear speedup.

11.2.2. Consumer Algorithms

Consider a consumer algorithm that runs in real time and whose chore is to process input generated by a producer that is running simultaneously. The consumer is activated periodically and runs until it "catches up" with the producer. The faster the consumer is, the less work it has to do. As extra processors are assigned, they split a smaller workload, raising the possibility of more than linear speedup. The StarOS *Garbage Collector* is such an algorithm. While the rest of the processors are busy copying and deleting object references (capabilities), the *Garbage Collector* is looking for unreachable objects.

Let r be the ratio of the rate at which capabilities are created to the rate at which the garbage collector can process capabilities. (If r is greater than one, garbage collection will be unsuccessful because capabilities will be created faster than the *Garbage Collector* can process them.) During a time period t, then, tr units of additional work accumulate for the *Garbage Collector.* While the *Garbage Collector* is performing this additional work, a further tr^2 units of work arrive, and so forth. Hence the time T to perform a garbage collection is given [Chansler 82] by

$$T = t + tr + tr^2 + tr^3 + \cdots,$$

or

$$T = \frac{t}{(1-r)} = t\left[1 + \frac{r}{(1-r)}\right]$$

If N parallel processes are used to collect garbage, then r is reduced by a factor of N, so the *total* time needed to collect garbage becomes

$$T_N = \frac{t}{(1 - r/N)}$$

If r is between 0 and 1, then $T > T_N$. So if the work is divided evenly among N processes, the per-process execution time is T_N/N, which is less than T/N. The speedup is superlinear. Suppose, for example, that $r = 1/3$. Then $T_2 = 0.8T$, yielding a theoretical speedup of 2.5 with two processors. Figure 11-4 shows that superlinear speedup has been noted in practice. Section A.17 treats these observations in more detail.

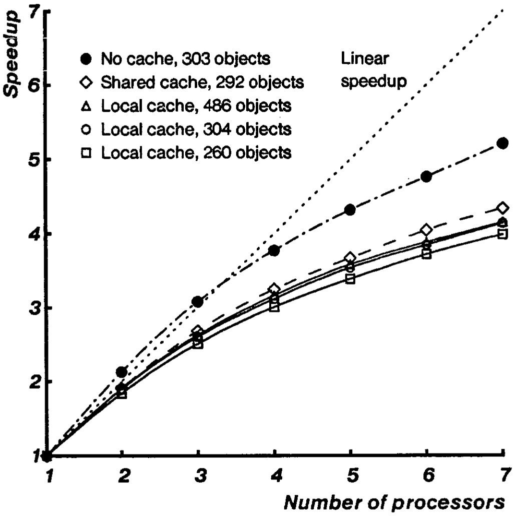
There is a sense in which this speedup, too, is illusory. If we assume that the *Garbage Collector* is activated at constant intervals (for example, each hour on the hour), high speedup manifests itself in a shorter run time and in correspondingly longer idle periods. More work accumulates between activations. If the rate of capability creation and garbage-collection time per capability are constant, the extra work offsets the extra speedup.[2] Ignoring startup transients, over an activation interval (e.g., one hour), the N-processor garbage collector is active no less than $1/N$th as long as the single-processor version.

11.3. Factors That Limit Speedup

Most parallel algorithms partition a fixed workload among a number of processors, each of which iterate through their entire partition. The algorithm terminates when all processors have finished. Linear speedup is the maximum obtainable by these

[2] That is, ignoring the possibility that the garbage collector requires enough processors to prevent other runnable processes from executing, in which case it affects the rate at which capabilities are created.

Figure 11-4 Speedup of the StarOS Garbage Collector



algorithms. It is achieved only if the workload is partitioned evenly among the processors, and no processor ever needs to wait for another—either to synchronize, access global data, or receive a message. Although linear speedup is sometimes approached, it is rarely attained, since some algorithms apportion data unequally, and almost all manifest some sort of interaction among processors. This degradation is known as the *decomposition penalty*.

The extent of the decomposition penalty depends on the nature of the algorithm, the computer on which it runs, and the interaction between the two.[3] These factors are summarized in Table 11-1 and described below.

11.3.1. Algorithm Penalty

The *algorithm penalty* arises from the nature of the algorithm itself. An algorithm suffers when it is unable to keep all its processors busy all the time.

Separation Overhead. Some algorithms lend themselves to static partitioning of data. For example, in a matrix multiplication, each processor can be assigned a portion of the matrices. The assignments can all be made at once, before any of the processors have begun their work. By contrast, some divide-and-conquer algorithms require dynamic partitioning. The first processor divides the data into two or more parts, passing each along to another processor, which in turn divides its portion. This process continues until the data can be divided no further or until it reaches a

[3] A similar, though less detailed, discussion can be found in [Talukdar 79].

Table 11-1 Factors Preventing Linear Speedup

Algorithm penalty	
Separation overhead	Cost of parceling work out to processors.
Reconstitution overhead	Cost of gathering up results.
Implementation penalty	
Access overhead	Cost of remote accesses to shared memory.
Contention overhead	Degradation of remote-access time due to contention.
Algorithm / implementation interaction	
Synchronization overhead	Cost of having some processors idle waiting for others to deliver results. Usually results from variations in processor speeds.
Parallelization overhead	Calculations redone in each process to diminish implementation penalty.

"manageable" size. Since some processors are necessarily idle at the outset, speedup is less than linear. We refer to this as the *separation overhead*.

Reconstitution Overhead. An analogous situation can occur at the end of the computation, if it is necessary to combine the results. As an example, a parallel insertion sort could statically partition its data among the processors, but a merge phase would be necessary to gather the results from the different partitions. This too could be performed in parallel, with a decreasing number of processors merging ever-larger subarrays, until only a single processor was active at the end. The idle processors induce a *reconstitution overhead*.

Also common are *multiphase* algorithms, in which parallel phases alternate with serial (single-processor) phases. Usually, the serial phase performs separation or reconstitution; its length limits the overall speedup of the algorithm.

A simple model serves to demonstrate how sensitive speedup is to the length of the serial phase. Assume that a unit of work is composed of a parallel (P) and a serial (reconstitution) (R) phase:

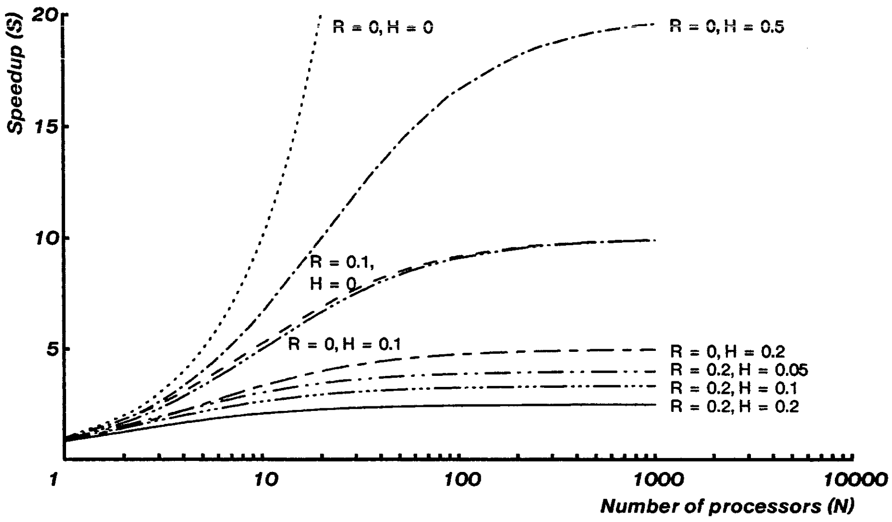
$$P + R = 1$$

Furthermore, assume that each parallel process experiences an overhead of H units due to other aspects of the decomposition penalty. The amount of work done by all processors in an N-processor multiprocessor configuration is

$$P + HN + R$$

Assume that P and H are subject to a linear speedup, yielding a total speedup of

Figure 11-5 Parallelization and Serial Phase Overhead vs. Speedup



$$S = \frac{1}{(P/N) + H + R}$$

Figure 11-5 plots the speedup S as a function of N for various values of H and R. The curve marked $R = 0$, $H = 0$ represents linear speedup. Note that an overhead factor of H asymptotically limits speedup to $1/H$. Likewise, the serial component of an algorithm limits speedup to $1/R$. For example, an algorithm that spends one-fifth of its time in reconstitution calculations cannot achieve a speedup of greater than 5.

11.3.2. Implementation Penalty

The *implementation penalty* is an artifact of the multiprocessor on which an algorithm is run. It measures the degradation due to interprocessor communication, via either shared memory or messages. Like the algorithm penalty, it has two components.

Access Overhead. The first component is the *access overhead*. It reflects the cost of fetching code or data from a remote part of the multiprocessor. Its magnitude depends on the placement of processes and data, which affects the frequency of remote references on a multiprocessor like Cm*. It also includes the cost of input and output. These are relatively slow operations; traditional I/O structures cannot feed a large multiprocessor fast enough to avoid processor idle time. I/O for applications measured on Illiac IV, for example, has consumed up to 60 percent of execution time [Haynes *et al.* 82].

Clearly, access overhead increases execution time, but unless significant memory contention results, it has little effect on speedup. We will consider its effects after we have examined the impact of contention.

Contention Overhead *Contention overhead* is the penalty imposed by competition among processors for access to global data. It occurs when processors request data more quickly than memory is able to deliver it. For the sake of this discussion, we will assume that requests are serviced in FIFO fashion.

To model the contention overhead, imagine that we have an iterative algorithm. Each iteration is made up of a certain amount of computation, requiring t_p time units, and a certain amount of access to global data (or I/O, etc.), taking t_a time units. Let us further assume that there is only one period of global access and one period of computation per iteration; that is, assume that all the global access in an iteration occurs before any of the computation, or vice versa. (Not all parallel algorithms meet this condition, but it represents the best case; algorithms that satisfy it perform no worse than those that do not. A proof, along with a complete description of the model, is found in [Vrsalovic *et al.* 84a]. Appendix C summarizes salient aspects of the model.)

If the algorithm is *asynchronous*—that is, if there exist no global synchronization points that all processors must reach before any can continue—then speedup depends only on whether all other $N-1$ processors can complete their global accesses during the time that a particular processor is computing (see Figure 11-6); in other words, speedup depends on whether $(N-1)t_a \leq t_p$. If so, then after transient startup effects, no processor will ever be delayed by another's global accesses, and linear speedup will be achieved. If not (assuming FIFO queueing for global data), speedup will be limited to $1+t_p/t_a$. In other words, speedup equals $\min[N, 1+x]$, where $x=t_p/t_a$. Vrsalovic ran a synthetic task force with these characteristics on Cm*. The experimental results correlate fairly well with those predicted by the experimental model, as shown in Figure 11-7.

One hidden assumption in this model is that the ratio x of processing time to access time remains constant as N increases. This is true for algorithms such as matrix multiplication and Fourier transform, where essentially all the data is global to the entire computation; it is not true for all parallel algorithms. Partial differential equations (PDEs) are a case in point. The PDE algorithm that runs on Cm* (see Section 11.5.3) processes a large grid. The grid is divided into N rectangles, one for each processor. Only the elements at the boundaries of these rectangles are global data; the rest are local to the processes that operate on them.

As the number of processes increases, the rectangles shrink. The amount of global data per rectangle diminishes only as fast as the perimeters of the rectangle—a rate proportional to the *square root* of N. Because the amount of processing is proportional to N, the processing-to-access ratio x declines as N increases. It is not a matter of dividing a fixed amount of global data among a larger number of processors; rather, the amount of global data actually *increases* with N.

Figure 11-6 Overlapped Processing and Access Time

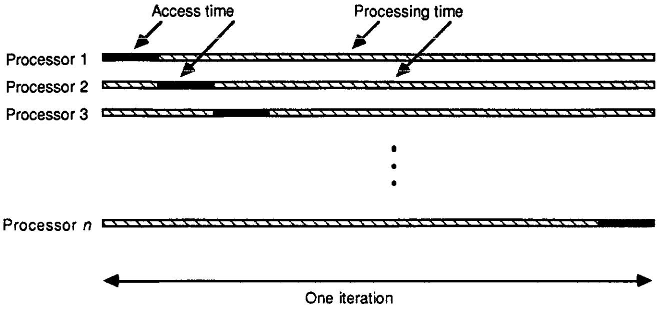
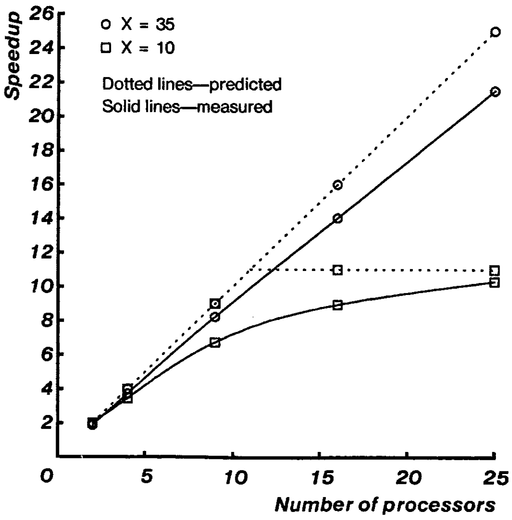


Figure 11-7 Speedup of an Asynchronous Algorithm with an *(N; N)* Decomposition



The overall rate of references to global data increases, and the access paths eventually saturate, limiting the speedup.

To accommodate a changing *x* in the model, let us define T_p and T_a as the processing and access times in the *uniprocessor* implementation of the algorithm.

Let

$$X = \frac{T_p}{T_a}$$

We must also specify how t_p and t_a change with respect to N. Let us define the *decomposition functions* as

$$f_p(N) = \frac{T_p}{t_p(N)}$$

$$f_a(N) = \frac{T_a}{t_a(N)}$$

Expressed in these terms, our "hidden assumption" was that $f_p = f_a = N$. This is called an $(N;N)$ *decomposition*. The PDE algorithm has (approximately) an $(N;\sqrt{N})$ decomposition. The model [Vrsalovic *et al.* 84] predicts speedup for an algorithm with an $(f_p;\ f_a)$ decomposition as

$$S = \min\left[f_a f_p \frac{1+X}{f_p + X f_a},\ f_a \frac{1+X}{N}\right]$$

For the particular case of an $(N;\sqrt{N})$ decomposition,

$$S = \min\left[\frac{(1+X)N}{\sqrt{N}+X},\ \frac{1+X}{\sqrt{N}}\right]$$

Vrsalovic measured the behavior of an algorithm with this decomposition on Cm*; Figure 11-8 compares predicted and observed speedups. Other common decomposition groups are $(N;1)$ and $(\log N;\ \log N)$. For an $(N;1)$ decomposition, the predicted speedup is

$$S = \min\left[\frac{N(1+X)}{N+X},\ \frac{1+X}{N}\right]$$

Figure 11-9 shows the results. For a $(\log N;\ \log N)$ decomposition, speedup is predicted as

$$S = \min\left[\log N,\ \frac{(1+X)\log N}{N}\right]$$

The results are given in Figure 11-10.

What conditions must be satisfied to achieve superlinear speedup? Assuming that $f_p = N$—that is, that as the number of processors increases, a constant amount of total computation is simply divided among the processors—then it is easy to show that f_a must be $O(N^2)$—that is, at least as large as a constant times N^2. In other

Figure 11-8 Speedup of an Asynchronous Algorithm with an (N; $\sqrt{N}$) Decomposition

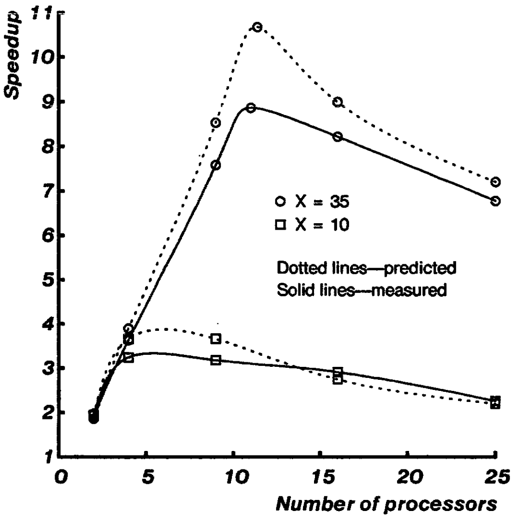


Figure 11-9 Speedup of an Asynchronous Algorithm with an (N; 1) Decomposition

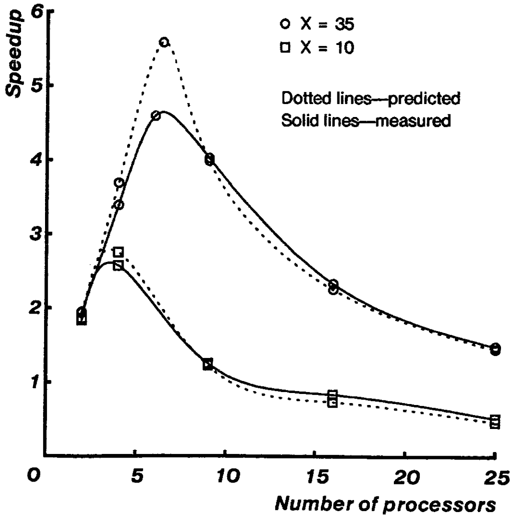
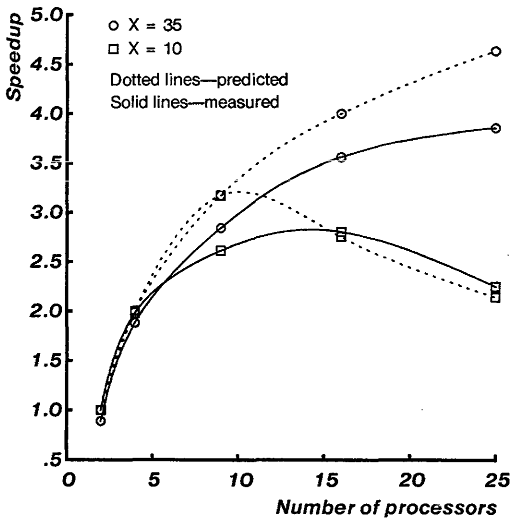


Figure 11-10 Speedup of an Asynchronous Algorithm with an (log *N*; log *N*) Decomposition



words, the amount of global access by each processor has to fall with the square of the number of processors. This means the total amount of global access *by all processors* must be inversely proportional to the number of processors. It is a most extraordinary algorithm that satisfies this constraint!

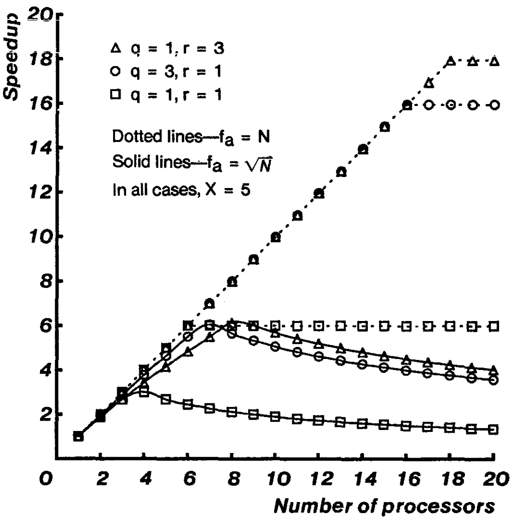
Access Overhead Revisited. The cost of accessing global memory depends on two characteristics: the *speed* at which data is delivered and the *bandwidth*, which we shall define as the number of requests that can be serviced at once. It is straightforward to extend our model to take these factors into account (see Section C.2.1). If q represents the global-memory speed relative to a reference speed of 1, and r represents the bandwidth, then

$$S = \min\left[f_a f_p \frac{1+qX}{f_p + qXf_a},\ r f_a \frac{1+qX}{N}\right] \tag{1}$$

Let us investigate the magnitude of the access overhead by hypothesizing that we could speed up global references by a factor of three. (In Cm* terms, this would be like changing all intracluster references to local references.) The model predicts a behavior like that shown in Figure 11-11. The figure also shows the model's predictions for an r of 3—that is, for a memory that can service three requests in parallel.[4]

[4] In Cm*, if intracluster references are directed to *different* Cm's, r is nearly 2, due to the Kmap's ability to maintain more than one context. Section 3.1.3 reports some measurements.

Figure 11-11 Effect of Increasing Global-Memory Speed



Notice that it matters little whether q or r is improved, although raising q is slightly more advantageous when the number of processors is small and slightly less advantageous as the number of processors increases.

When is speedup most sensitive to the global-access speed q? Clearly, it is most sensitive when an algorithm is global-reference intensive (i.e., when X is low). Figure 11-12 plots global-memory speed against speedup. Notice that the curves consist of two regions. Speedup first increases sharply and then levels off, or nearly so. In the first region, contention is present. Here,

$$r f_a \frac{1+qX}{N} < f_a f_p \frac{1+qX}{f_p+qXf_a} \qquad (2)$$

(i.e., the second term in equation 1 is the minimum). As q increases, contention disappears, and equation 2 no longer holds. In the special case that $f_p = f_a = N$, speedup in the second region is constant at N. If, however, the total demand on global memory increases with N, memory speed still slightly influences speedup, which asymptotically approaches N. For example, an LSI-11 processor has no floating-point hardware; hence floating-point algorithms for Cm* have a high t_p and thus a high X. Similarly, r has less effect on execution time as X increases (Figure 11-13). When

$$X \geq \left[\frac{N}{r} - 1\right] \frac{f_p}{q f_a}$$

contention disappears, and r has no effect at all.

Figure 11-12 Insensitivity of Speedup to Global-Memory Speed in the Absence of Contention

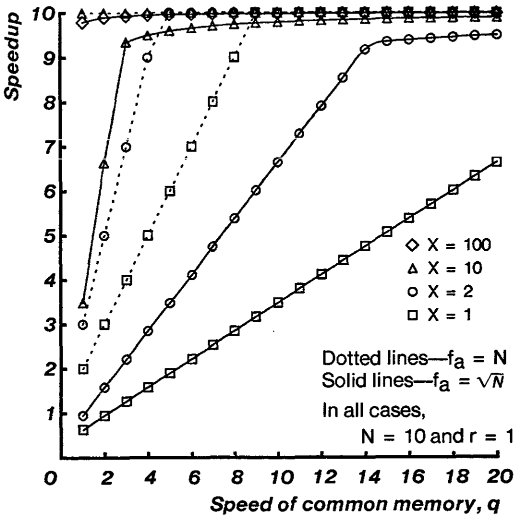
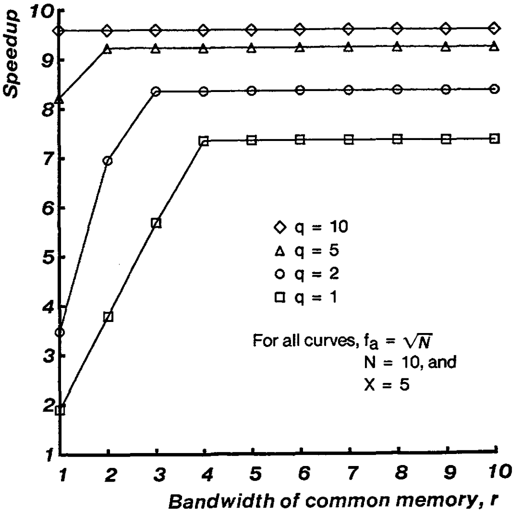


Figure 11-13 Insensitivity of Speedup to Global-Memory Bandwidth in the Absence of Contention



11.3.3. Algorithm / Implementation Interaction

It is now time to focus on performance factors that are influenced heavily both by the nature of a parallel algorithm and by the multiprocessor on which it runs. Synchronization is one such factor; it is a characteristic of particular algorithms, but its impact depends greatly on the architecture that executes it. For some algorithms —synchronous or not—it may be fruitful to do redundant calculations in each process to diminish the need for global-memory access or synchronization.

Synchronization Overhead. The presence of global synchronization points obviously limits the performance of a parallel algorithm. This is true for two reasons. First, algorithms tend to spawn parallel processes that are identical except for the data on which they operate. Running in lockstep, the processes tend to reference global data at almost exactly the same time, effecting maximal contention. Second, even without contention, processor speeds vary. Synchronization means that all processors are limited to the speed of the slowest.

The worst case, in terms of synchronization's impact on performance, occurs when there is only one global-access period per iteration. Then, there is one processor that cannot overlap its processing period (of t_p time units) with any global accesses at all. A single iteration takes $t_p + Nt_a$ time units. The theoretical speedup of the algorithm is given by [Vrsalovic *et al.* 84a]:

$$S = \frac{r f_a f_p (1 + q X)}{N f_p + q r X f_a}$$

Regardless of the amount of synchronization, then, the performance of the algorithm is bounded above by the best-case asynchronous algorithm (page 235) and below by the worst-case synchronous algorithm. Vrsalovic measured the performance of the worst-case synchronous algorithm for the same decomposition groups introduced in the graphs of Section 11.3.2. For the $(N;N)$ decomposition group (Figure 11-14), the synchronous speedup is given by

$$S = \frac{(1 + X)N}{N + X}$$

For the $(N;\sqrt{N})$ decomposition group (Figure 11-15), the synchronous speedup is given by

$$S = \frac{(1 + X)N}{N^{3/2} + X}$$

Even if there were no global access at all, the presence of synchronization points would slow down a computation because of variations in processor speed. In a synchronous algorithm, define an *algorithm iteration* as the time it takes the entire task force to complete an iteration and a *process iteration* as the time it takes a

Figure 11-14 Speedup of Synchronous and Asynchronous (*N*; *N*) Algorithms

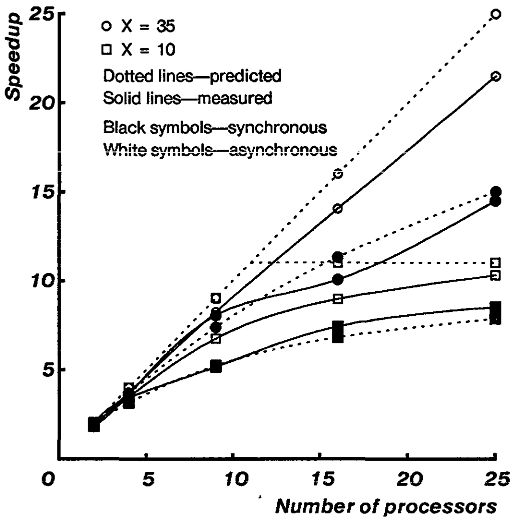
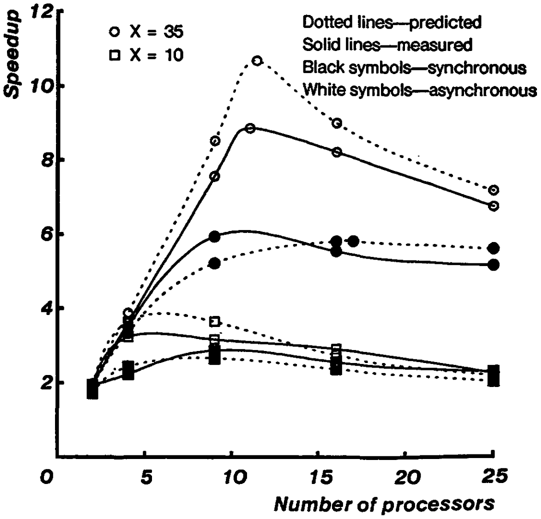


Figure 11-15 Speedup of Synchronous and Asynchronous (*N*; $\sqrt{N}$) Algorithm



single process to finish an iteration. Note that an algorithm iteration is not complete until all the process iterations are complete. Processor speeds vary. For example, the fastest Cm* processor is 4.6 percent faster than the slowest one (see Section 3.1.1). In addition to processor-speed variations, different processes may take different branches within an iteration. Finally, the overhead imposed by background tasks may differ from processor to processor; clocks must be updated, and accounting information must be collected, for example. These three factors introduce randomness into the execution time for a process iteration. The time taken for an algorithm iteration is the maximum of *N* draws from the process iteration-time distribution. To the extent that this value is greater than the mean, it represents synchronization overhead.

Taking into account these factors, the synchronization penalty itself becomes probabilistic and hence more difficult to model analytically. Mohan [Mohan 84] created a hybrid simulation model to predict the performance of parallel algorithms; synchronization was one of the major factors he investigated. The model is called the *Performance Estimation Program*, or *PEP*. It is a hybrid because:

- It uses analytic means (the machine-repair model) to account for contention for shared memory.
- It simulates the behavior of individual processes on the process-iteration level. That is, for each iteration of each process, a time is drawn randomly from some probability distribution. The time between two synchronization points is merely the maximum of these times.

PEP is a general model, capable of adaptation for a wide class of multiprocessor structures. To validate the model and make specific performance predictions, though, Mohan tailored it to represent the structure of Cm* / StarOS.

There are some important differences between Mohan's model of parallel computation and that of Vrsalovic *et al.* Vrsalovic considers a parallel algorithm as being divided into *processes*, which, at equilibrium, are repeatedly iterating over the same portion of code. Mohan considers the processes as being divided into *tasks*, with each task consisting of one process iteration. The tasks are independent, except for the precedence constraints among them. (This definition of "task" comes from the scheduling literature.)

Thus Mohan and Vrsalovic define a totally synchronous system in the same way: a set of processes that perform a single iteration between two global synchronization points. To specify an asynchronous system, Mohan defines *physical parallelism*, *N*, as the number of available processors, and *logical parallelism*, *L*, as the number of tasks that can execute simultaneously without violating any precedence constraints. In a totally asynchronous system, the logical parallelism is equal to the total number of tasks, *T*. In a totally synchronous system, it is equal to the number of processors, *N*; precedence constraints require all the tasks in the ith iteration to finish before any task in the $(i+1)$st iteration is started; thus when one processor finishes its task, it cannot start any other task without violating a precedence constraint until all the tasks in the ith iteration are done.

Given these definitions, the degree of synchronism can vary along a continuum. For example, suppose that $N = 25$ and $T = 1{,}000$. Then L can vary between 25 (totally synchronous) and 1,000 (totally asynchronous). Mohan studied the effect of synchronization on execution time by using PEP to simulate a parallel computation and varying the number of synchronization points.

PEP is a straightforward stochastic simulation program. At the beginning, one task is assigned to each (simulated) processor (unless the precedence constraints prevent this). Each time a task finishes, another task is selected to run on each idle processor, if such tasks can be selected without violating the precedence constraints. The execution time of a task is lengthened, or "degraded," by a factor representing the overhead imposed by memory contention from all the active processes. This overhead is determined by the analytic model.

Task times were determined by one of two methods:

Deterministically. Each iteration of each parallel process took exactly the same amount of time.

Random sampling from a beta distribution. A beta distribution is conventionally used for task times and for the duration of activities in PERT modeling. The beta distribution can take on many different shapes depending on its two shape parameters: α, the square of the standardized measure of skewness of a set of observations; and β, the standardized measure of their peakedness or kurtosis.[5] The uniform, triangular, and exponential distributions are special cases of the beta distribution.

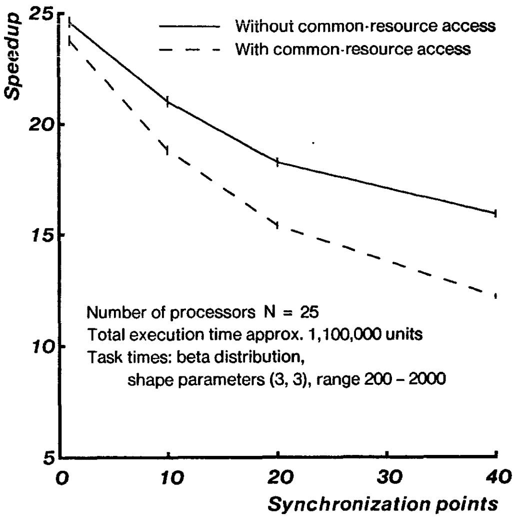
PEP was used to simulate the performance of a computation consisting of 1,000 tasks on 25 processors. The task times had a beta distribution with shape parameters (3, 3) distributed over a range of 200 to 2,000. The total computation time was about 1.1 million units, or about 44,000 units per processor. In other words, if linear speedup were achieved, the execution time would be 44,000. The upper curve in Figure 11-16 shows how the number of synchronization points influenced the speedup. Runs were made with 1, 10, 20, and 40 synchronization points. These numbers were chosen so that logical parallelism was always an integral multiple of 25, allowing the tasks to be evenly divided among the 25 processors.

With one synchronization point (at the end of the computation), execution time was 44,681 units—sublinear only because of the probabilistic task execution times. With 40 synchronization points, execution time rises to 69,018, yielding a speedup of about 16. Since the data points have been obtained by simulation, the results are only probabilistic. The vertical lines in the figure indicate the extent of the 95 percent confidence interval.

The degradation in speedup in this experiment is caused solely by the variation in iteration times. The next step [Mohan 84] is to add contention for a common resource (such as global memory) and measure its effect on speedup. Let the com-

[5] The beta distribution is defined by the density function $f(x) = Cx^{\alpha-1}(1-x)^{\beta-1}$ for $0 < x < 1$; otherwise $f(x) = 0$.

Figure 11-16 How Synchronization and Contention Affect Speedup



mon resource have a service time of 25 units, and let it be accessed once by each task, at some random instant within the first 100 units of execution. Otherwise, the simulated system is the same as in the previous experiment. The lower curve in Figure 11-16 plots the speedup and the confidence intervals with common access. It shows that speedup with contention varies from 23.78 with only 1 synchronization point (the asynchronous case) to 12.21 with 40 synchronization points (the completely synchronous case).

It is interesting to compare these results derived from simulation with those predicted by the analytic model of Vrsalovic *et al.* For the completely asynchronous case, it predicts a speedup of 25 [for the $(N;N)$ decomposition that arises from parceling out an equal number of identical tasks to each processor]. This is a higher speedup than that predicted by Mohan, owing partly to the nondeterministic task times in the simulation and partly to the fact that Vrsalovic assumes a fixed interval between resource requests by each process. As long as the global memory can service requests as fast as they are made $(N \leq 1+X)$, requests will never interfere with each other. In the simulation, requests are made at random times, so interference may occur, even when utilization is light.

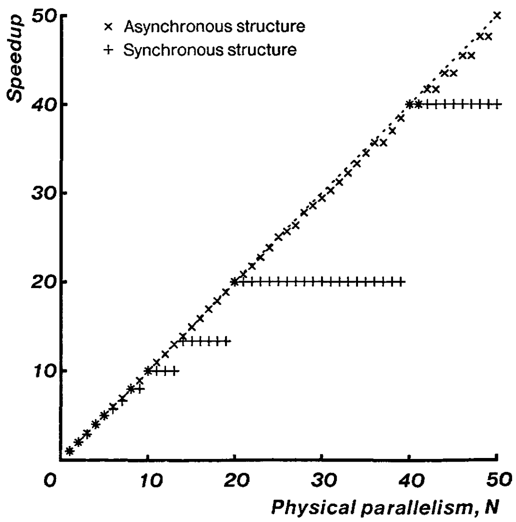
For the "worst case" of a synchronous algorithm, Vrsalovic's model gives a speedup of 16.3, whereas 12.21 is the result from simulation. The nondeterministic task times undoubtedly have much to do with this discrepancy—note from Figure 11-16 that nondeterministic task times and no contention yield a speedup of about 16, not 25 as would be obtained with deterministic task times and no contention. The ratio of 12.21 / 16.3 is higher than 16/25, because the randomness in access requests benefits the simulated system, when compared to Vrsalovic's model of synchronism, which posits that all processors make access requests simultaneously, a pessimal assumption.

Matching Parallelism and Task-Time Distribution. Most parallel algorithms divide a workload among *N* processors "as evenly as possible." If the workload cannot be divided exactly equally (that is, if its *granularity* is too high), linear speedup cannot be achieved. An equal division of the workload is called *matching parallelism* [Mohan 84]. Computations that are highly synchronous rarely approach matching parallelism because the logical parallelism (units of work between synchronization points) is on the same order as the physical parallelism (number of processors). In most asynchronous computations, the logical parallelism is large with respect to the physical parallelism, so the tasks can be distributed quite evenly among the processors. Nonetheless, matching parallelism and linear speedup can be achieved only when the number of processors evenly divides the number of tasks.

This is shown by Figure 11-17, which illustrates results for a computation of 1,000 tasks whose execution time is 1,000 units each. The synchronous computation has 25 synchronization points, for a logical parallelism of 40. The plot of speedup versus number of processors shows that the graph for a synchronous computation is a step function. The execution time of the computation is determined by the processor that has the most tasks to perform. For example, the last three steps, at speedups of 14, 20, and 40 ($\lceil 40/3 \rceil$, $\lceil 40/2 \rceil$, and $\lceil 40/1 \rceil$ respectively), are generated when the busiest processor has, respectively, three, two, and one task to perform. The asynchronous computation always has nearly linear speedup, but deviations from linear become

Figure 11-17

Speedup with Deterministic Task Times



larger as physical parallelism increases relative to the logical parallelism of 1,000.

Once again, it is easy to compare these results with Vrsalovic's model. The model assumes that a parallel computation is always evenly decomposable among N processors; thus no deviation from linear speedup is encountered in the asynchronous case. For the synchronous case, the model predicts a near-linear speedup for very large values of X (when the processing time is very large compared to the global reference time). Since this system performs no global resource accesses, speedup in the synchronous case should be linear if there is matching parallelism. It is.

Next, Mohan used a task execution time uniformly distributed between 0 and 2,000. Inequality of execution times on the different processors prevents linear speedup from being achieved, even in the asynchronous case and more notably in the synchronous case (Figure 11-18). The speedup of the synchronous computation rises smoothly to about 20, which is achieved at $N=36$, and levels off thereafter. In fact, it can be shown [Mohan 84] that with uniformly distributed task times, the speedup is bounded by the logical parallelism (40) times the mean task execution time (1,000) divided by the maximum execution time (2,000).

The third experiment used a beta distribution for task times, with shape parameters (3, 3) ranging over an interval from 0 to 2,000. Figure 11-19 shows that the asynchronous computation attains nearly linear speedup, while that of the synchronous computation levels out at 22.5, higher than with the uniform distribution because samples from this beta distribution tend to cluster more around the mean.

Parallelization Overhead. When the access penalty is serious enough, the performance of a parallel computation sometimes can be improved by changing the implementation to reduce the number of global accesses. A common example is the creation of *local copies* of frequently referenced global data.

Consider a program that multiplies two M-by-M matrices, solving the matrix equation

$$C = A \times B.$$

A straightforward parallel implementation partitions the result matrix into a set of square submatrices, each of dimension $M/\sqrt{N}$ *by* $M/\sqrt{N}$ (assume that N is a perfect square, and that $M/\sqrt{N}$ is an integer). We can view the submatrices themselves as being arranged in $\sqrt{N}$ rows and $\sqrt{N}$ columns, and let c_{RS} be the Sth submatrix in the Rth row. Notice that the process that computes c_{RS} accesses only the Rth row of A and the Sth column of B (Figure 11-20), but it makes $M/\sqrt{N}$ global accesses to each element within the Rth row and the Sth column. If the submatrices are large, it is clearly better to copy them into local memory at the outset; then each element from the global matrix is accessed only once. This is an example of how an implementation can be recast to improve performance. The extra work of copying, however, prevents the new version from attaining linear speedup with respect to the noncopying "basis" version. This extra work represents parallelization overhead.

The noncopying version of the algorithm has an $(N;N)$ decomposition. Assuming

Figure 11-18 Speedup with Uniformly Distributed Task Times

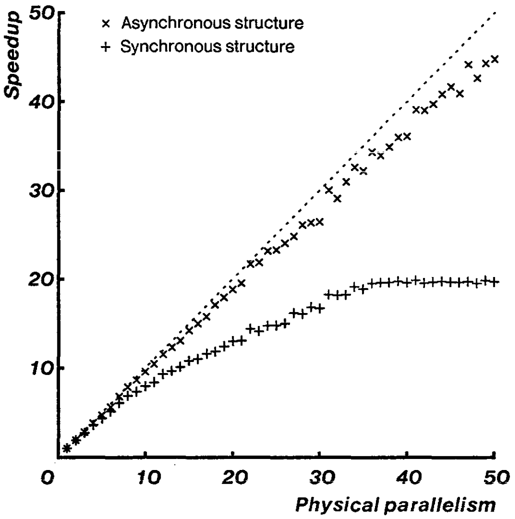


Figure 11-19 Speedup with Beta-Distributed Task Times

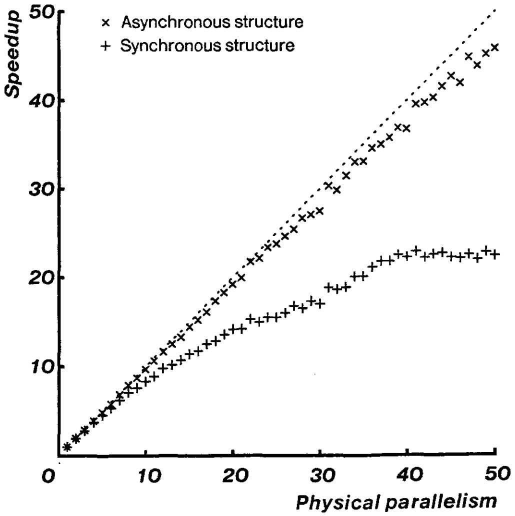
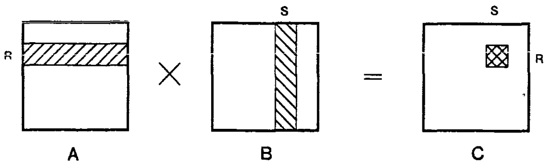


Figure 11-20 Submatrices Accessed During the Computation of a Submatrix Product



that the time to iterate through a row of the matrix is much greater than the time needed to copy a single matrix element from global memory, the copying version has essentially an $(N;\sqrt{N})$ decomposition. (The calculations are worked out in full detail in [Vrsalovic *et al.* 84b].) It can be shown that the speedup of an algorithm with an $(N;N)$ decomposition is strictly increasing, while an algorithm with an $(N;\sqrt{N})$ decomposition has a maximum (the graphs in Figures 11-2 to 11-7 [pages 229–236] illustrate this, and it is shown more formally in Appendix C). Assuming further that the processor time required to make a local copy of an element is much less than the time to access a global element, it becomes profitable to make local copies, as long as the number of matrix elements is at least somewhat larger than the number of processors.

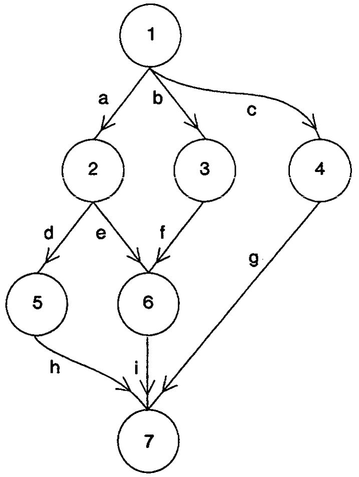
11.4. Parallel Algorithm Taxonomy

A typical parallel computation is structured as multiple processes, which are potentially executable in parallel. These processes generally synchronize the execution of various parts of their own computation with subcomputations of other processes. A *task* is the execution of a process between two consecutive synchronization points.[6] In other words, the subcomputations of a process that are executed independently of other processes are the tasks executed by that process. We can make the following observations about processes executing tasks: A process, whenever it is active, is executing one and only one task; a process, during its lifetime, typically executes a series of tasks; it does not execute any task when it is blocked or idle. Tasks, by contrast, synchronize with other tasks only at their end points; that is, any time ordering of tasks is done only at the start and completion of a task. Therefore, a task runs to completion without regard to the state of other tasks.

A parallel computation is a collection of computational fragments or tasks, possibly with some constraints on when a task can be executed relative to others. The tasks and their temporal interrelationships can be expressed as a *task graph* (see Section 10.3), where the nodes represent the tasks and the arcs represent the precedence constraints between them. Precedence constraints arise from data

[6] As "task" was defined on page 244, there could be more than one task between two consecutive synchronization points. In this section, we "lump together" all the tasks between two synchronization points and consider them to make up a single task.

Figure 11-21 An Example of a Task Graph



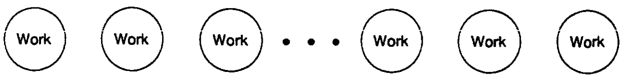
dependencies, control dependencies, and operator precedences inherent in computations [Lee 80]. Figure 11-21 portrays an example of a task graph, consisting of tasks (labeled 1 through 7) and precedence constraints (labeled *a* through *i*). Precedence constraint *a* specifies that task 2 can start executing only after task 1 is completed; precedence constraints *g*, *h*, and *i* specify that task 7 can be executed only after the completion of tasks 4, 5, and 6.

11.4.1. Six Algorithm Structures

The most visible characteristic of a parallel computation is its structure—the way its components relate to each together. The term "parallel structure" refers to the logical dependencies of the tasks within a computation. For example, an unlimited number of tasks can *logically* execute in parallel, although in any real system, the physical parallelism is limited to the number of processors. If a parallel computation is viewed as a collection of tasks, its parallel structure can be represented directly as a task graph. A large number of parallel computations fall into a surprisingly small number of prototypical structures. These include asynchronous computations, multiphase computations, and transaction-processing computations. Each of these structures has its particular characteristics and its own representative task graph.

Asynchronous Structure. In an asynchronous structure, the processes work independently. Cooperation among the processes is usually achieved by communication via shared data or message passing. It is assumed that such communication does not give rise to any explicit dependencies between tasks (no process, for example, blocks waiting for a message or for a lock to be released) but may only induce overheads and resource contention. The tasks executed by the

Figure 11-22 Task Graph for an Asynchronous Structure



processes can be considered to be an independent collection, with no precedence constraints between them. Figure 11-22 shows the task graph for an asynchronous structure. Each task of type "Work" executes independently of the others. A pure asynchronous structure as shown here is an idealized one; it can approximate a computation with a large logical parallelism compared with the number of processors or a computation where a task rarely waits for the completion of others. Because of the lack of precedence constraints, this structure generally achieves better performance than other structures if other attributes, such as computational work and resource use, remain the same.

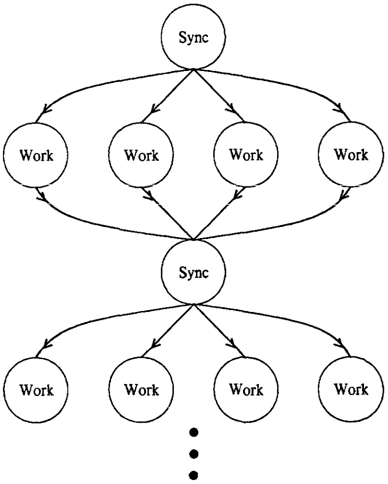
An example of an asynchronous structure is the matrix multiplication described in the previous section. While contention for global memory can have a serious impact on this algorithm's performance, the several processes create their result submatrices independently. If the calculation of each *element* of the result matrix is considered to be a task, then the tasks can be performed in any order, as there is no restriction on the order in which the elements are calculated. Other examples include the three asynchronous methods described in Section 11.5.3 for solving partial differential equations.

Synchronous Structure. In a computation with a synchronous structure, the processes execute their tasks in a lockstep fashion, using an explicit synchronization mechanism for the purpose. The task graph for a synchronous structure with a logical parallelism of four is shown in Figure 11-23. Each set of four tasks of type "Work" can execute in parallel. They are synchronized by the node labeled "Sync," which represents the action of a synchronization mechanism and can be treated as a null task with a compute time of zero.

A synchronous structure underlies other kinds of parallel structures. It is a special case of the multiphase structure we will encounter later, when the serial phase has zero compute time. Another special case of a synchronous structure is a linear synchronous pipeline, where data progresses from one process to another in lockstep under a global clock. Synchronous execution causes processor idle periods and longer execution times, unless the task execution times are nearly equal. As described on page 242, it can also lead to greater contention for resources, since synchronous tasks tend to use the same resources at roughly the same time.

Many parallel algorithms have a synchronous structure. An example is the synchronous Jacobi algorithm for solving partial differential equations, which will be

Figure 11-23 Task Graph for a Synchronous Structure



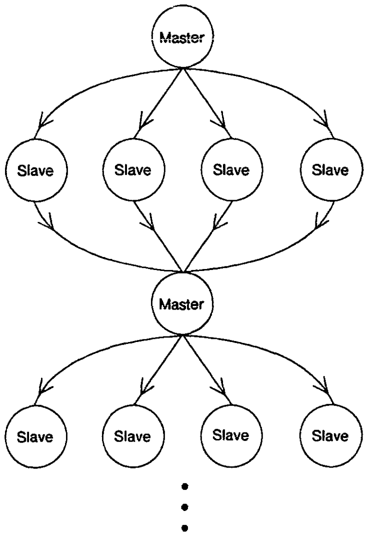
discussed in detail in Section 11.5.3. Two methods for simulating molecular motion, known as the Metropolis and molecular-dynamics methods (Section A.14), also have a synchronous structure. Section 11.5.2 compares these two methods.

Multiphase Structure. An algorithm with a *multiphase* structure is composed of a *serial phase* and a *parallel phase*, which alternate during the computation. During the serial phase, a single process is active, while the parallel phase is composed of a number of processes that execute simultaneously. Often the serial phase in some way controls the actions of processes in the parallel phase—for example, by parceling out work to them according to some criteria. In this case, the serial phase is called a *master process*, while the processes of the parallel phase are called *slaves*.

Figure 11-24 shows the task graph for a multiphase structure with a logical parallelism of four. Each iteration of the computation is composed of a serial phase and a parallel phase consisting of four tasks. The graph is identical to the earlier graph of a synchronous structure, except for the labeling of the nodes. Functionally, there is a significant difference because the tasks labeled "Master" have nonzero compute times and perform real work, in addition to serving as synchronization points.

Multiphase structures suffer from the same performance handicaps as synchronous structures. In addition, because only one processor is active when the master process is executing, processor utilization is lower. Lower utilization is directly reflected in longer execution time, and thus lower speedup, as detailed in Section 11.3.1. It is sometimes possible, however, to design a multiphase computation to avoid contention altogether, if the serial phase can copy all the global data it will need into the local memory of each parallel process.

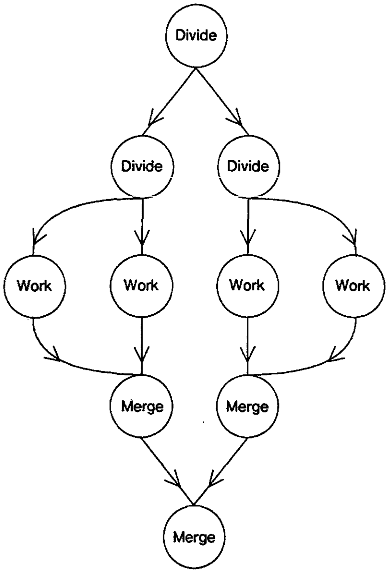
Figure 11-24 Task Graph for a Multiphase Structure



Examples of multiphase structure include the power-systems simulations of Dugan (Section A.10) and Carey (Section A.11). Parallel-phase processes select work from a global queue. When all the processors have finished this work, the serial-phase process solves a system of linear equations that the parallel phase has created. Mohan's traveling-salesman problem [Mohan 82, Mohan 83] also exhibits a multiphase structure. It uses a cost matrix. The algorithm produces successive refinements of this matrix until a least-cost solution is found. It is helpful to think of the algorithm as producing a tree of cost matrices, with each step producing two or more child nodes. Mohan studied two parallel implementations of this algorithm, the first of which exhibits a multiphase structure. The serial phase decides which child nodes are to be created, and the processes of the parallel phase create the nodes and calculate the cost of the resulting tours. This algorithm is described in more detail in Section A.18.

Partitioning Structure. In a *partitioning* structure, processes divide the work among themselves during a divide (separation) phase, do active computing on their partition during a work phase, and put their results together to reach a final solution during a merge (reconstitution) phase. Hallmarks of this structure are its diamond-shaped task graph and distinct multiple phases, all of which perform significant computation. Figure 11-25 shows the task graph for a partitioning structure with a logical parallelism of four. It has a degree of division of two; that is, each "Divide" task spawns

Figure 11-25 Task Graph for a Partitioning Structure



two other tasks during the divide phase. Usually the degree of merging (the number of tasks that must complete before the next "Merge" task can execute) is the same as the degree of division. Some algorithms, however, do not require a merge phase.

A partitioning structure may be used to implement divide-and-conquer algorithms, where the total computation needed to solve the problem can change with the degree and height of division. The computation done within the separation and reconstitution phases must be small relative to that done during the work phase, since the logical parallelism in the program develops incrementally. This structure also may be used to decompose a computation for parallel execution, even when the algorithm is not of the divide-and-conquer variety. Here, the increased speedup within the work phase that results from harnessing the power of more processors must be traded off against the extra cost of dividing and merging. Synchronization is inevitable during the merge phase and this can also be detrimental to speedup.

A partitioning structure tends to be characteristic of sorting and searching algorithms. The quicksort, implemented on Cm* by Raskin [Raskin 78] and Deminet [Deminet 82], is an example. The processors share a stack, which initially contains only one entry, a descriptor for the entire array to be sorted. Each processor attempts to pop a descriptor from the stack, and if successful, the processor partitions the task into two subsets, containing, respectively, all elements greater than or less than an estimated median value. In its simplest form, the algorithm terminates when there are no subsets of more than one element. Further details will be provided in Section 11.5.1.

The design-rule checker implemented by Lane [Lane 84] has a structure very similar to Mohan's traveling-salesman program. In the design stage, an integrated-circuit design is usually represented by a set of rectangles in a plane, with sides parallel to the coordinate axes. If two rectangles do not overlap, they must be separated by a certain distance, usually expressed in microns. Reporting violations of this rule is one purpose of a design-rule checker. To impose order on a design that consists of thousands of rectangles, the design is represented as a hierarchical collection of symbols, each of which contains other symbols. The leaf nodes of this hierarchy are rectangles. A processor selects a symbol from a global queue and parcels out to other tasks the work of analyzing its subsymbols. Thus the algorithm has the structure of a top-down tree search.

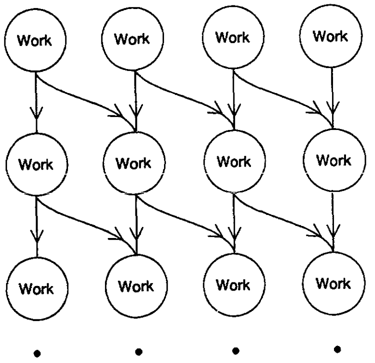
Pipeline Structure. A *pipeline* structure is so named because its processes are ordered and data moves from one process to the next, as though through a pipeline. Work proceeds in steps as on an assembly line; at the start of each step, each process (except perhaps one of them) takes its input from other processes, computes a result based on the input, and passes the result on to neighboring processes at the end of the step. A pipeline structure is possible only if the computation is composed of a sequence of operations that are applied in nearly identical fashion to successive collections of data. Even though one can envisage complex pipeline structures, the ones found in real computations are usually the simple linear variety, where processes are totally ordered and information is processed and passed from one end to the other in steps.

A pipeline structure can be synchronous or asynchronous [Chen 75]. A synchronous pipeline computation is organized around a global synchronizing mechanism, such as a clock. When the clock pulses, each process in the computation begins its next task. The synchronous task structure shown in Figure 11-23 also represents the task structure for a synchronous linear pipeline computation with four stages. The "Sync" node represents the global synchronization mechanism used at the end of each step. In an asynchronous pipeline computation, processes do not have a global clock to synchronize them; they synchronize only with their immediate neighbors using some local mechanism, such as message passing.

Figure 11-26 shows the task graph for an asynchronous linear pipeline computation with four stages, one stage represented by a vertical group of tasks. Each horizontal group of four "Work" tasks in the figure constitutes a step of the computation. The processes of a computation with this structure will be interconnected in a simple linear sequence, with communication links between neighbors; the tasks, however, have more complex interrelationships because a process must finish its current task and receive data from its neighbor(s) before commencing execution of the next task.

From a performance point of view, even an asynchronous pipeline structure is susceptible to synchronization overhead, albeit to a lesser extent. All pipeline computations pay a cost of reduced logical parallelism, both during the beginning of a computation when a pipe gradually fills (the fill-in phase) and toward the end when it empties (the flush-out phase). In addition, a pipeline computation usually involves

Figure 11-26 Task Graph for a Pipeline Structure



considerable communication of data between processes at each step, which can exact a high cost on architectures with a high interprocess communication cost.

The systolic molecular Metropolis algorithm, implemented by Whiteside [Whiteside *et al.* 82, Whiteside *et al.* 83], is an example of an asynchronous pipeline algorithm. It calculates the static and dynamic properties of a collection of molecules, given the microscopic interactions between them. Information on each particle pulses from one processor to another in the course of calculating the binding energy. More complete details can be found in Section A.14.

Transaction-Processing Algorithms. Most parallel programs that do not fall into one of the above categories can be loosely classified as *transaction processors*; requests arrive at the task force from an external source and are fielded by one of the parallel processes. It usually is important to maintain the consistency of global data, and frequently a major issue is how synchronization of access can be accomplished efficiently. Because of the wide variety of structures that fall into this category, it is not possible to draw a paradigmatic diagram.

Deminet's railway-network simulation [Jones and Gehringer 80, Deminet 82] is a parallel program in which each process represents a railroad station. Simulated trains pass from one station to another; stations service trains in the (simulated) order in which they arrive. The simulation is driven by a random-number generator, which determines the arrival times for the trains. Consequently, it is possible to consider the random-number generator as an external input to the simulation, with the simulation processing the transactions it generates. As explained in Section A.9, there are many synchronization points in the simulation, although there are no *global* synchronization points. The simulation does not have a pipeline structure because data moves between processes in a random, rather than a predetermined, pattern. Thus the simulation is not an instance of any of the structures described above.

Robinson [Robinson 82] implemented a transaction-processing system for a database on Cm*. The system consisted of as many as 11 processes:

- A master, which supplied the transaction processors with work.
- Up to 8 transaction processors, which performed the queries delivered to them by the master.
- A concurrency-control process, responsible for synchronizing *read* and *write* accesses to the database.
- A global memory manager, which maintained a mapping from each object name to the most recent version of the object.

While this structure includes both a master process and slave processes, it does not have a multiphase structure because the master and the slaves can be simultaneously active. As implemented, the transaction-processing system models a real-world system in which requests arrive from outside users; for the purposes of the experiment, however, transactions are created by a random-number generator.

11.4.2. Algorithm Structure and Overhead

Having completed our survey of the structure of parallel algorithms, it is instructive to consider which structures are susceptible to each kind of overhead identified in Section 11.3. This discussion is summarized in Table 11-2. From the preceding discussion, it is possible to draw tentative conclusions about structure/overhead relationships, although in view of the fact that only a few dozen algorithms have been studied, the conclusions may be somewhat anecdotal.

Decomposition and reconstitution overhead are associated mainly with partitioning algorithms, which have significant divide-and-merge phases. In a multiphase algorithm, the serial phase often performs part of the work of decomposition, thus

Table 11-2 Susceptibility of Various Algorithm Structures to Different Types of Overhead

	Algorithm penalty		Implementation penalty		Alg. / Impl. interaction	
Algorithm structure	Decomp. overhead	Reconst. overhead	Access overhead	Content. overhead	Synch. overhead	Parallel. overhead
Synchronous			x	+	+	x
Asynchronous			x	−		x
Multiphase	−	−	x	−	x	x
Partitioning	x	x	x	−	x	x
Pipeline			x	−	x	x
Transaction			x	x	x	−

Legend:
(space) Algorithms of the given structure do not suffer from this type of overhead.
x Algorithms of the given structure are susceptible to this type of overhead.
− Algorithms are susceptible to this type of overhead but to a lesser extent.
+ Algorithms are susceptible to this type of overhead to a greater extent.

multiphase algorithms also may manifest decomposition overhead. These two forms of overhead are less characteristic of multiphase algorithms than partitioning algorithms; hence a "–" has been placed in the table.

Shared memory is characteristic of multiprocessors (as opposed to computer networks). All parallel algorithms use it—even message passing requires mailboxes or ports that are accessible to both sender and receiver. Thus whenever shared memory is slower than local memory, all multiprocessor algorithms experience access overhead. For the same reason, all parallel algorithms encounter memory contention. It is most serious (a "+" in the table) in the case of synchronous algorithms, where all processors are likely to access common memory at about the same time. Asynchronous algorithms confront less contention, since contention delays some processors more than others during the first iteration and they tend to make global accesses at different times during each iteration thereafter. Most other algorithm structures can take measures to avoid contention overhead. For example, the serial phase of a multiphase algorithm can distribute some data to the parallel processes in advance, thus reducing contention. The decomposition phase of a partitioning algorithm accomplishes the same purpose, while the reconstitution phase tends to reduce contention in gathering results. In pipeline algorithms, each portion of data tends to be accessed by one processor at a time, although it is accessed by all processors at some time during the computation. Transaction algorithms are less capable of taking countermeasures because their sequence of input requests, and hence their memory-reference patterns, are unpredictable.

Synchronization overhead is most serious in synchronous algorithms, which have a synchronization point at the end of each iteration. Asynchronous algorithms have no internal synchronization points and consequently experience no synchronization overhead. The other four algorithm types may have some synchronization points, but in general these are less frequent than in a synchronous algorithm. (A synchronous pipeline is an exception.) Algorithms of any type can be reimplemented to take better advantage of a particular multiprocessor architecture and thus can experience parallelization overhead. Ideally, of course, the parallelization penalty should be more than offset by reduced overhead of other types. A transaction-based algorithm is perhaps the most difficult to restructure, owing to the unpredictability of its reference patterns.

11.5. Case Studies of Influences on Performance

Section 11.3 considered factors that influence performance, without reference to specific parallel algorithms. Indeed, the experimental results reported there were gathered by simulation of parallel algorithms with certain characteristics rather than by running actual algorithms. Cm*'s experimental annals contain examples of each type of overhead. Let us now consider some specific cases.

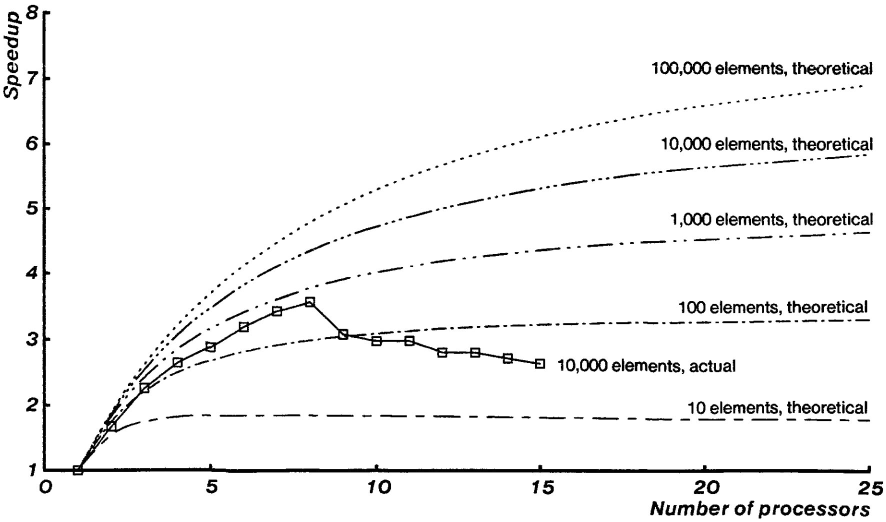
11.5.1. Algorithm Penalty

Separation Overhead. Quicksort is an algorithm that exhibits significant separation overhead. It works by partitioning a set of numbers into two subsets, according to whether each element is larger or smaller than some "median" value. Two processors can then work in parallel on partitioning the two resulting subsets, and so forth. The partitioning of a single set must, however, be performed by a single processor. Hence the time it takes to sort the set is dominated by the time to partition the largest subsets, and this time grows linearly with the size of the data. The separation overhead limits the speedup to the value given by this equation:

$$\frac{1}{S} = \frac{1}{N} + \frac{2 - \frac{\log_2 N}{N} - \frac{2}{N}}{\log_2 M}, \tag{3}$$

where S is the speedup, N is the number of processes, and M is the number of elements sorted. In fact, as shown in Figure 11-27, memory contention and other artifacts of the Cm* implementation of this algorithm prevent it from attaining even this speedup, especially when the number of processors exceeds eight. The curve marked "actual" is from a set of experimental measurements of the algorithm made by Deminet on a 10,000-element array. Other measurements of the same algorithm are described in Section A.8.

Figure 11-27 Theoretical Speedup of Quicksort for Different Array Sizes

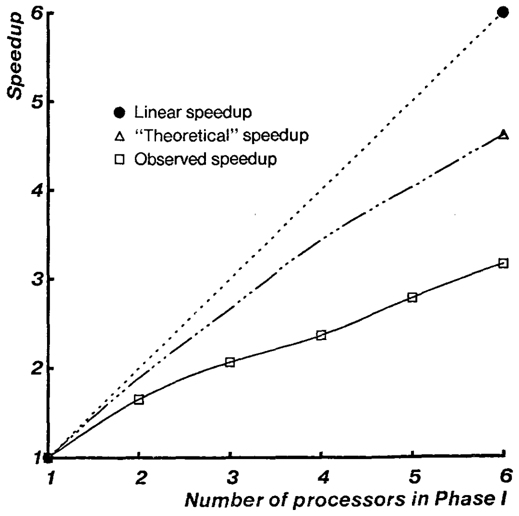


Reconstitution Overhead. The quicksort is a partitioning algorithm, an example of the structure that is most susceptible to separation and reconstitution overhead. Multiphase algorithms also can encounter these overheads, however, as happened in the power-systems simulation algorithm run on Cm* by Dugan [Dugan *et al.* 79, Durham *et al.* 79]. The simulation employed the *network model* to represent the electrical network hierarchically. The network is composed of a set of devices, each of which may be made up of more primitive devices. A nonprimitive device is called a *macrodevice*. Any primitive device or macrodevice may be characterized by the behavior of voltage and current values at its terminals. The mathematical model for a device is called its *macromodel*.

The algorithm performs two steps iteratively. Phase I, the parallel phase, solves the macromodels. Each processor repeatedly extracts an unprocessed device from the *pool* of unprocessed devices. The new voltage and current values for the device's output terminals are computed from the corresponding values for its input terminals, according to the macromodel. The resulting voltages and currents form part of a single linear system. When all the devices have been processed, Phase II utilizes a single processor to solve the linear system. The Phase I / Phase II cycle computes the results for a single *time step*. Then the cycle is performed again for the next time step and continues until the simulation is finished.

Phase II can be termed the "reconstitution phase." Because it is serial, the speedup of the entire algorithm is less than linear. Figure 11-28 plots a linear speedup, the "theoretical" speedup obtained by taking into account only the reconstitution penalty and the observed speedup on Cm*, which also exhibits degradation due to contention. The algorithm and results are described in greater detail in Section A.10.

Figure 11-28 Speedup of Power-Systems Simulation



11.5.2. Implementation Penalty

Access Overhead. Access overhead is directly related to the speed of global memory. More precisely, it depends on the speed of global memory relative to processor speed. Given the structure of Cm*, nonlocal (intracluster) references take at least three times as long as local references. It is not possible to speed them up, but the same effect can be obtained by slowing down the processors—for example, by inserting no-ops into the code.

One of the first experiments performed on Cm* was the measurement of Fast Fourier Transforms [Fuller *et al.* 77]. The same algorithm was run several times, each time with a different number of processors or a different effective processor speed. As processor speed fell, access overhead effectively decreased, resulting in a rise in speedup (Figure 11-29). More detail is provided in Section A.6.

The same phenomenon is illustrated by two algorithms for simulation of molecular motion [Ostlund *et al.* 82a, Ostlund *et al.* 82b]. These algorithms simulate the behavior of a system of particles (50, in this case) by computing the binding energy for each particle. One of these algorithms, the molecular-dynamics algorithm, shows nearly linear speedup in the case of matching parallelism, despite the fact that synchronization points were quite frequent. The other algorithm, the Metropolis algorithm, experiences more synchronization overhead and achieves somewhat poorer speedup. Both algorithms, however, make heavy use of floating-point operations, which an LSI-11 performs in software. For such algorithms, t_p tends to be very large, resulting in a fairly large x and thus low access overhead.

While t_p cannot be diminished except by changing the architecture, it is possible to *simulate* a lower t_p by replacing the floating-point calculations with delay loops. Of

Figure 11-29 Speedup of Fast Fourier Transform, Varying Effective Processor Speed

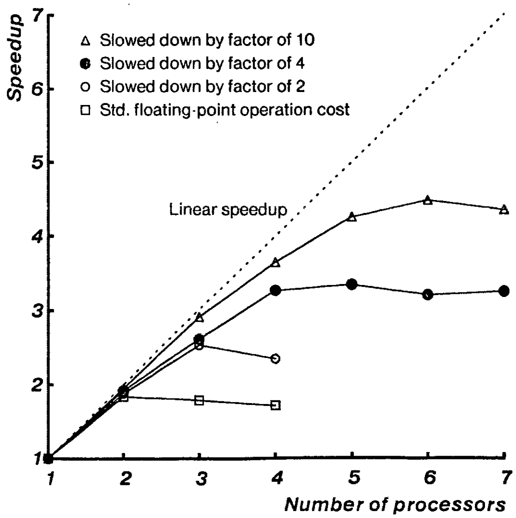
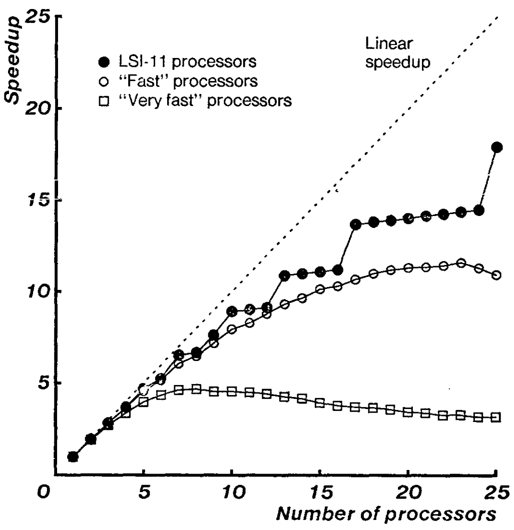


Figure 11-30 Effect of Processor Speed on Speedup of Metropolis Algorithm

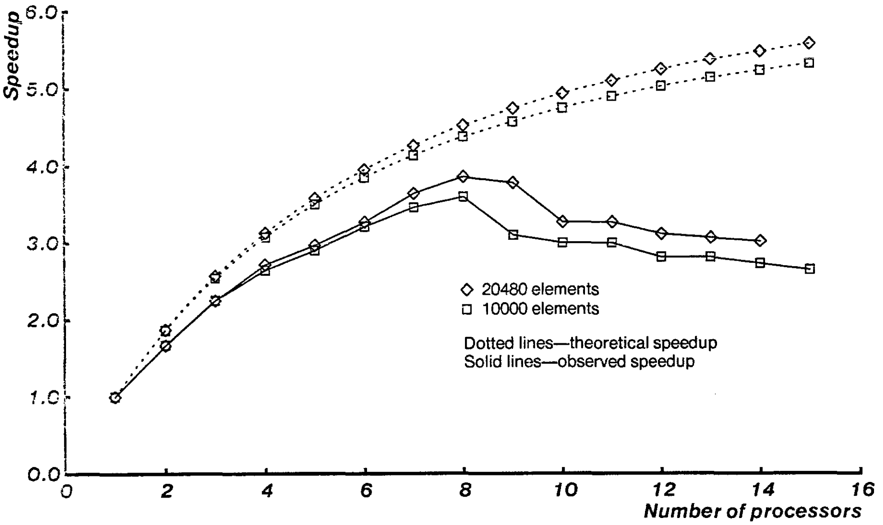


course, this renders the code useless for predicting molecular motion, but we are interested in the performance of the algorithm, not in its results. By varying the length of these delay loops, it is possible to simulate the effects of different processor speeds. (More detail can be found in Section A.14.) Figure 11-30 illustrates that speedup is inversely correlated with processor speed, due to the direct correlation between processor speed and access overhead. It also shows how increasing access overhead obscures the effect of matching parallelism, as synchronization overhead loses its leading role in determining execution time.

Contention Overhead. Access overhead and contention overhead go hand in hand. Algorithms that manifest one tend to manifest both. One sure indication of contention is declining speedup as the number of processors is increased. This phenomenon is especially obvious in the quicksort algorithm (Figure 11-31). Although separation overhead makes the theoretical speedup sublinear for all *N*, another factor clearly dominates as *N* increases past eight or nine. Here, adding more processors actually caused a decrease in speedup. Contention for shared memory is the culprit, as discussed at greater length in Section A.8.

One striking indication of the effect of contention is found in Deminet's distributed-data experiment with the partial differential equation solver PDE (Section A.7). Figure 11-32 graphs speedup for two different configurations of global data. In one, called the centralized data experiment, the global data in a 150-by-150 grid is stored in a single cluster. In the other, called distributed data, the grid is distributed among all the clusters whose processors are in use. As the number of processors increases past some threshold, speedup of the centralized version falls. This is due to contention, not only for memory but also for Kmap contexts, resulting from the large

Figure 11-31 Declining Speedup from Quicksort Algorithm



number of processors that are attempting to access the same destination Kmap simultaneously. Speedup continues to rise with the number of processors in the distributed-data version, despite the fact that the number of nonlocal references (both intracluster and intercluster) is approximately the same as in the centralized-data experiment. More details, including an explanation of the dip in the distributed-data curve, can be found in Section A.7.

Experiments with Lane's design-rule checker (Section A.20) also illustrate the effect of contention. All processors share a single queue of work to be performed, not unlike the shared stack of the quicksort. Global data is replicated in each cluster. By comparing the average number of busy processors with the speedup, it is possible to estimate the overhead due to contention and manipulation of the common queue (Section A.20 explains how this is done). Most of the overhead turns out to be due to contention for the global data. Figure 11-33 shows that the overhead rises sharply until a cluster boundary is crossed and the data is replicated. The slight increase with larger numbers of processors is evidently due to intercluster queue manipulation; although this effect is less significant, it is still noticeable.

11.5.3. Algorithm / Implementation Interaction

Synchronization Overhead. How much does synchronization cost? As a case in point, consider the algorithms for solving partial differential equations. (A more detailed description is given in Section A.1.) The objective is to solve Laplace's partial differential equation (PDE) with given boundary conditions (Dirichlet's

Figure 11-32 Effects of Distributed Data, Using Smap Microcode

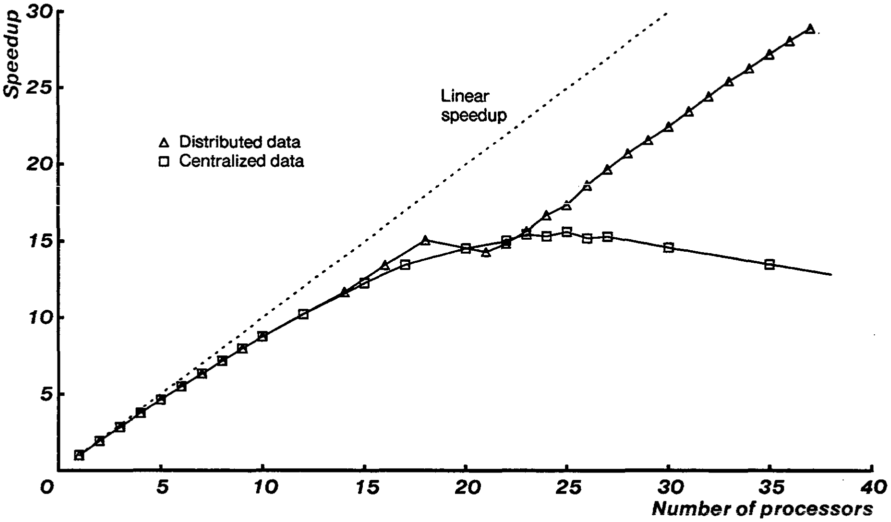
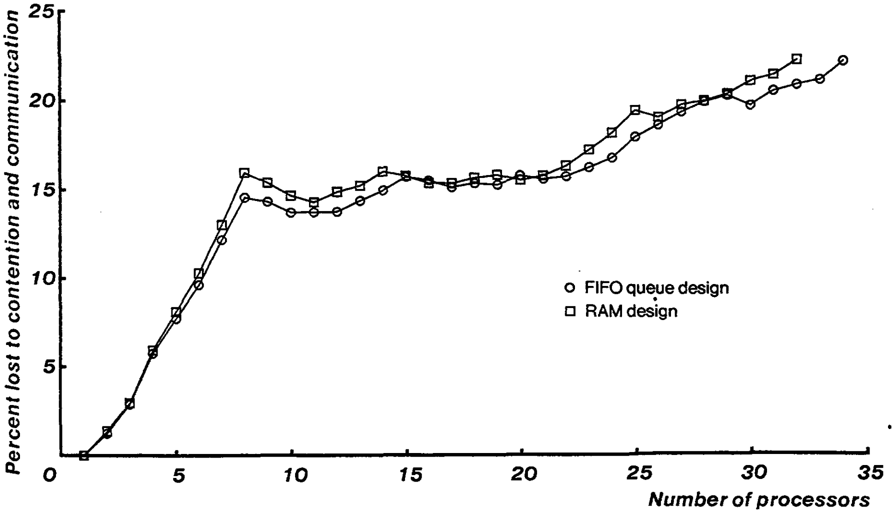


Figure 11-33 Overhead of Contention and Communication in Lane's Design-Rule Checker



problem) by the method of *finite differences*. A large matrix, or *grid*, is solved by several processes, each iterating on fixed, disjoint, equal-sized partitions of the grid until convergence is obtained. Each process runs on its own dedicated processor; the work that it performs will be called a *task*. Thus the processes are distinguishable.

Four different variations [Baudet 78] of this algorithm have been implemented on Cm*:

1. *Jacobi Method.* At the beginning of each iteration, a processor retrieves its partition from a global array. New values are computed for each element of the partition, then—within a critical section—stored back into the global array. The processor then checks to see whether its iteration has converged. If so, it reports that it has finished; otherwise it blocks until the other processors have completed the current iteration. Iterations are performed until all processors have finished.
2. *Asynchronous Jacobi Method.* This method is the same as method 1 except that a processor does not wait for the other processors to finish before starting on the next iteration.
3. *Asynchronous Gauss-Seidel Method.* This method is similar to method 2, except that the processor uses newly computed values as soon as they are available instead of the values known at the beginning of the iteration.
4. *Purely Asynchronous Method.* To compute new array values, this method uses the most recent values of all components by reading them directly from the global array and writing the updated values back to the global array (without any critical sections or synchronization).

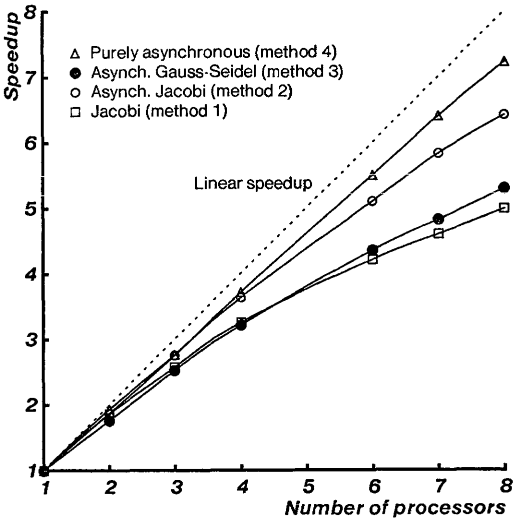
Table 11-3 summarizes the differences between the methods.

Raskin [Raskin 78] compared these algorithms using a 21-by-24 array (504

Table 11-3 Comparison of Multiprocessor Methods for Solving PDE

	All processors synchronize at beginning of each iteration	Computes only with data known at beginning of iteration	Copies data from global array to local array and back again
Jacobi (Method 1)	Yes	Yes	Yes
Asynchronous Jacobi (Method 2)	No	Yes	Yes
Asynch. Gauss-Seidel (Method 3)	No	No	Yes
Purely Asynchronous (Method 4)	No	No	No

Figure 11-34 Comparison of Speedup for Different Methods of PDE



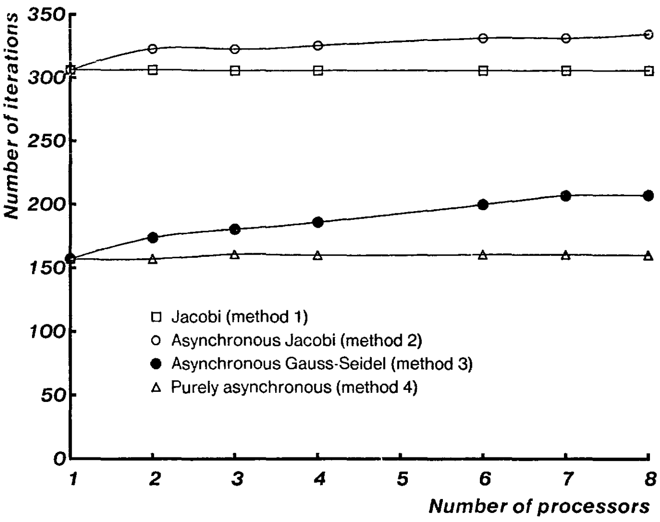
elements) on a one-cluster Cm* system with a maximum of eight processors. The results are shown in Figure 11-34. With eight processors, method 1 yields a speedup of just over 5.0, method 3 gives a speedup of better than 5.3, method 2 yields 6.4, and method 4 yields almost 7.3.

A broader perspective on the impact of synchronization can be gleaned from examining the number of iterations taken by each method. This data is taken from a C.mmp [Baudet 78] experiment similar to those on Cm*. No similar measurements have been made on Cm*. With method 1, the number of iterations is constant and lower than with method 2. In one sense, this method requires less computation. If all the processors took exactly the same time for each iteration and the time to copy data from and to the global array could be neglected, it would be faster. In this algorithm, as in several others, the beneficial effect of synchronization is its tendency to decrease the amount of computation; its detrimental effect is that it requires some processors to be idle some of the time. For most algorithms studied on Cm*, the costs of synchronization exceed the benefits.

All the asynchronous PDE methods require more iterations as the number of processes increases. The uniprocessor method 3 is almost twice as fast as the uniprocessor method 2. But as processors are added, method 3's iterations increase faster, and it shows less speedup. If the trend continued, for some number of processors, method 2 would be faster. But that point was not reached in this experiment. By contrast, method 4 takes fewer iterations than methods 2 and 3. As processors are added, it widens its lead in execution time.

Disregarding the effects of iterations on execution time, let us consider the effect

Figure 11-35 Number of Iterations Required to Solve the PDE



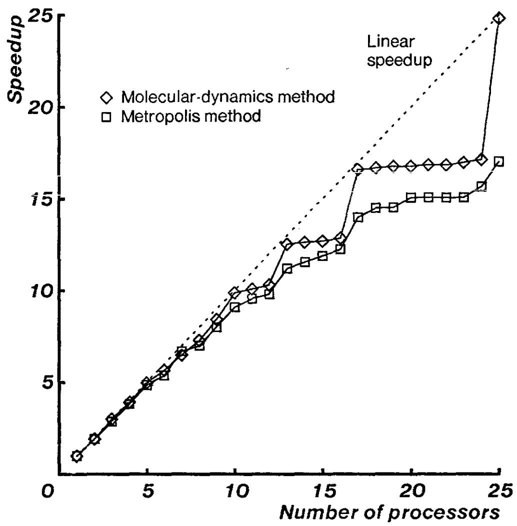
of synchronization on speedup. The asynchronous Jacobi and Gauss-Seidel methods are the only two that demand precisely the same amount of synchronization (critical sections at the beginning and end of an iteration, no waiting for other processors). Each time the synchronization requirements are decreased—from method 1 to methods 2 and 3, and then again to method 4—speedup improves. The degree of synchronization seems to be the dominating factor affecting speedup of the PDE.

The impact of synchronization is also illustrated by a comparison of the two molecular-motion simulations first mentioned in Section 11.5.2. The Metropolis method employs a method known as *ensemble averaging*. It consists of several passes, during each of which the location of each particle in the simulated structure may be perturbed. To determine whether a particle moves or not, its binding energy must be calculated. This is an $O(N)$ calculation, where N is the number of particles in the system. When K processors are used, the complexity of this step is reduced to $O(N/K)$. After this step, the contributions calculated by this step must be added before the next move can take place. This is a global synchronization point.

The molecular-dynamics algorithm uses time averaging. An initial set of velocities for the particles is calculated. These velocities can be used to predict the locations of the particles at a later time. The computation of the binding energy in the system has complexity $O(N^2)$, but since the particles move simultaneously, this serial computation may be converted to a K-processor parallel computation of $O(N^2/K)$ without any internal synchronization points. Both these algorithms are described in more detail in Section A.14.

Because it requires synchronization only every N^2/K steps, the molecular-

Figure 11-36 Speedup of Metropolis vs. Molecular Dynamics Algorithm

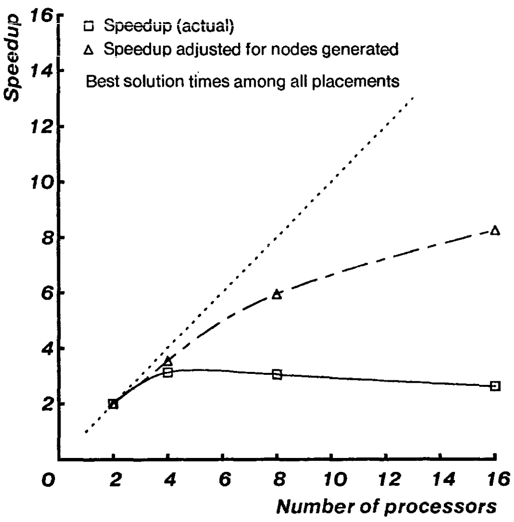


dynamics algorithm should outperform the Metropolis algorithm, whose synchronization points occur after every *N* / *K* steps. Figure 11-36 shows that this does happen. Where matching parallelism is achieved, the molecular-dynamics algorithm exhibits nearly linear speedup. The zigzag shape of the curves in this graph corresponds closely to the idealized step function of Figure 11-17. This zigzag shape is characteristic of pure synchronization overhead and disappears when access overhead begins to play a more important role (see page 263).

Parallelization Overhead. Mohan's traveling-salesman problem is an example of an algorithm that uses heuristics to speed up a search. One implementation of the algorithm, called TSP1 (Section A.18), selects a set of *edges* to search in its serial phase (page 253), and parcels out the resulting work to individual processors. As more processors cooperate to solve the problem, a larger set of edges must be selected at one time, and more work must be performed by the system between applications of heuristics. This causes the heuristics to become less effective, so more total work is done by the system. The extra work is a good example of parallelization overhead. Using up to four processors, the extra work is more than offset by the increased parallelism (Figure 11-37), but with more than four processors, parallelization overhead begins to dominate, and speedup declines.

The dashed curve in Figure 11-37 represents the speedup the program exhibits *in generating nodes* on which the processors can work. As can be seen, the program generates nodes more quickly as the number of processors increases. In essence, this would be the speedup of the system if there were no parallelization

Figure 11-37 Speedup of TSP1



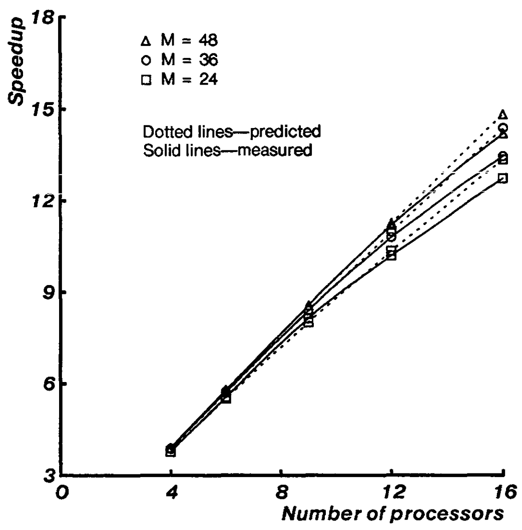
overhead. Thus the distance between the solid and dashed curves represents parallelization overhead. The distance between the dashed line and linear speedup is due largely to synchronization overhead.

Another example of parallelization overhead is the extra work involved in copying matrices before multiplying them in an effort to avoid contention. One experiment performed by Vrsalovic (Section C.5) shows that the model has accurately predicted this overhead. The ratios between access time and processing time were measured, along with the extra processing time required to make local copies of the *M-by-M* global matrices. The model was explicitly extended to account for loop initialization times. The extended model predicted execution times that were everywhere within 5 percent of the observed times (Figure 11-38).

11.6. Summary

This chapter focuses on the theoretical and practical lessons in parallel algorithm performance gleaned from experimentation on Cm*. Most of the performance studies use speedup as the metric to compare implementations of different algorithms. Speedup, which is the ratio between the elapsed time required by the one-processor version of a parallel algorithm to the elapsed time for its *N*-processor counterpart, ordinarily has a value between 1 and *N*. For some algorithms, speedup may attain a value greater than *N*, but it is more common for speedup to be substantially lower than *N*. Two issues are of interest: first, which factors are responsible for this behavior, and second, what number of processors will maximize the speedup of

Figure 11-38 Speedup of Local Copies Version of Matrix Multiplication



a given algorithm. The first question can be answered by dividing the factors into three categories, namely algorithm penalty, implementation penalty, and the interaction between the two.

The algorithm penalty is composed of the separation overhead (cost of process decomposition and data partitioning) and the reconstitution overhead (cost of the interchange and reporting of intermediate and final results). On the implementation side, the access overhead (cost of accessing shared resources) and the contention thereby induced are the main factors influencing the speedup. The interaction between the algorithm and implementation leads to two other types of overhead: the overhead of synchronization and the cost of adapting a parallel algorithm to a specific architectural implementation. This chapter has presented a detailed analysis of these factors, based on Cm* experiments and theoretical models derived or calibrated from the experimental data.

The second question, concerning the maximum number of useful processors for executing a parallel algorithm, also draws its answers from theoretical models derived from the experimental data. Such models come in two "flavors." First, at the macro level we have divided parallel algorithms into six classes: asynchronous, synchronous, multiphase structure, partitioning structure, pipeline, and transaction processing. These classes of parallel algorithm structures represent distinct cross-sections of the algorithm-overhead and implementation-overhead profiles. Hence once a parallel algorithm has been classified into one of the six categories, one could predict how it will perform. Further, we have parameterized the characteristics of a parallel architecture to predict the performance of a parallel implementation.

Finally, we have shown how this taxonomy and modeling is reflected in a number of case-study experiments on Cm*. This section contains a comprehensive set of conclusions derived from the Cm* experiments and attempts to generalize their results. In Chapter 12, we shift our concentration from the algorithm to the architecture and explore several experiments where Cm* has been used to provide insights into the performance of parallel algorithms on different types of multiprocessor architectures.

Acknowledgment. The descriptions of task graphs and algorithm structures in Section 11.4 have been taken from Chapter 2 of [Mohan 84].

12. Experiments in Multiprocessor Architecture

As indicated in Chapter 1, a major challenge facing multiprocessor-system designers is to demonstrate that a broad class of applications can profitably exploit the parallelism of a multiprocessor. Chapter 11 has provided a taxonomy of parallel applications and identified a set of factors that limit the parallelism that can be realized. Further details and examples are included in Appendix A. In addition, Appendix C has developed an analytical model to predict the impact of these limiting factors. Of equal importance is the characterization of the limits of the system architecture. Hardware-imposed limits were discussed in Chapter 3. This chapter explores limits imposed by the Cm* virtual machine.

To quantify performance accurately in a multiprocessor, it is necessary to read a system clock with a high degree of precision. Section 12.1 develops a methodology for determining clock accuracy and compensating for delays due to system load and contention for shared resources. The clock-reading software is utilized by subsequent experiments to ensure accurate measurements. Section 12.2 explores virtual-machine extensions for improving reliability. Software voting can harness the inherent redundancy of a multiprocessor so that component failures can be tolerated, provided that the overhead of voting is not too great.

Finally, extensions to the IIE (Chapter 10) can help simulate a variety of multiprocessor interconnection structures and measure the performance of an application running on each structure. Section 12.3 explores the consequences of varying hardware interconnection structures while holding the virtual-machine interface constant.

12.1. Measurement of Time

Software methods that rely only on a high-resolution clock are more commonly used to measure the performance of computer systems than are methods involving the use of special-purpose hardware monitors. Software methods are not only cheaper but also more flexible: They simplify the automation of performance evaluation, from data collection to data reduction, and they permit performance analysis to be done remotely, without probing the internals of the machine. The disadvantage of software methods is that they may be inaccurate because of interactions between the measuring and measured software.

Clock readings may be inaccurate when the system clock is used as a time source because the system load may affect how long it takes to read the clock; this leads to readings that are neither accurate nor reproducible. The problem is most serious in a distributed system where many processors can attempt to read the clock simultaneously and where message traffic influences communication delays. In Cm*, for example, a preliminary study showed that the result of a clock read can be in error (i.e., outdated) by as much as 2 milliseconds. Thus it is desirable to develop methods for measuring time more accurately.

12.1.1. Clocks in a Multiprocessor

One technique for providing access to a global time source is to have a globally readable clock with a communication delay that is small (compared to the clock resolution) and fixed, regardless of system load. Such a clock requires a special bus allowing multiple simultaneous read accesses. An example of such a bus structure is the interprocessor bus of C.mmp [Wulf *et al.* 81]. C.mmp has a 56-bit global clock with a 4-μs. resolution. The value of this clock is continually broadcast on the interprocessor bus.

In a more loosely coupled system, however, it is not feasible to devote a special-purpose bus to the global clock because of the amount of cabling involved and because of the problems associated with bus arbitration over long lengths of wire. Also, broadcasting the clock value on the general-purpose bus uses a large portion of the cycles available on the bus. For these two reasons, the clock value is generally not broadcast in large distributed systems.

In Cm*, when a subsystem needs to know the system time, it establishes a connection with the clock and reads its value. This way, communication occurs only when necessary. However, because the time required to establish a connection depends on bus activity and the transmission delay depends on the physical location of the processor, the total delay is unpredictable. When multiple requests for the system time arrive simultaneously, bus contention results, and a queue is formed. The wait time in this queue adds further uncertainty to the total communication delay. This section examines the problems in reading a central clock, studies their effect on the performance of the Cm* clock,' and then proposes schemes to obtain more accurate clock readings.

Cm* Clocks. Cm* provides three 32-bit real-time clocks for time measurements, one of which is used to provide the system time. These clocks have a quartz-crystal time base with an adjustable resolution. The maximum resolution is 0.5 μs. The clocks can be zeroed under program control for interval measurements. All the clocks are hardwired to give a resolution of 2 μs. This yields a maximum range of $2^{32} \times 2$ μs., or 2.386 hours. The clocks are connected as peripherals to Cm 3 on cluster 1, Cm 4 on cluster 2, and Cm 14 on cluster 5.

Because the LSI-11 uses memory-mapped I/O, the clock is read by an ordinary read to a specific location in the I/O page (page 15) of the LSI-11 address space.

Clock-Reading Routines and Their Performance. In both StarOS and Medusa, the system clock is read via remote memory references. The clock value consists of 32 bits; the data bus is only 16 bits wide, so it takes at least two memory references to read the full clock. Because the two 16-bit words cannot be read *simultaneously*, there is the danger that the second (or least-significant) word of the clock may "wrap around" in the meantime, causing an erroneous reading.

At the outset of this project, both StarOS and Medusa provided a standard routine for reading the clock. Let us call this algorithm the *varying-read* algorithm because it makes either three or four memory references to the clock register, depending on the value of the clock. Figure 12-1 shows the pseudocode for this routine. When *SecondHi* is greater than *FirstHi*, the low-order word must have wrapped around between the two reads of the high-order word. Because it is not known whether the reading of *FirstLow* occurred before or after the wraparound, a second reading of the low-order word must be taken.

A preliminary experiment was performed to evaluate the performance of the Medusa *varying-read* clock routine in order to determine the average execution time of *varying-read* and to see how the system load affected accuracy. The experiment measured the elapsed time between two successive clock-read procedure calls. On average, the elapsed time is identical to the execution time of the routine, including its remote memory references.

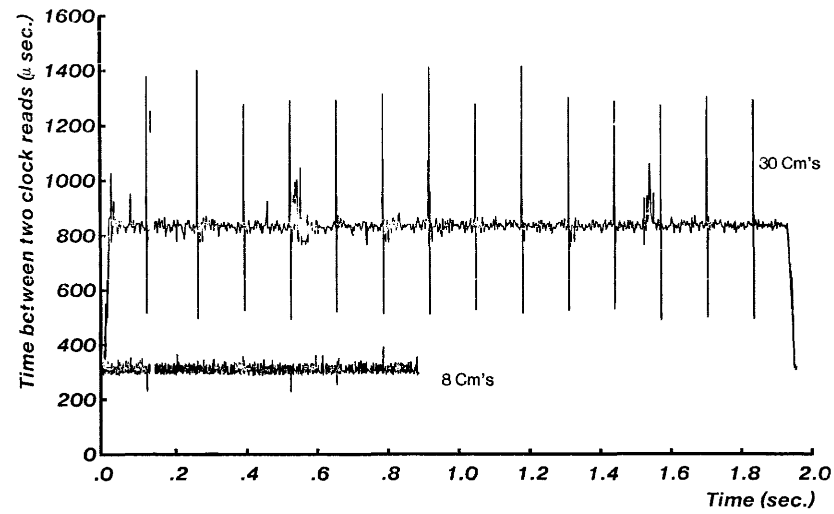
The experiment was performed with 8 Cm's in clusters 2 through 5 reading the clock in cluster 1. The experiment was then repeated with 30 Cm's, also distributed between clusters 2 through 5. The results for all processors are summarized in Figure 12-2, which plots the elapsed time between two clock reads against the time elapsed since the beginning of the experiment. The intercluster memory reference rate is approximately 90,000 references per second in the case of 30 Cm's and is approximately 53,000 references per second in the case of 8 Cm's.

Figure 12-2 reveals periodic peaks and troughs in the 30-Cm curve. The peaks occurred when the low-order word of the clock wrapped around during the second invocation of *varying-read*, causing that invocation to read the low-order word a second time. The troughs occurred when the low-order word wrapped around during the first invocation of the *varying-read* routine so that the first invocation had to read the low-order word again. Kong's report [Kong 82] explains the peaks and troughs in greater detail. Note that in the 8-Cm case, there was less contention for the clock, so

Figure 12-1 *Varying-Read* Algorithm

```
Varying-Read:

    FirstHi ← read high-order word of clock;
    FirstLow ← read low-order word of clock;
    SecondHi ← read high-order word of clock;

    if SecondHi > FirstHi then begin,
        SecondLow ← read low-order word of clock;
        return SecondHi and SecondLow as the clock result;
    end
    else begin
        return FirstHi and FirstLow as the clock result;
    end;
```

Figure 12-2 Performance of MEDUSA *Varying-Read* Clock Routine



the low-order word was much less likely to wrap around during a reading; hence the peaks and troughs did not appear regularly.

Figure 12-2 also shows that in the 30-Cm case, the elapsed time rose at the beginning of the experiment from approximately 300 μs. to more than 800 μs. and then fell from 800 μs. to approximately 300 μs. at the end of the experiment. This was because not all the Cm's started and finished simultaneously. Consequently, there was less system load and contention at both the beginning and the end of the experiment, resulting in lower elapsed times. In the 8-Cm case of Figure 12-2, no rise or fall was observed, indicating that 8 Cm's did not create sufficient traffic to slow the clock-read routines perceptibly.

Because of the *varying-read* routine's erratic behavior when the low-order word of the clock flips, two new clock-reading routines were written. The first, known as the *four-read* clock routine, was a modification of the original *varying-read* routine. It reads both the high-order and the low-order word twice and always returns the first low-order word read as the low-order word of the clock. Its execution time is essentially independent of the value of the clock readings, as shown in Figure 12-3. Here, the rate of remote memory references was 96,000 per second with 30 Cm's, while with 8 Cm's it was 64,000 per second.[1] Figure 12-4 shows the pseudocode for the routine that is referred to as the *four-read* clock routine.

The second routine reads only the low-order word of the clock and computes the value of the high-order word. It uses two static variables, *$OldLow* and *$Hi*. During a

[1] These numbers are higher than for the previous clock-read routines because four remote memory references are needed for each clock read.

Figure 12-3 Performance of MEDUSA *Four-Read* Clock Routine

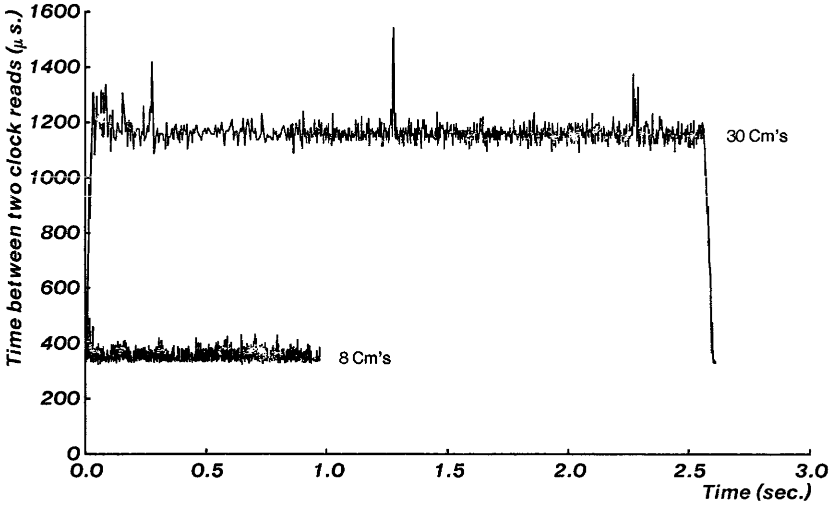


Figure 12-4 *Four-Read* Algorithm

```
Four-Read:

    FirstHi <—read high-order word of clock;
    FirstLow <—read low-order word of clock;
    SecondHi <—read high-order word of clock;
    SecondLow <—read low-order word of clock;
    if SecondHi > FirstHi then                /* clock flipped */
        if SecondLow > FirstLow then          /* flip was before FirstLow */
            return FirstLow and SecondHi as result
        else                                  /* flip was after FirstLow read */
            return FirstLow and FirstHi as result
    else                          /* no flip occurred between FirstHi and SecondHi */
        return FirstLow and FirstHi as result;
```

clock reset, these variables are zeroed. Every time the routine is called, the low-order word of the clock is read and is compared with the value of *$OldLow*. If the value of *$OldLow* is higher than the current value of the low word, a flip must have occurred, so the value of *$Hi* is incremented. Assuming that the time between two consecutive low-order word flips exceeds the interval between any two consecutive calls of this routine, then if the value of *$OldLow* is lower than that of the low word of the clock, no flip has occurred, and the value of *$Hi* remains unchanged. This routine is called the *one-read* clock routine because it reads the clock register only once. Figure 12-5 shows the pseudocode for this routine.

This routine has a constant execution time. Its performance is illustrated by Figure 12-6. Here the remote memory-reference rate was 76,000 per second for the 28-Cm case and 23,000 per second for the 8-Cm case. This lower rate of remote memory reference makes the routine execution time quite insensitive to the increasing number of Cm's reading the clock. Therefore, the differences between the two curves are much smaller than in the previous figures.

For this routine to work, each Cm must have its own local copy of *$Hi* and *$OldLow*, and each Cm must read the clock at least once in every T_f seconds, where T_f is the time between two low-word flips and is equal to $2^{16} \times R$, where R is the number of seconds between two clock ticks. With the present R of 2 μs., T_f equals 0.131 seconds.

A new set of experiments investigated the performance of both the *one-read* and

Figure 12-5 *One-Read* Algorithm

```
One-Read:

    static $OldLow;
    static $Hi;

    Low <—read low-order word of clock;
    if $OldLow ≥ Low then $Hi <—$Hi + 1;
    $OldLow <—Low;
    return Low and $Hi as the result;
```

Figure 12-6 Performance of MEDUSA *One-Read* Clock Routine

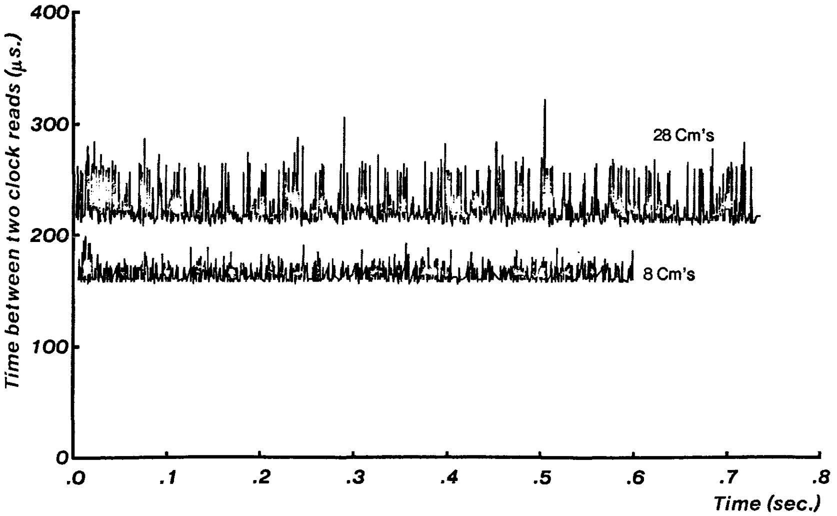
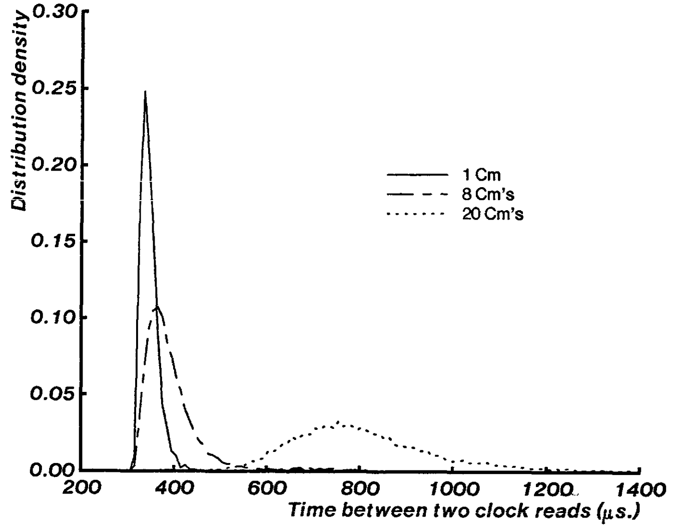


Figure 12-7 Execution-Time Histograms for StarOS *Four-Read* Clock Routine



four-read routines under StarOS and Medusa. The experiments measured elapsed time between two clock reads as a function of the number of Cm's reading the clock.

StarOS RESULTS. As increasing numbers of Cm's read the clock using the *four-read* routine, both the average execution time and its standard deviation increase. Figure 12-7 summarizes several histograms, which are normalized to give the same area under the curve. All distributions appear to be Rayleigh with a lower bound of 310 μs., which is the minimum time required to execute the *four-read* clock routine. Although it cannot be discerned in the figure, a careful study of the data shows that besides the main peak, there is a small peak between 630 μs. to 760 μs. that is due to 60 Hz line-frequency clock interrupts occurring between two clock reads.

The average execution time of the *one-read* routine is quite insensitive to the number of Cm's reading the clock because the low remote-reference rate of this clock routine does not saturate the Kmap. Figure 12-8 summarizes the histograms, normalized to give the same area under the curve. Because the average execution time varies only slightly and the standard deviation of the results remains almost constant, all the curves are similar and overlapping. Due to the line-frequency clock interrupts, there also is a small secondary peak between 470 μs. and 550 μs.

Medusa RESULTS. The minimum time required to execute the *four-read* Medusa clock routine is approximately 320 μs. The average time increases as more Cm's read the clock. Figure 12-9 is the normalized plot of the histograms. Figure 12-10 summarizes the results of the Medusa *one-read* clock routine in normalized fashion. The standard deviation of the result is extremely small, and the mean does not change significantly as increasing numbers of Cm's read the clock. This is because the load

Figure 12-8 Execution-Time Histograms for StarOS *One-Read* Clock Routine

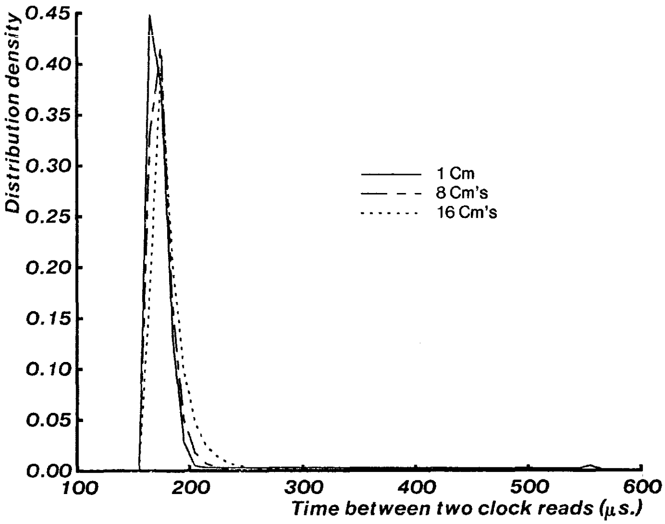


Figure 12-9 Execution-Time Histograms for Medusa *Four-Read* Clock Routine

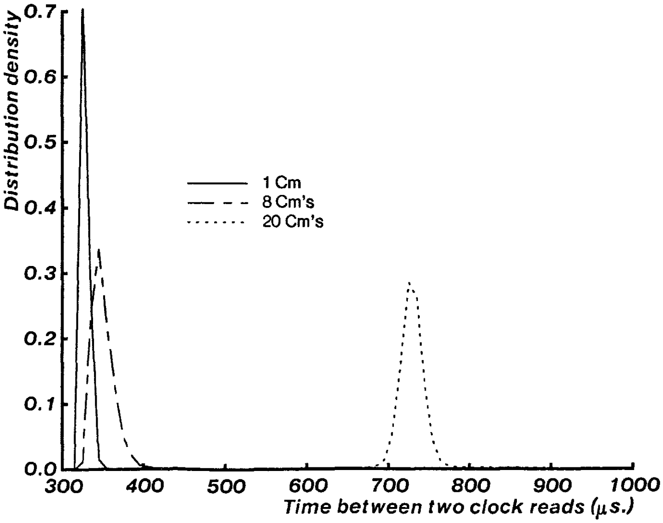
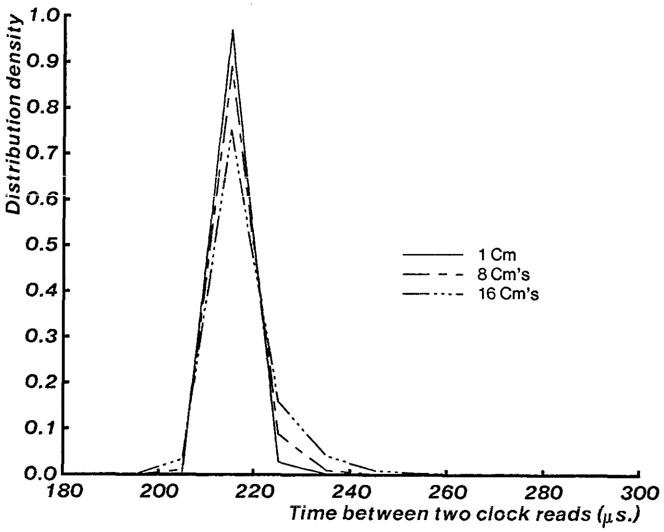


Figure 12-10 Execution-Time Histograms for MEDUSA *One-Read* Clock Routine



imposed by this routine is small enough for all clock-read requests to be processed immediately by the Kmap without waiting in the Kmap queue.

Discussion. Comparing Figure 12-7 with Figure 12-9, one sees that the average execution times are roughly the same at light load, but execution time increases faster as a function of increasing load under STAROS than under MEDUSA. The difference in the shape of the curves in Figure 12-7 and Figure 12-9 indicates that the two operating systems have very different strategies for handling memory contention (see Section 3.1.3). Even when the effects of interrupts are ignored, the STAROS results show a larger standard deviation.

The 0.5 µs. resolution of the global Cm* clock does not help to measure short time because intercluster references are subject to uncertain delays in the presence of load. The results of these experiments show that short time intervals (500 µs. or less) cannot be accurately measured using any of the clock-reading routines described so far.

Another alternative is to read only the low-order word of the clock. Then the clock can be read with a single LSI-11 instruction, and the resolution should be greatly improved. The cost imposed by this strategy is a loss of clock range. With the clock tick set at 2 µs., the low-order word provides only a 0.131-second range.

Clearly, a larger range can be obtained by increasing the clock-tick interval. If the usable resolution is limited by the uncertainty in communication delay, the tick interval can be increased without sacrificing resolution. For example, under very light system load, a MEDUSA clock word is read with a standard deviation of 2.3 µs., and a STAROS clock word is read with a standard deviation of 7.13 µs. (assuming no processor interrupts). Therefore, the 2-μs. resolution of the present clock is useful, and increasing the tick interval would sacrifice resolution. Under heavy loads, however, the standard deviation of reading a clock word can be very high, so the tick interval can be lengthened to increase range without losing accuracy. Alternatively, if accuracy is more important than range, the methods described in the next section can be used to improve it when the load is high.

12.1.2. Methodologies for Measuring Time

Is it possible to obtain more accurate time measurements by compensating for Kmap load and clock contention? This section attempts to develop methods based on *synthetic workloads* (see Section 10.3).

The synthetic workload used in these experiments can be described as follows. Assume that the execution time of a software routine R is to be measured under different system loads. Several processes, each on its own Cm, execute R simultaneously. If this experiment is repeated as the number of processes N is varied, the the execution time of R can be studied as a function of N.

An efficient way to measure performance is to designate only one process per cluster to read the clock. This lowers system load by reducing the number of clock reads, improving performance of both the clock-reading software and the measured processes. The measurements will be valid if all Cm's execute the measured process at the same speed.

Clock Compensation. Let us consider two ways of compensating for the inaccuracies in clock readings. The interval between two consecutive clock reads (a clock-read pair) is a random variable with a mean and variance that are both functions of the system load. The net elapsed time for any experiment can be computed by subtracting the average value of this interval from the measured result. Using this method, the expected value of the computed result $E(T_c)$ equals the true elapsed time T_N, while the variance of the computed result is identical to the variance of the measured result. Let us call this method the *long-term averaging technique*.

There are several ways to select the clock-read pairs that are used to adjust the measurement. These methods can be evaluated by using the concept of an improvement factor k. In any experiment that measures some time interval, let $V(T_c)$ be the variance of the corrected result and let σ^2 be the variance of the uncorrected result. Then k is defined such that

$$k = \frac{\sigma^2}{V(T_c)} .$$

The larger the value of k, the better the improvement. The range of k is between 0.5 and infinity [Kong 82]. When k is unity, the variance of the corrected result is unchanged. Note that the long-term averaging technique always yields a k of unity.

One method (called Method I) is illustrated in Figure 12-11. If the processor that

Figure 12-11 Short-Term Method I

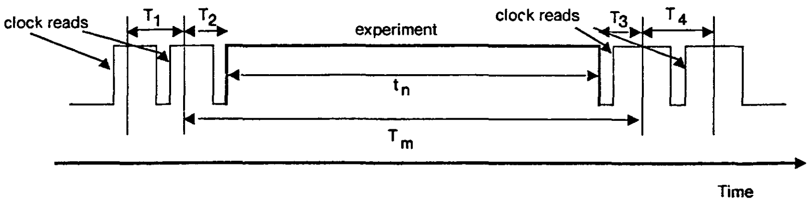
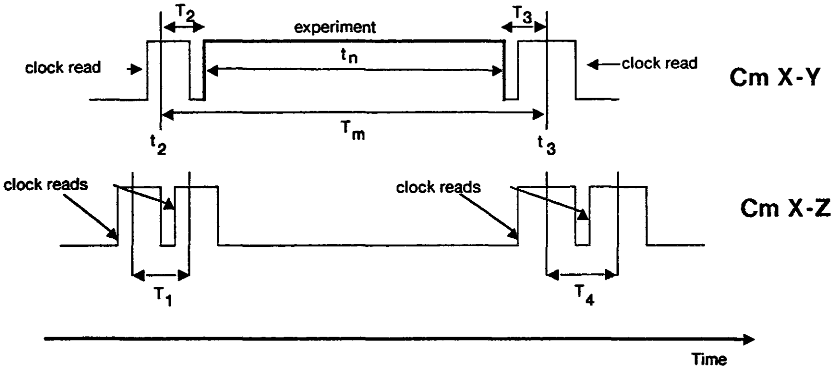


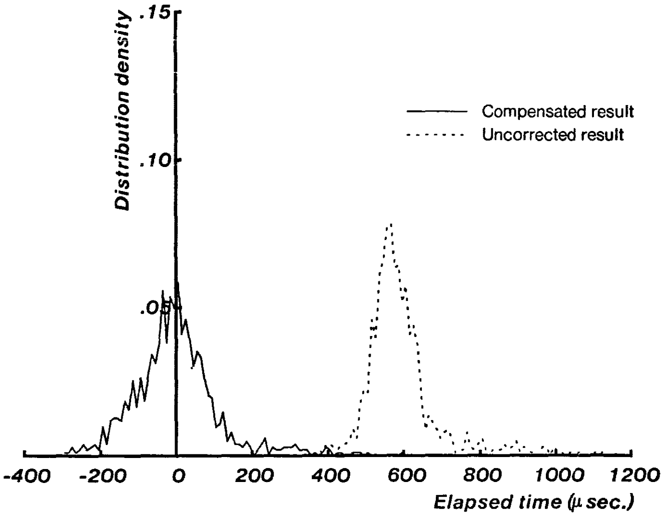
Figure 12-12 Short-Term Method II



starts the measurement reads the clock twice at the beginning of the experiment and the processor that terminates the measurement reads the clock twice after the experiment, then T_1 should correlate highly with T_2, while T_3 should correlate highly with T_4 because these clock reads occur at almost the same time. This method has the advantage that no clock read occurs during the experiment, so the performance of the experiment under measurement is not affected. If an interrupt occurs in the middle of a clock-read pair, however, the interval between the two reads will be too long, perturbing the measurements.

Another strategy, Method II (Figure 12-12), minimizes this danger. If a clock *compensating* process runs concurrently with the experiment and periodically samples the load by reading the clock twice in succession, the elapsed time between the two reads can be used for compensation. One approach is to obtain T_1 from the clock process's clock-read pair that is closest in time to the clock read that starts the measurement and to obtain T_4 from the clock-read pair nearest the clock read that stops the measurement. If a *systemwide* interrupt (such as an interrupt from the line-frequency clock) occurs, it affects both the measured process and the compensating process so the time to handle an interrupt during a clock read is compensated for. Method II does, however, induce contention for the clock, which

Figure 12-13 Measuring Zero Elapsed Time Using Method I with StarOS *Four-Read* Routine



can itself be a source of perturbation. Subsequent sections will show that it nonetheless yields quite accurate results.

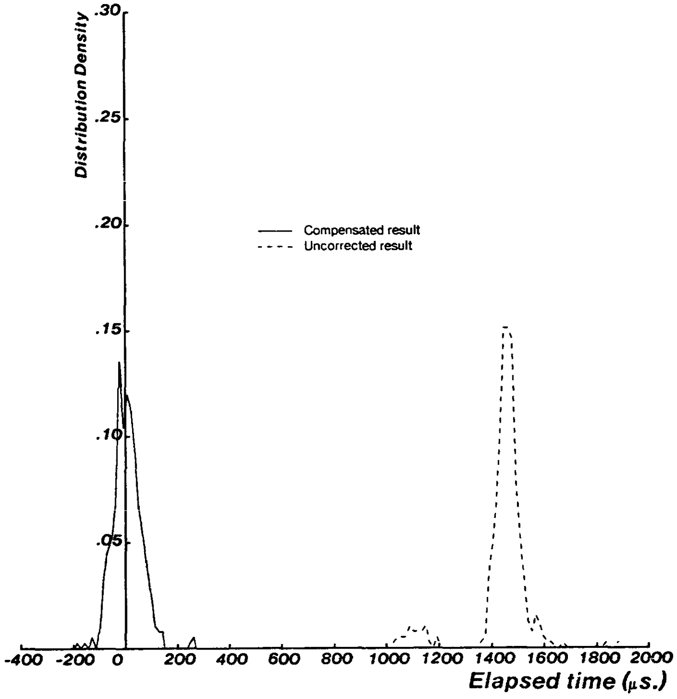
Evaluation of Method I. The experiment to validate[2] Method I consists of a process reading the clock four times in succession. The first two clock reads are used to compute T_1; the second and third clock reads measure a null experiment (which must have zero execution time); and the third and fourth clock reads are used to compute T_4. The synthetic workload is generated by distributing among the clusters a large number of processes that read the system clock.

Figure 12-13 shows the results of the experiment using the StarOS *four-read* clock routine to measure time. The solid curve is the distribution density of the compensated result, while the dashed curve is the distribution density of the result before correction is applied. The ideal result is an impulse of unit magnitude at 0 μs. The mean compensated result was −1.90 μs., and the improvement factor, k, was 0.8. Recall that for $k < 1$, the correction increases the variance of the result. The same experiment using the *one-read* clock routine gave an improvement factor of 0.81.

Figure 12-14 illustrates the result under Medusa using the *four-read* clock routine to measure time. The mean compensated result was 6.69 μs., and the improvement factor was 3.57. This represents a great improvement in the variance of the results. The *one-read* clock routine gave an improvement factor of 0.68.

[2] The methods for adjusting the clock readings employ only heuristic approaches, so they cannot be proved to produce correct results. To validate these methods, an attempt is made to show only that they produce accurate results for measuring some fixed interval under some reasonable system load.

Figure 12-14 Measuring Zero Elapsed Time Using Method I with MEDUSA *Four-Read* Routine



Evaluation of Method II. The experiment that validates Method II consists of the following:

- One process, called the *compensating* process, reading the clock in the same cluster as the process to be measured (in Cm X-Z in Figure 12-12).
- A process that does two clock reads to measure the elapsed time (in Cm X-Y in Figure 12-12).
- A number of pairs of communicating processes that exchange messages.

Each pair of these communicating processes is independent of the other processes in the system, and their sole purpose is to generate a synthetic load on the Kmaps through which clock-read requests are routed. The experiment process (in Cm X-Y) is synchronized with the compensating process. It signals the compensating process (in Cm X-Z) to start reading the clock, reads the clock twice *successively* (thus measuring a null experiment), and then sends the results of the two clock reads to the compensating process, which computes the net elapsed time according to Method II.

Figure 12-15 shows the distribution density of the results of the STAROS experiment. The dashed curve is the result of the measured reading (T_m in Figure 12-12). The solid curve is the result (T_c) after Method II has been applied. The results were

Figure 12-15 Measuring Zero Elapsed Time Using Method II with StarOS *Four-Read* Routine

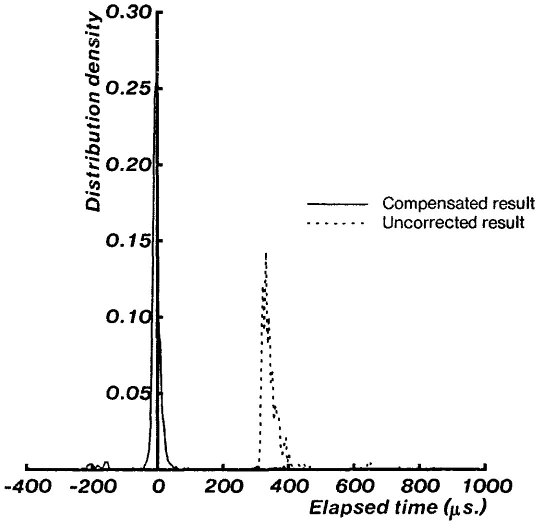
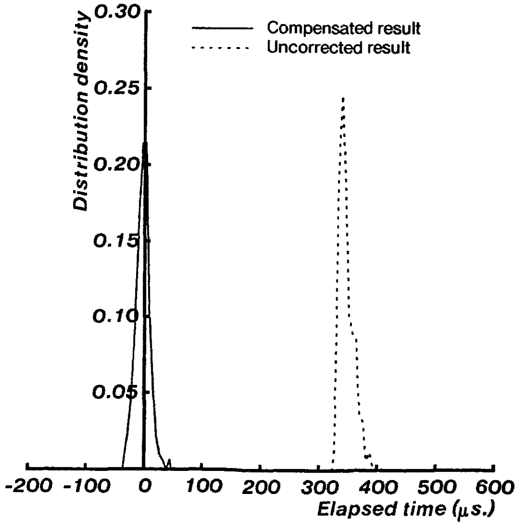


Figure 12-16 Measuring Zero Elapsed Time Using Method II with Medusa *Four-Read* Routine



derived from 1,000 repetitions of the experiment. The mean value was –5.24 μs., while the improvement factor was 1.14. The improvement factor for the *one-read* clock routine was 1.98.

When executing under MEDUSA, the experiment yielded different results (Figure 12-16). The mean value of the compensated result was 6.69 μs., and the improvement factor was 1.11. The improvement factor was 0.81 for the *one-read* clock routine.

Discussion of Results. These experiments all produce a mean corrected measurement whose magnitude is less than 6.7 μs. Measurements with greater accuracy than this value cannot be reliably obtained using either measurement method. One may conclude that these methods are not suitable for measuring elapsed times that are less than, say, 50 microseconds because the relative error for small interval measurements is high.

In four of the eight experiments, the correction improved the variance of the result, with the improvement factor ranging from 1.11 to 3.57. The other four cases showed a *k* less than 1 but greater than 0.67. Recall that the worst-case *k* would be 0.5. If the clock reads used for compensation were totally uncorrelated to the clock reads that they were supposed to compensate, the value of *k* would be 0.67 [Kong 82]. In the MEDUSA experiment using Method I with the *one-read* clock routine, the value of *k* was 0.68. This shows that during that experiment, the system load was changing so rapidly that the execution time of any clock read was essentially uncorrelated to the execution time of any previous or subsequent clock reads.

It is interesting to note that three out of the four experiments using Method II resulted in improved variance, compared with only one out of the four experiments using Method I. This phenomenon is largely due to the difference in the type of system load. In all the Method II experiments, the system workload was created by a large number of processes sending and receiving messages. Because message operations take a long time (on the order of a millisecond), the load of the system is trackable by the clock reads. Conversely, when the granularity of the system load has a duration comparable to or shorter than the time required to execute a clock read, the tracking of the system load using clock reads fails. This was the case for the Method I experiments. The synthetic workload consisted of a large number of processes reading the clock. Because the load on the system had the same duration as the clock reads used to sample the load, the tracking of the system load failed.

In other words, Method I is theoretically superior because it does not affect the experiment under measurement. Thus it was given the difficult task of executing under a system load of very small granularity. Results showed that Method I was unable to perform properly in small granular system loads. Method II was tested with a more reasonable load and was found to perform quite well. The short-term averaging technique using Method II performed better than the long-term technique would have performed in three of the four experiments.

As noted above, Method I does not track system load correctly in the presence of interrupts. This explains why Method I did not work very well under STAROS, since the STAROS processes were interrupted 60 times per second. Method II tracks well even with interrupts. Because Method I does not perturb the experiment, it should perform at least as well as Method II under a reasonable system load, provided that there are no interrupts. When there are interrupts, Method II is the preferred method.

12.2. Voting

Future computer systems will be used in environments that require increased reliability due to the nature of the tasks being performed. For example, avionics computers will replace the mechanical control of present aircraft. These computers will make thousands of decisions per second concerning the stability of the aircraft. The system must be designed so that the computer will never fail in flight, since the stability decisions cannot be made by the pilot. In many cases, the required reliability can be obtained by replicating hardware components and comparing the outputs of the components to determine the correct result. The replication allows the system to tolerate failures in components without loss of reliability.

One technique for enhancing reliability is to replicate components an odd number of times and then compare their outputs. If a majority of modules agree, then their output is deemed to be correct. The comparison process is called *voting*, and the system that compares the outputs is called a *voter*. If the hardware is replicated *N* times, and the *N* outputs are voted to discover the majority, then the system is said to have *N*-modular redundancy (NMR). For example, if the hardware is replicated three times, then the system has triple-modular redundancy (TMR).

In NMR, the redundant modules are often processor-memory pairs. The computers communicate information to be voted on either by hardware voters [Siewiorek *et al.* 78a], or by software voters running on some of the processors [Goldberg 81, Michalopoulos 82]. Software voting has a number of distinct advantages over hardware voting, one of which is the flexibility of the voter. For example, the Software Implemented Fault Tolerant computer (SIFT) [Goldberg 80] has a voter that can handle a three-way vote or a five-way vote. The system can determine which voter to use depending on the number of processors available. To improve system reliability, the software voter routine can be modified as the system changes.

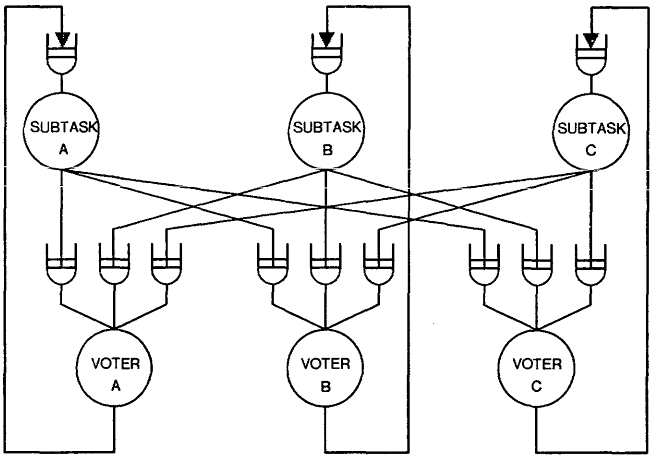
Most of the research on NMR has assumed that the modules are synchronized [Davies 78]. Because it is very difficult to enforce tight processor synchronization, this assumption is not valid for a large class of systems. Some researchers are beginning to realize that asynchronous systems offer distinct advantages in reliability [Michalopoulos 82] and simplicity, but the problem of how to make an asynchronous system meet the reliability objectives still remains.

A TMR experiment was performed on Cm* using a system consisting of six processes running on six separate processors. Three of the processes (called *subtask processes*, or simply *subtasks*) execute identical copies of the same iterative synthetic workload. Together, these processes are said to execute a single *task*.[3] The outputs of all three subtasks were simultaneously fed, one word at a time, to each of three voter processes, as shown in Figure 12-17.

More specifically, the triplicated subtasks each calculate the *i*th data value, send a copy of the data to each voter, and receive the voted value of the data from their associated voter. The new data value is then used in calculating the (i + 1)st data

[3] Do not confuse this notion of "task" with the different concept of execution time between synchronization points, as used in Chapter 10 and Chapter 11.

Figure 12-17 TMR Experimental Structure



value.[4] The time each subtask takes to calculate the *i*th data value is a random variable. This time is a function of the *granularity* of the subtask. The granularity is defined as the number of operations (each of which consists of four LSI-11 instructions) executed between votes, not counting the overhead due to voting. The granularity of the subtasks is set before each experiment.

The subtasks send each voter two data words per vote. The first word is a sequence number associating an iteration number with the data. The second word is the data to be voted. When a voter has received data from a majority of the subtasks (two), it checks to see whether the data values agree. If so, a consensus has been reached, and the value is sent to the subtask associated with the voter. Otherwise, the voter waits for the data value from the third subtask to decide on the correct value. Since each process runs on a separate processor, voting may proceed in parallel with subtask execution.

Three types of voters are used in the experiments. The first voter, called the *simple* voter, is a synchronizing voter. It has no internal storage of data from one iteration to the next. It requires the subtasks to reach a full point of synchronization after each iteration. The second voter, called the *internal-queue* voter, has an internal queue that allows it to handle data from different iterations. The subtasks are not required to synchronize fully after each iteration. This voter has been optimized for high execution speed in the average case, and thus it has the shortest execution

[4] One can imagine wanting to pass more than one data value from one subtask to the next. This can be done with a more complicated voter. The entire state of a processor (or selected parts) could be passed as data, allowing a faulty processor to recover from a transient by accepting the voted state as its new state. Adding this capability to the experiments would complicate them, without yielding additional information about the voting.

time. The third voter, called the *sequence-number* voter, uses the sequence numbers that are sent by the subtasks to order data by subtask iteration. This voter has the longest execution time.

As long as the subtasks have similar execution speeds, the voter should receive the *i*th iteration from each subtask at approximately the same time. The sequence number voter and the internal queue voter do not require full synchronization, so if one subtask is faster than the others, the voter may receive the $(i+1)$st data value from the fast subtask before the others send the *i*th data value. Since the voter now has data from more than one iteration, a queue is used to associate data values with subtasks and iterations. Each entry in the queue contains an iteration number, a boolean array telling which subtasks have delivered data for the iteration, and the data values that have arrived. The sequence-number voter adds an entry to the queue when the first data value for a particular iteration arrives. When all the data values for the iteration have been received, the voter reports an error if necessary and then removes the entry from the queue.

The voter queue has a finite maximum length. If one subtask has not sent any data to the voter in the same period in which the other two subtasks have sent many data messages, the queue could conceivably become full. The voter handles a full queue by removing the oldest entry (associated with the earliest iteration for which some data is still missing), reporting an error, and adding an entry for the new iteration number. In all experiments described here, enough storage was allocated to the queue to prevent it from overflowing.

12.2.1. *Voter-Overhead Experiments*

In any NMR system, less useful work will be done than in the corresponding non-replicated system because the overhead of voting reduces system throughput. Among other factors, the overhead includes interprocessor communication and the time it takes to perform comparisons.

York [York *et al.* 83] developed an expression for throughput (work performed per unit time) in terms of voting overhead. If

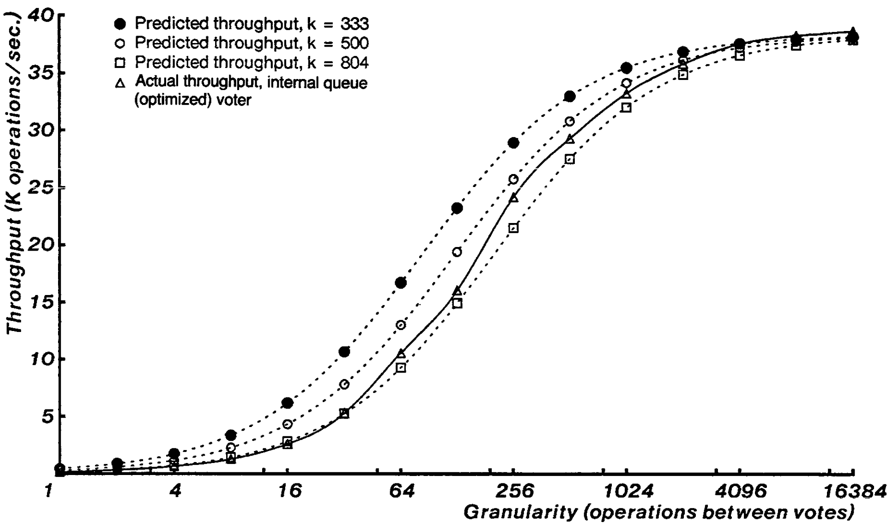
- t_{ave} is the average instruction execution time,
- G is the granularity of a subtask,
- a is the number of instructions executed by a subtask when G is 1, and
- k is the total overhead per iteration, including voting,

then the throughput T is given by

$$T = \frac{1}{t_{ave} \cdot (a + k/G)} \tag{1}$$

The throughput is inversely proportional to the average instruction execution time, the number of instructions per subtask iteration, and the number of overhead instructions per unit granularity (k/G). The values of k, t_{ave}, and a are experimental con-

Figure 12-18 Predicted Voting Overhead



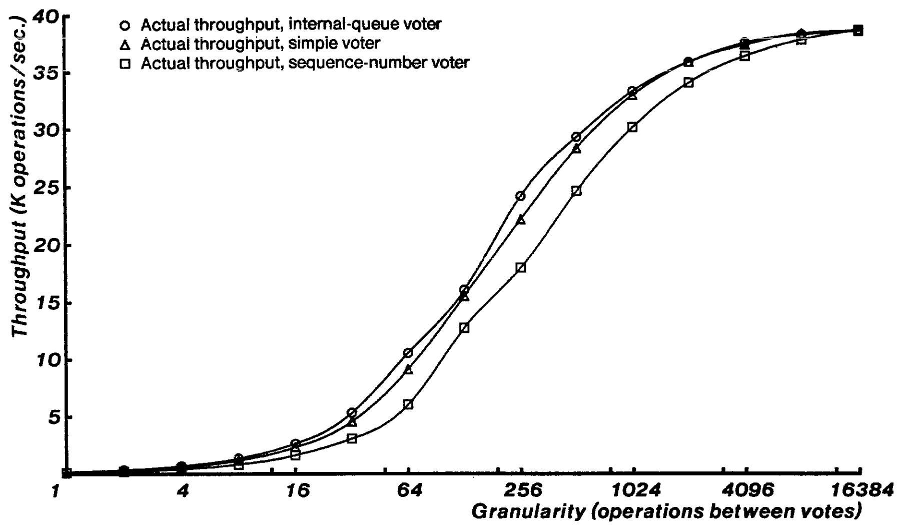
stants, so we can plot the throughput versus the granularity. For typical values of k, t_{ave}, and a ($k=800$, $t_{ave}=6.5\mu s$, $a=4$), the curve is shown in Figure 12-18.

In the experiments on Cm*, the number of iterations I and the granularity G were varied during the experiments. The execution time t_T was recorded for each set of values of I and G. The total work done W was kept constant by choosing a value of I and using it to calculate the value of G according to the formula $W=I \cdot G$. The value of W was always 16,384 operations. All three types of voters described previously were used in this experiment. From the overhead model it can be seen that changing the type of voter should affect only the value of k in equation 1. Throughput versus granularity is plotted for various voter configurations in Figure 12-19. Even significant changes in voting methods seem to have little effect on throughput.

The model is extremely accurate in predicting the overhead in a system. One problem with the model is the difficulty in finding values for the constant k. The value cannot be determined simply by adding instruction times from the subtask and voter because some of the Kmap-implemented instructions have times that depend on system load. In addition, sometimes the voter and the subtask are executing in parallel, so instruction counts would give an upper bound on k but not an accurate value. The amount of parallelism is difficult to quantify without seriously perturbing the experiment. Therefore, the value of k used in Figure 12-18 was estimated using experimental results. The comparison of the predicted and actual curves, however, loses some credibility when the values of k for the predicted curves must be experimentally determined.

The value of k can be given an upper bound in the nonerror case. The upper bound will change as the voter changes, but it can be determined for any given

Figure 12-19 Actual Voting Overhead for Various Voters



experiment. For the optimized voter and subtask experiment, this upper bound has been found by adding the instruction execution times for the subtask overhead to the voter time. The actual value of k will be less than this time because the voter will be executing simultaneously with the subtask. An upper bound on k is approximated by

$$k_{max} = k_{smax} + k_{vmax}$$

where k_{smax} is the maximum subtask contribution to k and k_{vmax} is the maximum voter contribution to k. By analyzing the programs written for the experiments, it is found that

k_{smax} = 68 instructions + 3 ***Sends*** + 1 ***Receive***

and

k_{vmax} = 237 instructions + 3 ***Conditional Receives*** + 1 ***Send***

The execution times for ***Send***s and ***Receive***s on Cm*/MEDUSA are given in Section 7.4. The average execution time for LSI-11 instructions in the voter and the subtask was determined to be 6.5 μs. Using this information, k_{max} is determined as follows:

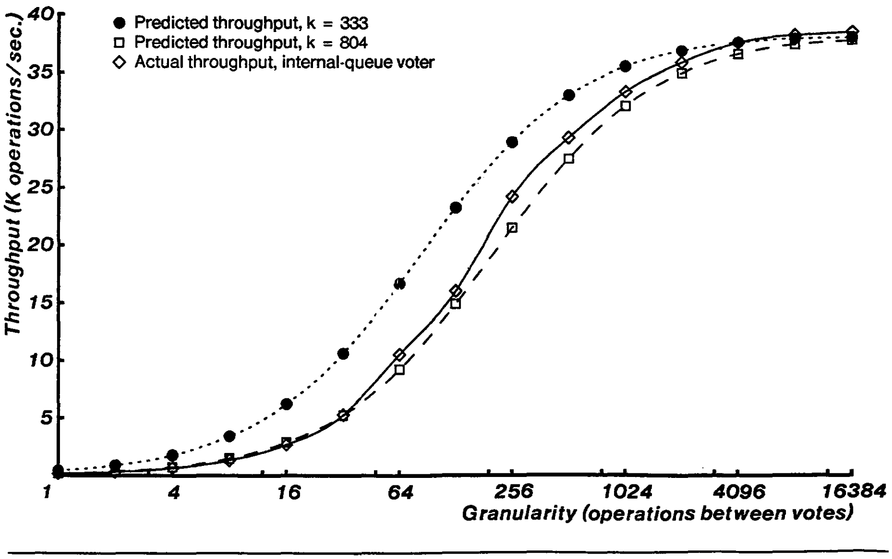
$$k_{smax} \approx 333 \text{ LSI-11 instructions}$$
$$k_{vmax} \approx 471 \text{ LSI-11 instructions}$$
$$k \leq 333 + 471 = 804 \text{ LSI-11 instructions} \tag{2}$$

Similarly, the lower bound can be approximated by the following:

$$k_{smin} = 68 \text{ instructions} + 3\ \textbf{\textit{Sends}} + 1\ \textbf{\textit{Receive}}$$
$$k_{vmin} = 127 \text{ instructions} + 2\ \textbf{\textit{Conditional Receives}} + 1\ \textbf{\textit{Send}}$$
$$k_{min} = \max(k_{vmin}, k_{smin})$$
$$k \geq 333 \text{ LSI-11 instructions} \tag{3}$$

Equation 3 assumes maximum simultaneous execution of the subtask and the voter. The experiments with the optimized voter yielded values of *k* between 350 and 712. These experimental results fall between the minimum and maximum theoretical values calculated above. The bounds should be recalculated if the voter or subtask is changed. Figure 12-20 compares the minimum and maximum predicted curves and an experimental curve (for the optimized voter). One result that the model does not take into account is that the value of *k* changes as the granularity changes; it assumes that the value of *k* is a constant. During the optimized voter experiment, the value of *k* varied by more than 350 instructions. This is due to the change in load on the Kmap processors as the granularity changes. In spite of these deficiencies, the overhead model does give accurate predictions of expected voting overhead.

Figure 12-20 Comparison of Actual and Predicted Voting Overhead



12.2.2. Voter Queue-Length Experiments

In an asynchronous NMR computer system, the processors have their own clocks and make little or no effort to synchronize the clocks with each other. The random variation in clock speed, and the difference in process execution patterns, will cause data to reach the voter at different times. The voters should be able to vote on data as soon as they have received it from a majority of processors. As noted above, a queue must be maintained for data that is received asynchronously. As long as no data dependencies exist, one processor should not be forced to wait for another to finish a calculation.

Even when data dependencies do exist, when a majority of the processors agree on the value of a step, there is no reason to wait for the rest of the processors to finish before continuing with the next step. In fact, waiting can reduce reliability if a processor is faulty because it may never respond to the voter. There should, however, be a limit to the amount a processor should be allowed to fall behind before it is considered faulty. Experiments have been performed to discover the nature of how variations in process execution speed affect the amount a process falls behind the others. These experiments have examined the effects of varying both process execution speed and the number of instructions between votes.

Experiments have been performed to explore two different aspects of the synchronization problem. Experiment 1 has a single process execute more instructions for every step in the experiment. This process is continuously slower, and as it falls behind, voter overhead is observed to increase. Experiment 2 has one process slower for a period, followed by a period of normal speed. This experiment is realistic because in many systems, processes are likely to fall behind due to random or load-induced speed variations.

Experiment 1. The first experiment was designed to measure the ability of the voter to synchronize the subtasks when one subtask is continuously slower than the others. The slower subtask performed 10 to 50 percent more operations in calculating a value. It represents a process that requires more execution time due, for example, to retrying instructions or handling interrupts. The frequency of voting (or granularity of the subtasks) was varied, as was the execution speed of the slower subtask. The queue lengths of the voters were recorded as a measure of how far the slow subtask fell behind the two faster subtasks.

Each voter recorded its queue length each time it added an entry. The queue-length information was sent as a message to a process that stored the data in a file. Although the queue-length recording added some overhead to the voter, each voter paid the same cost.

The queue length was plotted against the iteration number for two different granularities and various subtask degradations, as shown in Figures 12-21 and 12-22. For granularity equal to 1,024 operations, the queue length stays at one if one subtask is up to 10 percent slower. This implies that the voter overhead is great enough to mask the differences in speed. For larger differences in speed, the queue length grows to some value and then levels off. The queue length is bounded due to

Figure 12-21 Granularity Equal to 1,024, One Subtask Always Slower

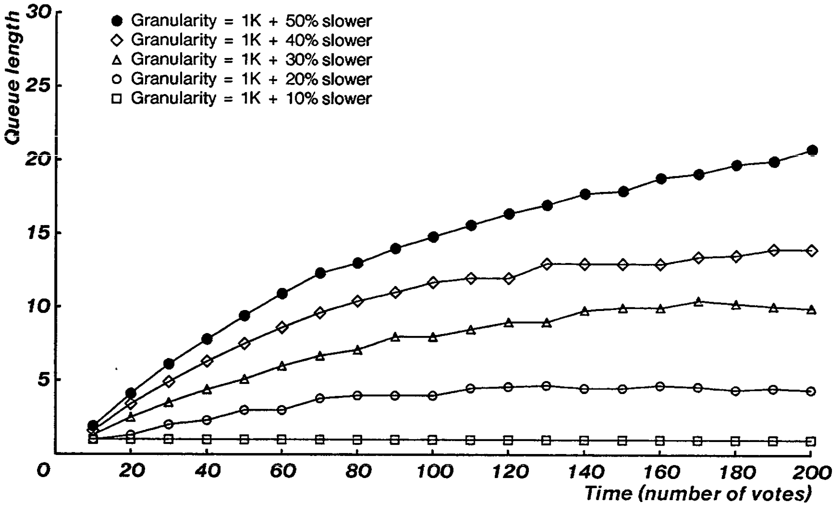
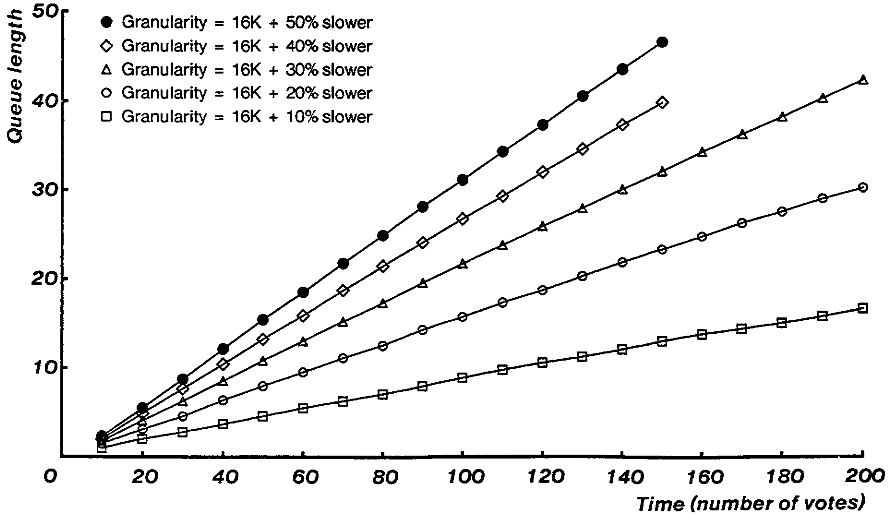


Figure 12-22 Granularity Equal to 16,384, One Subtask Always Slower



an increase in voter execution time as the queue length increases. The voter uses a linear search to find the iteration number in the queue. The slower subtask will not pay this overhead cost because it has n-1 messages waiting for processing, where n is the queue length.

As the granularity increases, the queue length grows more rapidly. With granularity equal to 1,024 (Figure 12-21), the 10 percent to 40 percent additional-operations curves appear to be bounded, but the 50 percent additional-operations curve is not bounded. All curves for granularity of 16,384 (Figure 12-22) appear to have an unbounded queue length. This is due to the fact that the voter overhead takes a smaller percentage of the total execution time when granularity is larger. The slower subtask is incapable of "catching up" while the voter executes.

Experiment 2. A subtask that is performing a calculation may experience a temporary slowdown, followed by a period of normal behavior—for example, a subtask that must perform a recovery routine because of a bus error or must perform a one-time operating-system task. Is the processor running the subtask doomed to stay behind, or will it eventually catch up, even though it takes just as long to calculate new data values as the others? As soon as a subtask falls behind, it no longer pays the voter overhead cost because it has messages queued up waiting for processing. This fact would suggest that a subtask can catch up, and the rate at which it catches up is the incremental voter overhead cost per iteration.

The experiment can be described as follows: one subtask performs 10 to 50 percent more operations for 20 iterations, followed by a period of normal behavior (performing as many operations as other subtasks). The results of the experiment are shown in Figures 12-23 and 12-24. It can be seen that during the periods of normal operation, the queue length declines; given a long enough period of normal behavior, it would reach 1.

Experimental Conclusions. These experiments give a clear picture of a synchronization model for the equal-subtasks paradigm. The model appears to illustrate two phenomena. First, there is a minimum voter overhead that is due to the time required by the voter to receive a message, handle the data, and vote on the data. The subtasks that have a queue length of 1 must pay this overhead cost every iteration of the experiment. Second, the overhead cost increases as the voter queue length increases due to an increase in the data-handling cost. This factor would indicate that for a long enough queue, the voter could mask any difference in process speed. For practical queue lengths, though, the increase in voter overhead masks only some of the subtask speed variation.

The synchronization experiments can give some design principles for TMR asynchronous voting systems. These principles can be applied to optimize the voter queue length, to choose a subtask granularity, and to determine the permissible variation in process speeds. Proper application of the principles will yield a design that has a bounded queue length for all possible variations in process execution rates. The principles can be summarized as follows:

Figure 12-23 Granularity Equal to 1,024, One Subtask Slower Half the Time

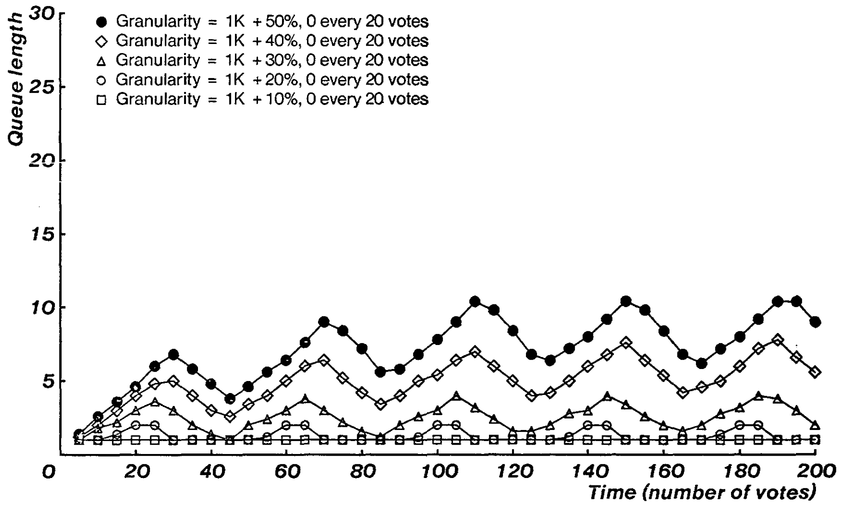
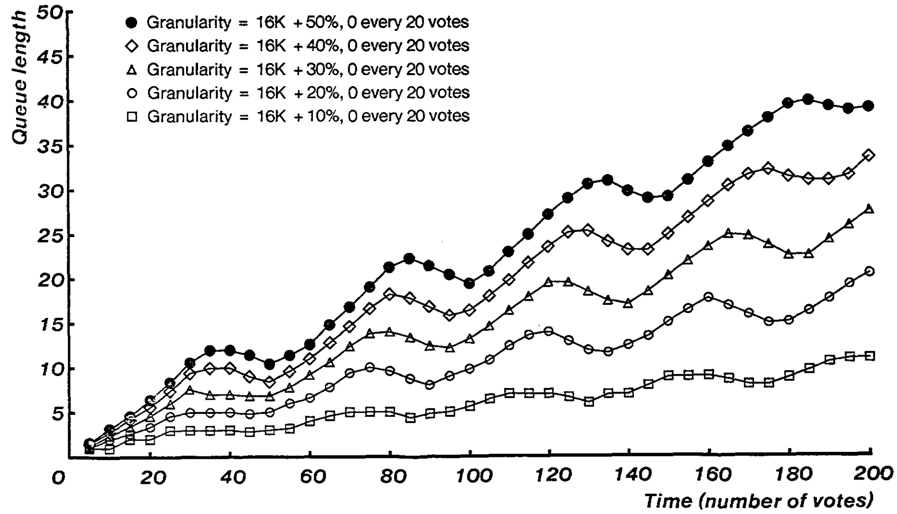


Figure 12-24 Granularity Equal to 16,384, One Subtask Slower Half the Time

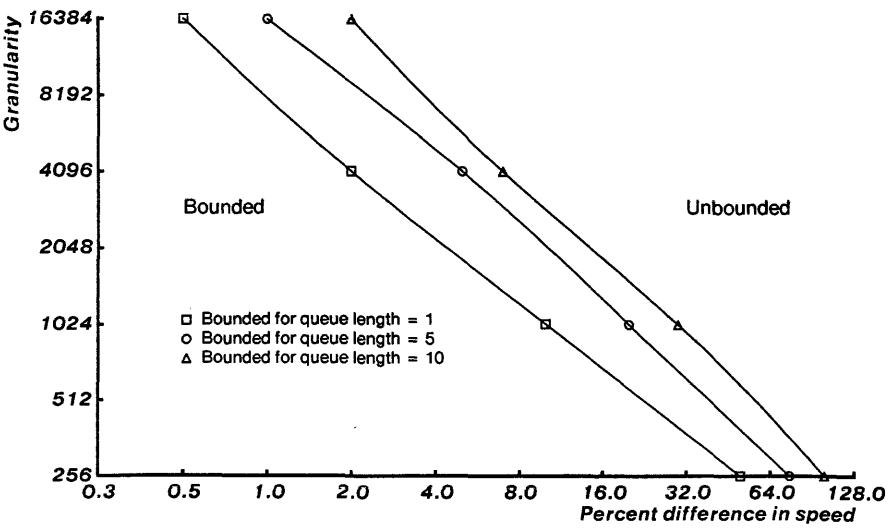


- Smaller granularity subtasks have a higher probability of having a bounded queue length.
- As subtask granularity increases, the random variations in process speed have an increasingly important effect on queue length.
- Greater voter overhead allows a greater variation in process execution rate. This implies a trade-off, as a faster voter process increases system throughput but decreases the amount of variation permitted in process execution rates.

These results can be generalized to synchronous voting as well as asynchronous voting. If the maximum voter length is fixed at 1, then the system is synchronous like SIFT [Forman 79, Goldberg 80, Goldberg 81] and C.vmp [McConnel 81, Siewiorek *et al.* 78a]. C.vmp has a hardware voter with a built-in wait feature. The delay through the voting hardware corresponds to the voter overhead in these experiments. The voter overhead corresponds to the design margin in the fixed schedule (the time between the end of the process execution and the end of the time slot).

Experiment 1, in which one subtask was continually slower, will be used in developing the model of queue behavior. Each experimental curve in the previous section begins to peak as time proceeds. The rate of increase slows with time. It appears to approach some bound that is dependent on the granularity and the difference in execution speed. Some curves have observable bounds. The information from all the Experiment 1 curves could be summarized if this bound information could be collected. If a curve shows a maximum queue length greater than the storage allocated to the queue, then the queue will overflow during the experiment. This is represented by the region labeled "unbounded" in Figure 12-25. Otherwise, the queue falls in the "bounded" region of the figure.

Figure 12-25 Summary of Experiment 1 Data



The system designer can select a maximum queue length, which then determines the granularities and subtask execution-speed differences that will prevent overflow. The curves that determine the regions appear to be linear on the log-versus-log scale. This implies that

$$\log_2 \mathit{Granularity} + \log_2 \mathit{PercentDifference} = \mathit{constant}$$

and, therefore, that

$$\mathit{Granularity} \times \mathit{PercentDifference} = \mathit{constant} = \mathit{voterOverhead}$$

This result indicates that for a given queue length, the granularity of the subtasks is inversely proportional to the percentage difference in processor speed. The constant is a number of operations that is dependent on the voter overhead. A first approximation would equate this number of operations to the voter overhead for one iteration. The voter overhead is constant along a boundary separating the bounded and unbounded regions. A subtask can be constantly slower by a number of operations (the voter overhead) and still fall only some constant number of iterations behind the other subtasks.

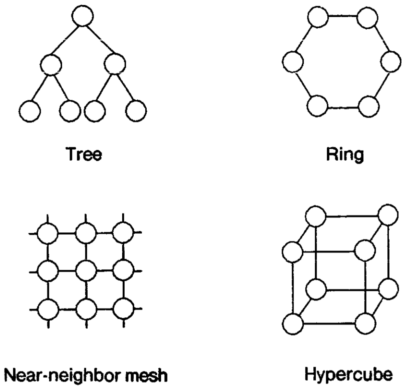
12.3. Interconnection Strategies

All multiprocessors require an interconnection structure that physically implements the shared address space. One obvious structure is a *fully connected multiprocessor*, in which each processor can directly communicate with any other processor without the mediation of a third processor. Numerous other proposals for interconnection structures appear in the literature, covering a wide range of performance, cost, and reliability [Anderson and Jensen 75, Feng 81, Haynes *et al.* 82, Thurber 74]. Unfortunately, comparatively few interconnection structures have been implemented in existing multiprocessors, so the principal sources of comparative information come from modeling, simulations, and educated guesses.

The design and evaluation of interconnection structures is a major area of research in multiprocessor architecture. A few of the possibilities are point-to-point networks such as Cube-Connected Cycles [Preparata and Vuillemin 81]; nearest-neighbor meshes; shared-bus networks, such as the mesh scheme originally proposed for Cm* [Swan 78]; and multistage networks, such as the Augmented Data Manipulator [Adams and Siegel 82] and Omega [Lawrie 75], where data may traverse a path through several switching elements. Interprocessor communication may be through direct memory references or through explicit messages. The multistage networks tend to favor direct memory references or short messages, while the more sparsely connected point-to-point networks favor long messages with store-and-forward operations at the nodes. Some examples of the multiprocessor interconnection networks suitable for experimentation on Cm* are shown in Figure 12-26.

Cm* can be configured to compare alternative interconnection strategies. With

Figure 12-26 Typical Interconnection Networks Suitable for Cm* Testbed Emulation



Tree

Ring

Near-neighbor mesh

Hypercube

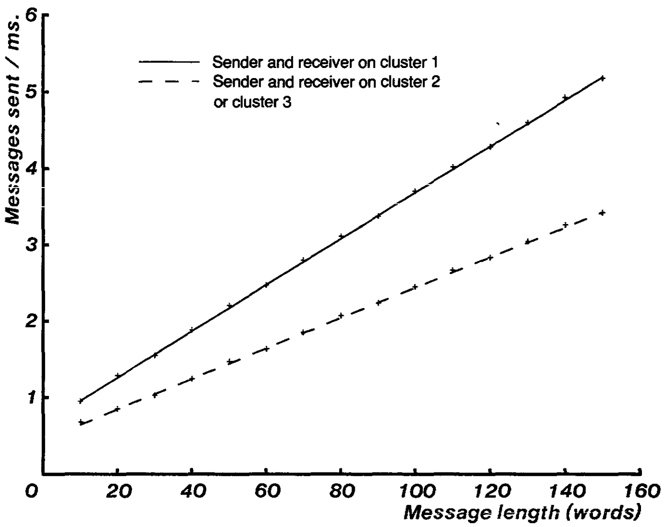
the standard Kmap microcode for either StarOS or Medusa, Cm* is a fully interconnected multiprocessor. By modifying the microcode, however, certain communication paths can be removed, with the result that sometimes it is necessary for information traveling from one processor to another to pass through a third processor along the way. The interconnection network has, in effect, been modified. Even without modifying the microcode, the same experiments can be performed by modifying the benchmark programs to ignore some of the paths. In this case, when information is to be transferred between processors, software *routing tables* are consulted to see which Cm's are in the path.

In the emulations, accesses to the code and data in local memory proceed directly without reference to the routing tables. Communication with remote modules is done via messages. The Medusa message system (see Section 5.2) is well suited to emulation of alternate architectures through software routing. The time cost of the routing software generally is less than the time it takes to send the message, as would be the case in a real system. Intercluster messages take approximately the same time as intracluster messages, provided that pipes are properly located, as described below. Consequently, the impact of Cm*'s hierarchical structure on the experimental results is minimal. Four different multiprocessor configurations were emulated:

1. The *basic experiment* used full interconnection directly implemented with Medusa communication primitives.
2. The *fully connected* configuration used emulated direct connections; software routing tables routed messages for any processor directly to that processor.
3. Another configuration used emulated *nearest-neighbor* communication.
4. The last configuration was an emulated *ring*.

In the initial set of experiments, both the *Sender* and *Receiver* processes were

Figure 12-27 Effect of Placing Pipe in Same Cluster as Sender and Receiver

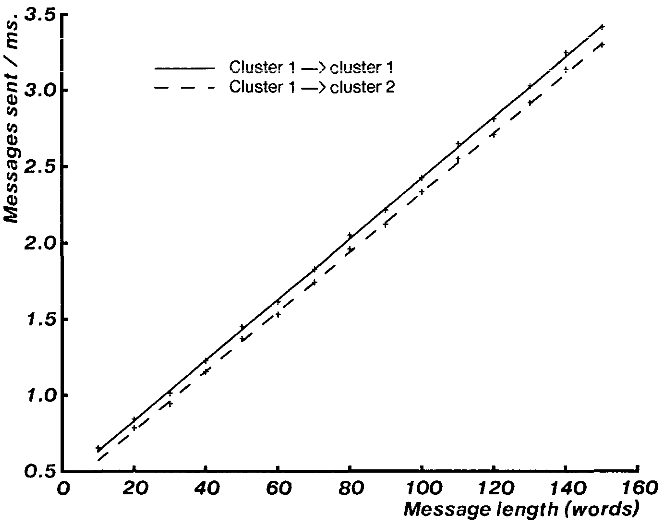


located in the same cluster (although on different processors). Sometimes the pipe was in that cluster; sometimes it was not. The greatest message throughput was obtained when the pipe was on a different cluster from the *Sender* and *Receiver* processes. This surprising observation appears to be due to Kmap contention caused by the heavy use of the communication mechanism. Placing the pipe on a different cluster distributes this load over several Kmaps, resulting in greater throughput, even when several messages are being transmitted simultaneously. No significant penalty is paid for intercluster transmission, since MEDUSA message operations take essentially the same amount of time regardless of where the pipe is located (see Section 7.4.2).

Figure 12-27 shows the effect of placing the pipe on the same cluster. For all three curves in the figure, the pipe was on cluster 1. The sender and receiver were on the same cluster, either cluster 1, 2, or 3. The cluster 1 to cluster 1 transfers were considerably slower than those on clusters 2 and 3.[5] If the cluster 1 transfer measurements are redone with the pipe on another cluster, the transfer rate is identical to those of the other two clusters. There is a small decrease in transfer rates when the *Sender* and *Receiver* processes are both on the same cluster, as shown in Figure 12-28. The decrease is quite minor, however. Thus if pipes are properly placed, a nearly flat communication structure results, providing a base for accurate emulation.

[5] The data in this plot indicates somewhat higher message throughput than the Sindhu-Singh measurements reported in Section 7.4.3. The results are consistent because the measurements presented here do not include the time for the sender to write the data.

Figure 12-28 Effect of Placing Sender and Receiver in Different Clusters



12.3.1. Methods Used to Emulate Multiprocessor Architectures

The emulation package was implemented by adding table-lookup code to each process to perform message routing and delivery. Each process can communicate only with logically adjacent processes as determined by routing tables. Messages destined for nonadjacent processes must be forwarded by intermediate processes. Along the way, they are buffered in pipes known as *virtual buffers*. (In this experiment, "buffer" is simply a synonym for "pipe.") Routing information is contained in the first word of the message. It consists of source, destination, and virtual-buffer indices. Routing tables computed partly at compile time and partly during initialization determine the exact path taken by each message. At fixed intervals, each process checks whether any messages have arrived and delivers or forwards them as appropriate.

12.3.2. Description of Modeled System

All the experiments model the behavior of the same software on different emulated interconnection structures. The software is a system written for a three-cluster minicomputer, with four processors in each cluster. Shared memory is used for communication within a single cluster. Messages are used to move data between clusters. Thus the software in this experiment makes no intercluster memory references.

The software system is driven by inputs from external sensors. Its outputs consist of status displays and actuators. The three clusters perform distinct functions in the

Table 12-1 Communication Rates in Simulated Computer System

Dataflow direction	Bits in a word	Words in a message	Messages per second	Data bits transferred per second
C&D — SPU				
C&D —> SPU	32	256	2	
C&D <— SPU	32	256	14	
				131K
ACS — SPU				
ACS —> SPU	32	138	49	
ACS <— SPU	32	132	49	
				423K
C&D — ACS				
C&D —> ACS	32	256	16	
C&D <— ACS	32	256	16	
				262K

overall system. The *Actuator Control System* (ACS) has primary control over the mechanical devices used in this system. A second cluster, termed the SPU, controls a *Signal-Processing Unit*. The final cluster handles overall control and information display, earning it the title of *Control and Display* (C&D).

This is a real-time system that must respond immediately to any significant event. There are minimum throughput requirements, as well as constraints on the maximum latency of certain operations. Performance is measured by determining the maximum sustainable communications rate for different combinations of synthetic workload (which simulates the system behavior) and message length.

To function effectively with limited buffer space, the emulated architecture must have the capacity to handle the message traffic, which consists of communication between processes in different clusters and between processes and external devices. These communication rates are given in Table 12-1. From these rates, one can estimate real-time response requirements. These limits proved to be difficult to measure with the present level of Cm* instrumentation. The message-event generator (see Section 10.3) does, however, facilitate the generation, at fixed time intervals, of messages that can be used to stress the communications system. This generator is used to simulate the inputs arriving at each cluster from outside the system; the external I/O determines the intercluster communication rates. Failure to meet real-time requirements was indicated by message-buffer overflow somewhere in the system.

12.3.3. Description of Experimental Methodology

The task force for this experiment consists of one or more processes in each of the three clusters, plus several support activities. The experiment utilizes resources from the synthetic workload generator system (Section 10.3) and can be described in the

B-language. The support processes consist of the Pegasus user interface, the message-event generator, and a monitoring routine. The user interface provides communication with the user's terminal, controls operation of the message-event generator, and allows control of user-specified variables in the user's processes. The message-event generator monitors a real-time clock and sends short (one-word) messages to specific processes at specified intervals. The monitor process maintains an error vector where other processes may record the occurrence of malfunctions. It repeatedly scans this vector for evidence of changes and reports them to the user. In this experiment, the vector records instances of message-buffer overflow, as detected during attempted message ***Send***s. The use of shared memory is a much less expensive way of communicating error information than sending short error messages.

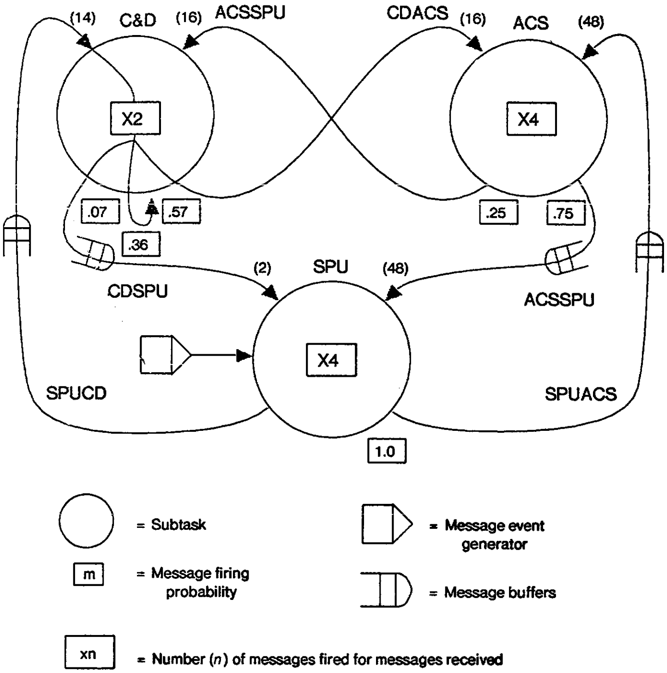
Figure 12-29 shows the modeled system. The simulated workload is driven by the message-event generator, which periodically sends event tokens to the SPU to initiate a unit of system activity by an SPU-to-C&D message. This message in turn generates other messages in such a manner that the average communication rate on the data paths in the simulated system is similar to that experienced by the real system. This is done by generating new messages to send to other nodes under the control of a random-number generator. As shown in the diagram, the C&D process, for example, will generate two new messages for each message received from SPU, with 7 percent going back to SPU, 57 percent going to ACS, and 36 percent going nowhere (simulating messages to external devices). In accordance with the synthetic workload generator methodology, the destination for each message is selected probabilistically, based on these probabilities. The SPU messages generate messages back to ACS, where they are terminated.

When a process receives a message, it performs some simulated work. The work simply consists of executing a null loop as many times as the user has requested through the Pegasus user interface. Each process executes the same average amount of work; that is, the more messages it receives on average, the fewer loop iterations it executes per message. For example, the SPU process receives an average of 50 messages a second and executes 256 iterations per message. The ACS process receives 64 messages per second and executes 200 iterations per message. Thus both processes execute 12,800 loop iterations per second.

An important performance measure for an interconnection structure is its maximum sustainable message rate—that is, the number of messages that can be transported per unit time without causing buffer overflow. This rate was determined for various combinations of interconnection structure and synthetic workload by repeated runs, increasing the intervals between message events until a sustained period of operation was observed during which no buffer overflowed. (Since each processor did the same amount of work, overflow was symptomatic of message-system saturation.)

The maximum sustainable message rate was measured for several combinations of parameters. Processing power was varied by changing the number of processes per cluster. Workload per message and message length were also varied as an aid in characterizing the performance of each interconnection structure. The com-

Figure 12-29 A Graphical Representation of Data Flow in the Simulated System



munication abilities of the fully connected, nearest-neighbor, and ring networks were then compared.

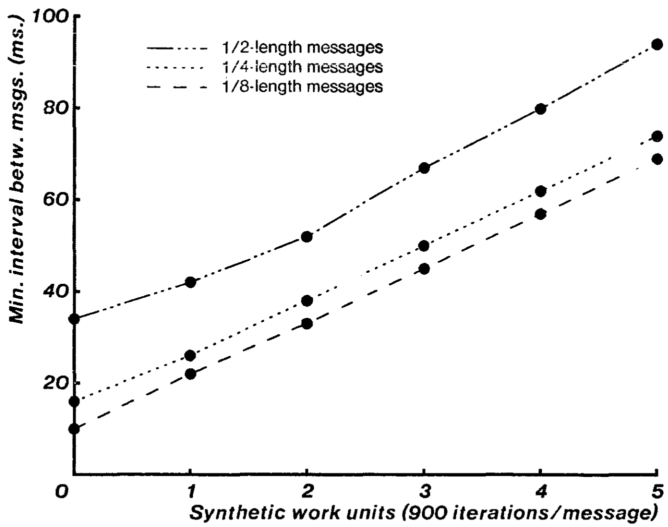
12.3.4. Results

The fully connected multiprocessor, with one process per cluster, was the first configuration to be tested. Three different message lengths were used: one-half, one-fourth, and one-eighth of those specified in Table 12-1. (Unfortunately, the full message length proved to be too much for Cm* / MEDUSA to handle and was deleted from the experiments.) The system was deemed to have saturated when any message buffer overflowed.[6] Message-system saturation periods were found for each message length, as the amount of work per message was varied from zero to five units (of 900 iterations each). The results, averaged over three successive runs, are shown in Figure 12-30.

Note that the three curves are essentially linear. When there are few synthetic work units per message, however, the message system itself contributes more to the

[6] Usually the SPU-to-ACS buffer overflows first, probably because it is the busiest buffer and the ACS process receives the most total messages.

Figure 12-30 Saturation Curves for the Fully Connected Architecture, Basic Experiment



system load, and thus the period between messages is disproportionately longer. As message length diminishes (i.e., the scale factor decreases) the curves become linear even at low synthetic workload, again indicating the reduced effect of messages. Even when the synthetic workload is zero units, message traffic is limited both by the need for the processes to perform housekeeping tasks and by the processor time expended for the message operations themselves; thus the saturation period does not fall to zero.

For the same reason, the small-workload saturation period increases disproportionately as message length increases. When a process sends or receives a message, it is suspended until the message operation completes. Since MEDUSA messages take about 20 microseconds per word to transport, transport time becomes significant at longer message lengths.

The basic experiment performed MEDUSA ***Send*s** and ***Receive*s** directly, without using the software routing tables and periodic polling that were used by the other experiments. To measure the overhead of these factors, the fully connected structure was emulated again, this time with the routing tables and polling. The logical interconnection structure is shown in Figure 12-31. The circles labeled *S0* to *S5* represent the message-switching procedures that route and forward messages. The other circles represent the activities of the simulated task force. Each switching node resides on the same Cm as its adjacent activity. The solid arcs represent connections that are present in the experiments that use one process per processor. The experiments that use two processes each to perform the ACS, SPU, and C&D tasks include the dotted arcs as well.

The results were compared with those from the basic experiment. The curves

Figure 12-31 Fully Connected Emulation Experiment

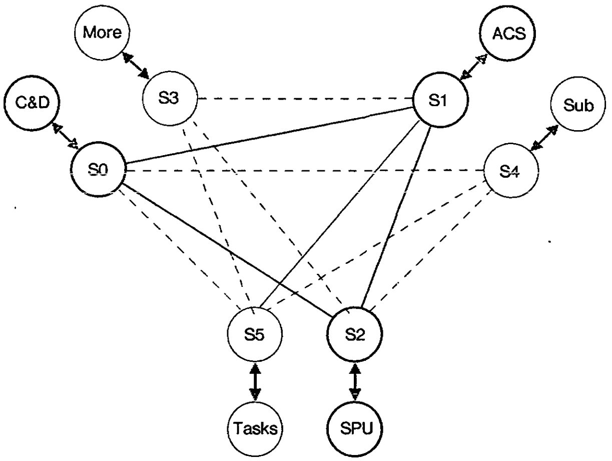


Figure 12-32 Saturation Curves for the Fully Connected Network Emulation

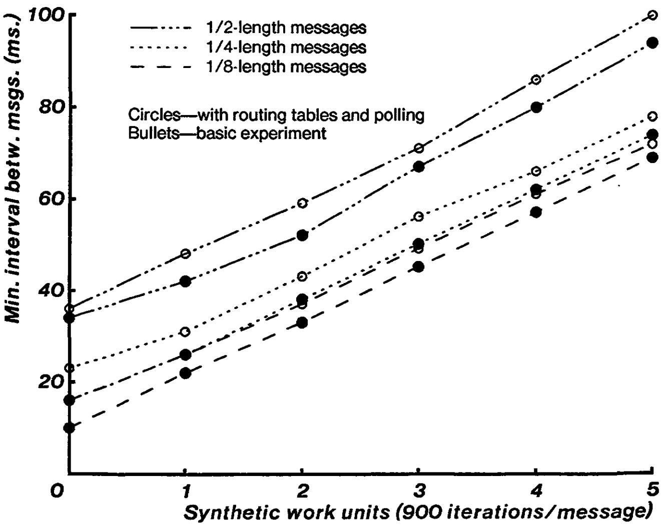
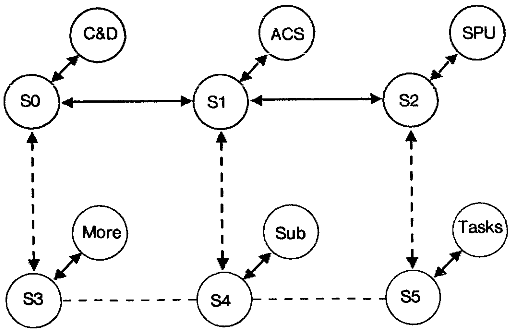


exhibit a small offset from the basic experiment, which is due to the extra overhead of the ***Send*** and ***Receive*** procedures and the periodic polling for incoming messages. As can be seen from the curves presented in Figure 12-32, this overhead is a

Figure 12-33 Nearest-Neighbor Interconnection Scheme



constant 4 to 6 ms. over a wide range of workload and message length. Therefore, its effect can easily be factored out.

These experiments were repeated for the nearest-neighbor interconnection structure. Since there were only three processes used in the initial trials, the interconnection graph actually looks more like a line than a mesh. Figure 12-33 shows the interconnection structure; as above, dashed arcs indicate paths present only in the six-processor version. The processes were arranged in the optimal order, as determined by their communication behavior. A comparison was made with a different arrangement to see how much difference it would make. As shown by Figure 12-34, the time between messages increased dramatically because the system was able to handle fewer messages per unit time.

The curves for the nearest-neighbor network, seen in Figure 12-35, again show a decrease in linearity with increasing message length. The anomalous behavior of the long-message-length curve at low workloads (i.e., the surprisingly low saturation period) appears to be an isolated, though repeatable, case. Even with optimal configurations, the saturation points occur at a significantly lower message rate (i.e., larger intermessage period) than in the basic experiment. Some of this is due to the increased overhead of the message-routing subroutines, as evidenced in the fully connected emulation results, and some is due to the additional burdens of message forwarding, resulting from the poorer connectivity.

After results were obtained for the three-processor system, the experiments were redone with six processors. In addition, a ring network was tested. Since the three-processor ring configuration is identical to the three-processor fully connected network, there was no need to collect separate data. A fully connected network of six processors was included to provide a baseline for the other architectures and to determine the amount of overhead to subtract from the six-processor configurations.

Figure 12-36 presents both the three- and six-processor fully connected curves. At large workloads, the throughput of the six-processor configurations were approximately double those of the three-processor configurations, as would be expected from the dominance of processing over message passing at large workloads.

Figure 12-34 Comparison of Different Process Placement

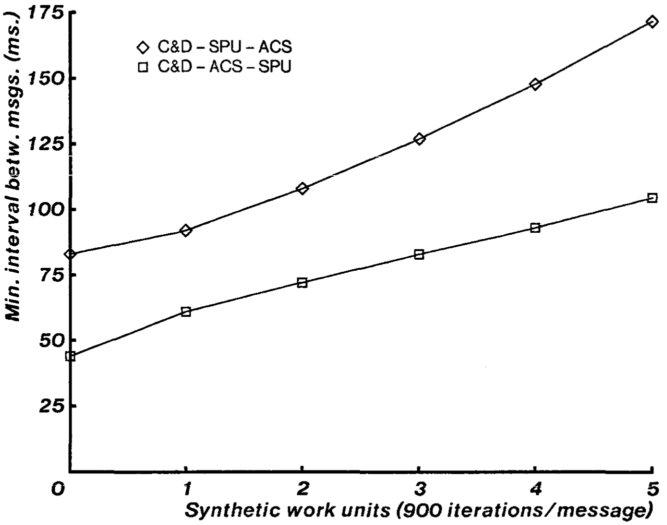


Figure 12-35 Saturation Curves for the Nearest-Neighbor Network

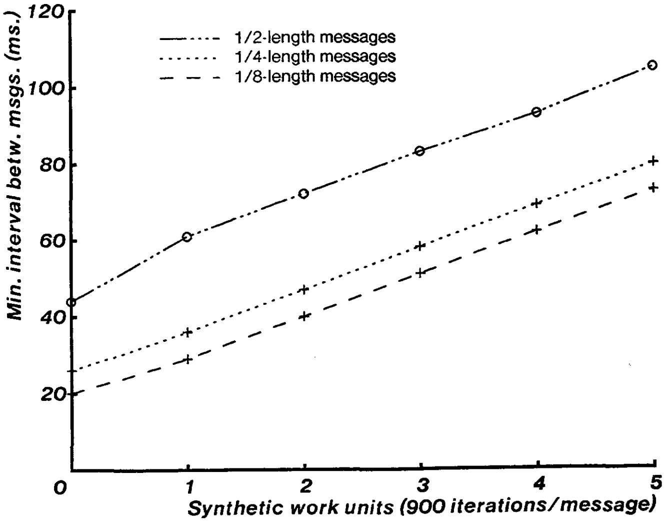
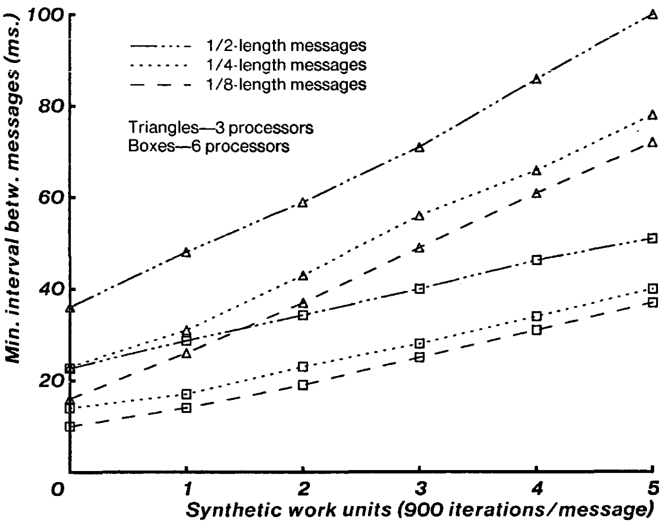


Figure 12-36 Saturation Curves for 3- and 6-Processor Fully Connected Networks



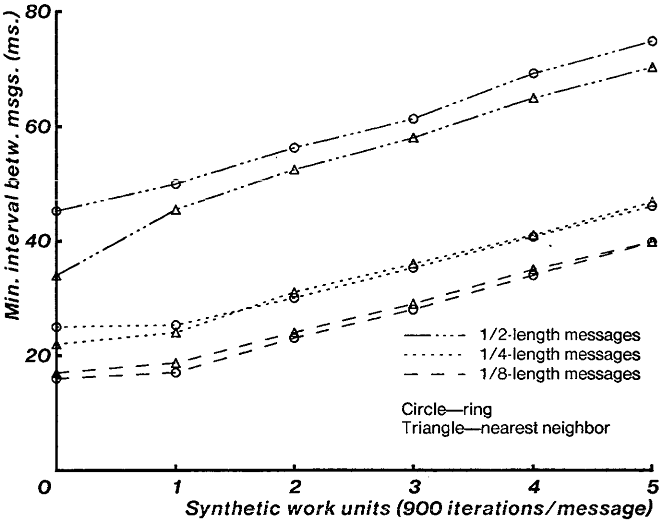
A correction factor similar to the correction factor obtained by comparing the basic and emulated three-processor workloads can be applied. A correction factor one-half as large as that needed for the three-processor configuration predicts the six-processor curves almost exactly for large workloads. This halving of the correction factor is consistent with the notion that the observed overhead is due to the extra computation required by the emulation routines. With six processors instead of three, the emulation overhead per processor is halved for any given message rate. The correction factors derived for three- and six-processor cases will be applied later to the raw data when making detailed comparisons of the interconnection structures.

The raw data for the six-processor nearest-neighbor and ring structures are presented in Figure 12-37. The reduced connectivity of the ring network is reflected in its poorer performance as compared to the nearest neighbor at long message lengths, though they exhibit similar behavior at shorter message lengths.

12.3.5. Usable Processing Power

Of particular interest to multiprocessor designers is the amount of useful work they can obtain for a given application on various architectures. One way of estimating this is to compare the processing power that is "left over" for an application after subtracting the overhead of message transportation. This was done by first subtracting out the emulation overhead, as described in the previous section, and then computing the percentage of time the processors were executing the application. The computation times were calculated from the number of synthetic-workload loop iterations executed per external message (received from the message-event generator) and the measured external message rates.

Figure 12-37 Six-Processor Nearest-Neighbor and Ring Results



The results of the calculations of usable processing power can be used to compare the various network configurations. Figure 12-38 compares the long-message-length (scale of one-half) results for all network configurations, after adjustment for overhead. At first glance, it appears that the usable processing power of each processor, with two processors per cluster, is substantially higher than with only one processor. This is because each processor handles only half the messages sent or received by the cluster. Thus if the amount of processing done by each cluster is directly determined by the incoming message rate, as it is in the system modeled here, the message rate per processor must be considered. Figure 12-39 compares the three-processor configuration of the fully connected network with the six-processor configuration, after accounting for the lessened message rate per processor. Note that the six-processor architecture is actually using its processors less efficiently than the three processor configuration. Note also that the point where each curve intersects the *X*-axis is the message rate that will saturate that architecture.

Even though the experiments involved only a small number of processors, some interesting results can be seen. Again, examining Figure 12-38, it is seen that for this application a six-processor ring utilizes its processors no more effectively than a three-processor nearest neighbor. Of course, more processing power is available because each cluster contains two processors. Notice, though, that above 20 messages per second, a three-processor fully connected system has more than twice as much available power per processor than the six-processor ring. Thus it would definitely be the preferred system at high message rates. Below 20 messages a second, the six-processor ring has more total processing power available and might

Figure 12-38 Usable Processing Power for Various Architectures

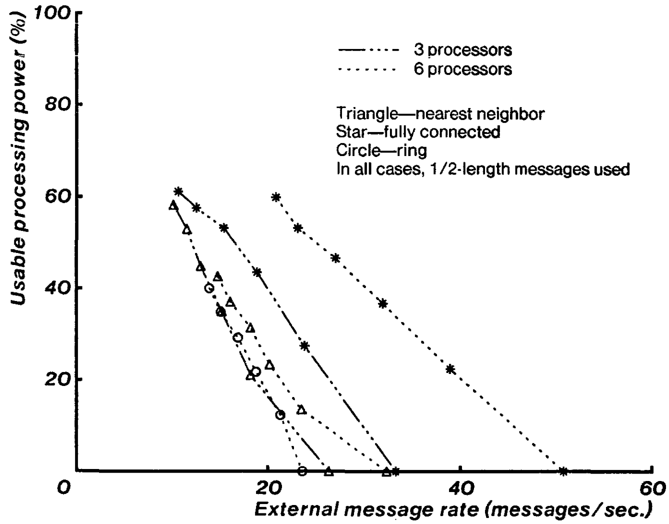


Figure 12-39 Comparison of Usable Processing Power, Adjusted for Per-Processor Message Rates

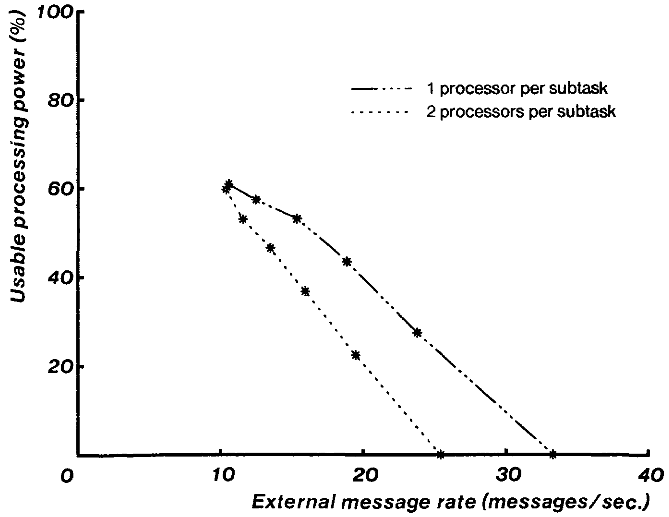
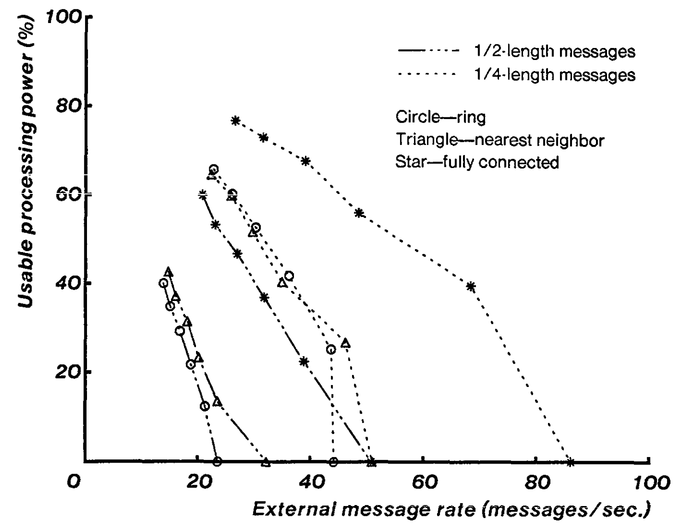


Figure 12-40 Comparison of Six-Processor Interconnection Schemes



be preferred if large amounts of processing were required. It would be cheaper than either the six-processor nearest-neighbor or fully connected systems, provided it supplied sufficient processing power.

Of particular interest is the relative performance of the three multiprocessor architectures when six processors are employed, as the connectivity of the interconnection structures has a great effect on the results. Figure 12-40 compares the curves produced by all three structures when one-half and one-fourth length messages are used. The effects of emulation overhead have been factored out in these graphs so that direct performance comparison is possible. The ring and nearest-neighbor networks provide similar amounts of processing power to their application tasks, though at long message lengths, the nearest neighbor does perform slightly better. In all cases, the fully connected architecture performs markedly better, doing almost as well with long messages as the other two architectures do with short messages. At an external message rate of 20 messages per second, the fully connected network provides almost 2.5 times the processing potential of the nearest-neighbor network. Although the fully connected network requires 15 interconnection links to the nearest neighbor's 7, it often is the better choice.

These experiments demonstrate how it is possible to use Cm*, or other testbeds, to emulate alternative multiprocessor architectures. The methods employed can easily be extended to emulate additional multiprocessors covering a large class of interconnection mechanisms as well. Although some of the overhead incurred in emulating the routing algorithms is due to the emulation software, this can be measured by comparison with nonemulated systems, as was done for the basic experiment. Once the overhead is subtracted, detailed studies of comparative network performance are possible. It is expected that the methodologies presented in this chapter, in conjunction with an experimental environment such as that described in Chapter 10, can furnish accurate assessments of the results of experiments on other multiprocessor systems.

12.4. Summary

Multiprocessor experiments rely on a timing facility to measure time-related information. The Cm* timing facility uses a system clock and provides a number of system calls to read the clock. Preliminary experiments showed wide variance of the clock readings, due to the multiword structure of the Cm* clock and the computational load. Several experiments were performed with the goal of alleviating this problem. The latency, resolution, and variance of the timing measurements were calibrated, and improved clock-reading algorithms were developed. Two methods of compensating for clock variance were evaluated. Both methods are based on the idea of sensing the average load and using it to correct the time measurement.

Calibration of the timing facility is a prerequisite to performance measurement on a multiprocessor. Another aspect of calibrating the experimental infrastructure is to study the virtual software / hardware architecture running on Cm*. Two such sets of experiments have been described. The first one investigates the implementation of triple-modular redundancy and voting for increased availability. A number of implementation parameters were varied, such as synchronization strategies, communication and voting overhead, granularity of voting, and voter queue lengths. Based on the experimental results, an analytical model was created to generalize and predict the voting behavior.

The second set of experiments relate to the emulation of interconnection architectures on the top of Cm*. Cm* was configured to compare alternative interconnection strategies. Certain communication paths were created and removed by modifying the microcode, simulating the effect of changes in the interconnection network. Four different multiprocessor configurations were emulated: the basic connection, which used standard MEDUSA communication primitives (for overhead calibration); the fully connected configuration with software routing tables; the nearest-neighbor configuration; and a ring configuration. For performance comparisons, a real-time synthetic application was implemented on these architectures. The results illustrate the effectiveness of the various interconnections for the application under study. For example, a six-processor ring utilizes its processor no more efficiently than a three-processor nearest neighbor.

These experiments show how one can use Cm*, or other testbeds, to emulate alternative multiprocessor architectures. The methods can easily be extended to emulate additional multiprocessors, covering a large class of interconnection mechanisms as well. Although some of the overhead incurred in emulating the routing algorithms is due to the emulation software, this can be measured by comparison with nonemulated systems, as was done for the basic experiment. Once the overhead is subtracted, detailed studies of comparative network performance are possible. It is expected that the methodologies presented in this chapter, in conjunction with an experimental environment, such as that described in Chapter 10, can furnish accurate assessments of the results of experiments on other multiprocessor systems.

At this point, we have described the Cm* hardware architecture, the operating systems, the software and experiment development environments, the parallel algorithm taxonomy, the timing facility, the interconnection emulation, and the experimental results. The interested reader can now peruse the details of the various Cm* experiments, as described in Appendix A.

Acknowledgments. Section 12.1 was adapted from [Kong *et al.* 83] by Thomas H. Kong, Alfred Z. Spector, and Daniel P. Siewiorek; Section 12.2 was condensed from [York *et al.* 83] by Gary York, Daniel P. Siewiorek, and Zary Z. Segall; and Section 12.3 was adapted from [Wilson *et al.* 83] by Andrew W. Wilson, Daniel P. Siewiorek, and Zary Z. Segall.

Appendix A
Experiments Performed on Cm*

A.1. Unicluster Partial Differential Equation Solver

Algorithm name: Partial differential equations (PDE).
Cm configuration*: Eight Cm's, one cluster.
Operating system: Smap microcode.
Other software in environment: None.
Experimenter: Levy Raskin, 1978.
Reference: [Raskin 78].

The objective is to solve Laplace's partial differential equation (PDE) with given boundary conditions (Dirichlet's problem) by the method of *finite differences*. The equation

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0$$

is solved for points of an *m*-by-*n* rectangular grid, where only the values at the outer edges of the grid are given. The solution is found iteratively. On each iteration the new value of every element is set to the arithmetic mean of the values of its four adjacent neighbors. Each process runs on its own dedicated processor; it performs the iteration for a fixed, contiguous subset of the grid array, which will be called a *task*. Thus the processes are distinguishable.

Raskin implemented several different variations [Baudet 78] of this algorithm. All methods use one process per processor, so these two terms can be used interchangeably. The processes iterate on equal-sized partitions of the grid. One process (the *master*) starts the other *slave* processes and then works on a portion of the grid array, just like a slave process.

1. *The Jacobi Method.* At the beginning of each iteration, a processor retrieves its partition from a global array. New values are computed for each element of the partition, then stored back into the global array. This storing is performed inside a critical section. The processor then checks its error vector (computed from the difference of the new and old values in its partition). If the error vector is smaller than a prespecified limit, the processor reports that it has finished. Otherwise it blocks until the other processors have completed the current iteration. Iterations are performed until all processors have finished.
2. *The Asynchronous Jacobi Method.* This method is the same as method 1, except that a processor does not wait for the other processors to finish before starting on the next iteration.

3. *The Asynchronous Gauss-Seidel Method.* This method is similar to method 2, except that the processor uses newly computed values as soon as they are available instead of the values known at the beginning of the iteration.
4. *The Purely Asynchronous Method.* To compute new array values, this method uses the most recent values of all components by reading them directly from the global array and writing the updated values back to the global array (without any critical sections or synchronization). It uses a critical section only for a processor to report that it has finished.

Results. Raskin studied these algorithms using a 21-by-24 array (504 elements) on a one-cluster Cm* system with a maximum of eight processors. Measurements were directed toward two goals: to determine how much degradation would be caused by mapped memory references (see Section 4.2) and to compare the performance of the four methods.

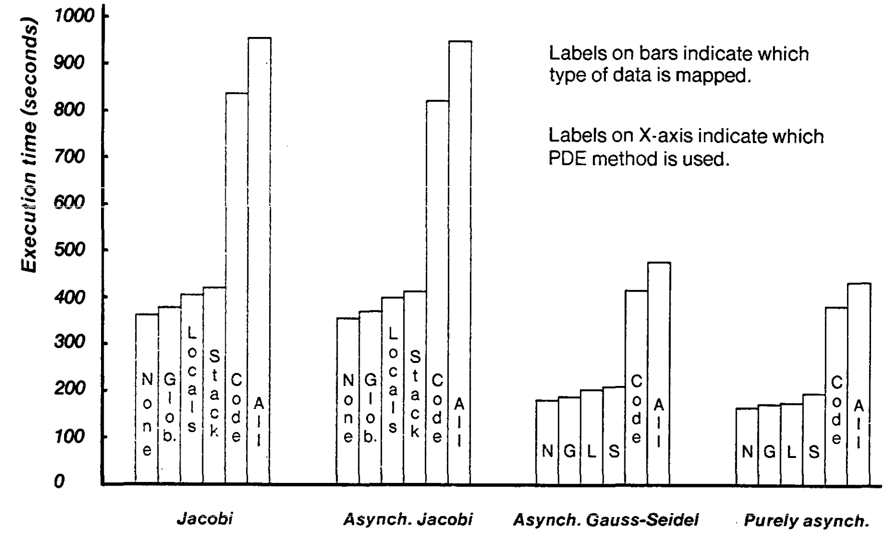
MAPPED MEMORY REFERENCE TIMES. Each memory reference made by a process falls into one of four classes: The reference is either to code, to the process stack, to a private variable, or to a global variable. Using a uniprocessor version of each algorithm, Raskin measured execution time when each class of memory reference was mapped by the Kmap rather than proceeding directly to local memory. (Since only one cluster was in use, these were all intracluster references.) The results are presented in Table A-1 and graphed in Figure A-1. For each method in the figure, the first bar represents execution time without any mapped references, and the last bar gives execution time when all memory references are mapped. The middle bars show execution time when exactly one class of memory references is mapped. Regardless of which algorithm was being measured, placing the global variables in mapped memory degraded performance about 4 percent, mapped globals cost 16 to 18 percent, and mapped code cost about 130 percent. Methods 3 and 4 were always about twice as fast as methods 1 and 2.

Table A-1

Uniprocessor PDE Memory Reference Overheads

	1. Jacobi		2. Asynch. Jacobi		3. Asynch. Gauss-Seidel		4. Purely asynchronous	
Memory reference pattern	Exec. time	% of local exec. time	Exec. time	% of local exec. time	Exec. time	% of local exec. time	Exec. time	% of local exec. time
All local	362	100	355.5	100	181	100	165.5	100
Only globals mapped	378	104.5	370	104	188	104	173	104.5
Only private vars. mapped	405	112	399	112	203.5	112	176.5	107
Only stack mapped	420	116	413	116	210	116	196	118.5
Only code mapped	835.5	231	820	231	417	230	382	231
All mapped	954	263.5	948	267	478	264	433	261.5

Figure A-1 Comparison of Uniprocessor PDE Implementations



SHARED-MEMORY CONTENTION AND REFERENCE DEGRADATION. Raskin also measured multiprocessor versions of each algorithm. Some memory references were made directly to local memory; others were mapped to nonlocal memory. Some of those nonlocal memory references were to objects that were shared by all the processes. For example, all processes might access the same copy of the code, leading to Kmap and memory contention. The graphs for method 1 (Figure A-2) and method 4 (Figure A-3) are typical, but note that methods 3 and 4 are again approximately twice as fast as methods 1 and 2 in all cases.

A closer look at these graphs shows the effects of two kinds of overhead: nonlocal (mapped) memory references and memory contention. Each graph shows three curves.

- The first curve plots results for runs that incur the least possible amount of either kind of overhead. No data is shared except globals (which must necessarily be shared), so memory contention is kept to a minimum. Similarly, no memory references are mapped except references to global data, which must be mapped if the data is shared.
- The second curve shows what happens when all memory references are mapped. These runs incur maximum mapping overhead but minimum contention overhead. The curve levels off or turns upward at about six processors, which indicates that at this point, mapping overhead begins to dominate, probably due to saturation of the Kmap.
- The third curve illustrates the effect of mapping all memory references and sharing all data, thus maximizing both mapping and contention overhead. No

Figure A-2 Influence of Mapped and Shared References on Method 1 (Jacobi) PDE Execution Time

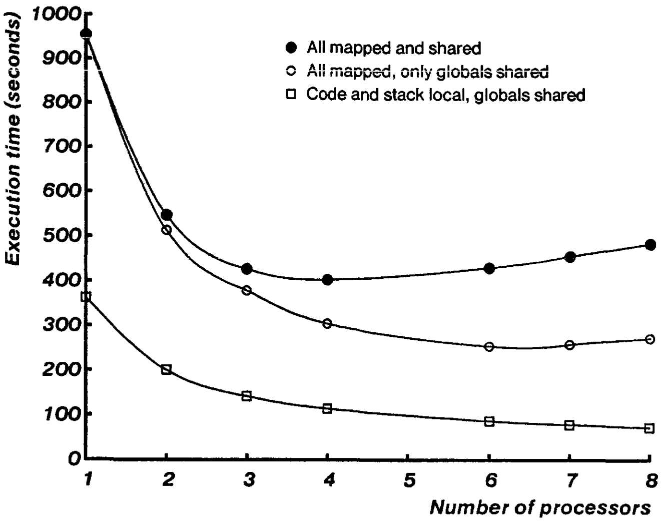


Figure A-3 Influence of Mapped and Shared References on Method 4 (Purely Asynchronous) PDE Execution Time

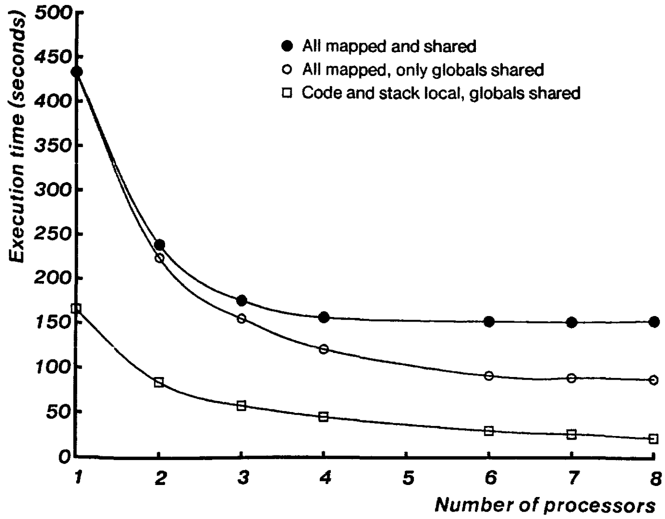
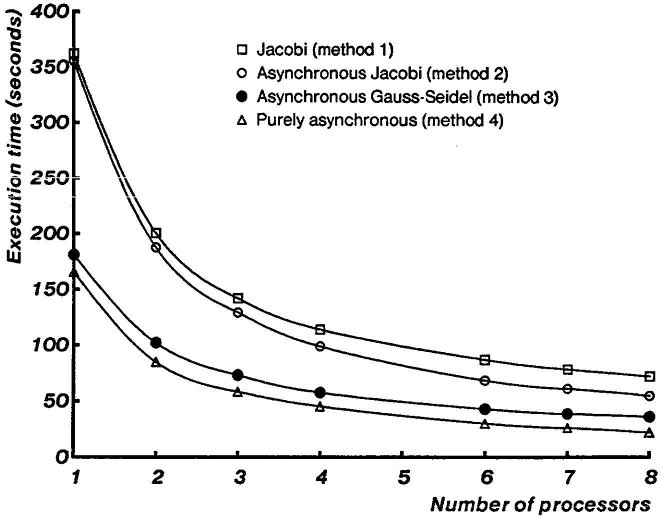


Figure A-4 Comparison of Multiprocessor PDE Execution Times



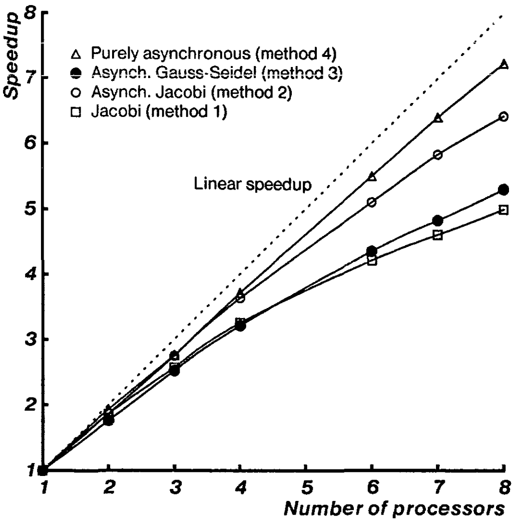
further improvement is possible beyond four processors because of saturation of the shared memory. Consequently, these experiments demonstrate that, as processors are added, shared memory saturates before the Kmap.

SPEEDUP OF THE FOUR METHODS. It is instructive to compare the four methods against each other, in terms of both execution time and speedup. We can re-plot the curves for each method for the case in which only globals are shared. The methods whose synchronization requirements are less demanding find the solution more quickly (Figure A-4). Despite requiring more synchronization, however, method 2 exhibits more speedup than method 3 (Figure A-5) because the uniprocessor asynchronous Gauss-Seidel method has a large speed advantage over the uniprocessor asynchronous Jacobi method—an advantage that cannot quite be maintained as processors are added.

..2. Unicluster Quicksort

Algorithm name: Quicksort.
Cm configuration*: Eight Cm's, one cluster.
Operating system: Smap microcode.
Other software in environment: None.
Experimenter: Levy Raskin, 1978.
Reference: [Raskin 78].

Figure A-5 Comparison of Multiprocessor PDE Speedup



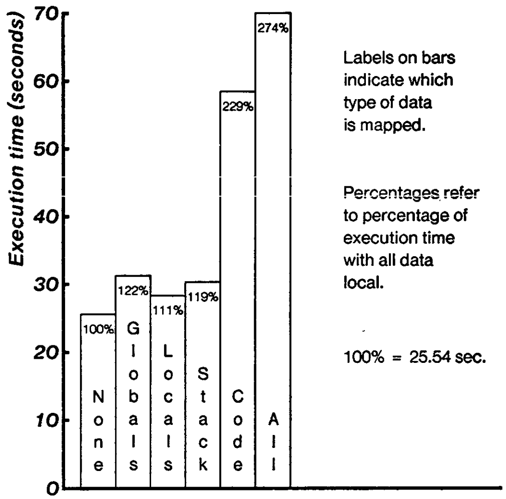
In the multiprocessor quicksort [Sedgewick 78], a number of indistinguishable processes, one per processor, take part in sorting a global array of integers. The processors share a *stack*, which contains descriptors for continuous subsets of the array that have yet to be sorted.

On each pass, a processor tries to pop a descriptor for a new subset from the shared stack. If successful, the processor partitions the subset into two smaller ones, consisting, respectively, of all elements less than and greater than an estimated median value. (The estimate of the median is rather crude; it is the median of the first, middle, and last elements in the subset. This is just slightly more sophisticated than the simplest implementation of quicksort, which on each iteration partitions a set into two subsets, the first consisting of the elements less than the middle element.) After this partitioning, a descriptor for the shorter of the new subsets is pushed onto the stack, and the longer subset is further partitioned in the same way. When a subset cannot be partitioned further, the processor pops the next descriptor from the shared stack.

Unlike the PDE algorithm, the multiprocessor quicksort algorithm [Raskin 78] cannot approach linear speedup (see Section 11.3.1). Its theoretical speedup S (assuming zero implementation penalty) can be computed from

$$\frac{1}{S} = \frac{1}{N} + \frac{2 - \frac{\log_2 N}{N} - \frac{2}{N}}{\log_2 M}, \tag{2}$$

Figure A-6 The Cost of Mapped Quicksort References



Labels on bars indicate which type of data is mapped.

Percentages refer to percentage of execution time with all data local.

100% = 25.54 sec.

where *N* is the number of processes and *M* is the number of elements sorted. Note that the speedup is enhanced if *M* grows large relative to *N*. Raskin's experiments used an *M* of 18,000. With two processors, the theoretical speedup is 1.87. For an *M* of 1,800, the theoretical two-processor speedup would be only 1.83.

Like the multiprocessor PDE, the multiprocessor quicksort employs a master/ slave implementation. After starting the slaves, the master participates in the sort, just like the slaves. The master and each slave execute on distinct processors. The shared stack, the vector of numbers to be sorted, and the other global variables are kept in the local memory of the Cm that runs the master.

Results. Raskin sorted an array of 18,000 elements on a one-cluster system with a maximum of eight processors. As in the PDE experiment (Section A.1), the goals were to measure degradation due to mapped memory references and to measure the speedup as the number of processors was increased.

Data referenced by a process can be classified into four categories: executable code, private variables, global variables, and the process stack.[1] Unlike the PDE, the quicksort references global variables more frequently than private variables; otherwise its reference patterns are similar. Since only one cluster was in use, all references are intracluster. These values are graphed in Figure A-6.

[1] This refers to the call/return and expression-evaluation stack, not the shared stack of subset descriptors, which are global variables.

Figure A-7 Multiprocessor Quicksort Speedup

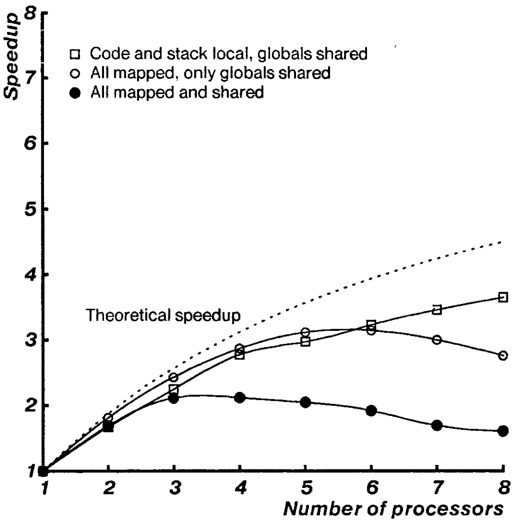


Figure A-7 shows how speed varies according to the number of processors used. Comparing the different curves illustrates how mapped references and memory contention affect the results. If all data is shared, memory saturates at about three processors. If all data is mapped, the Kmap saturates at five to six processors. Adding additional processors slows down the experiment. Note the similarities between this experiment and the uniprocessor PDE (see Section A.1).

A.3. Integer Programming

Algorithm name: Set-partitioning integer programming.
Cm configuration*: Eight Cm's, one cluster.
Operating system: Smap microcode.
Other software in environment: None.
Experimenter: Levy Raskin, 1978.
Reference: [Raskin 78].

Set-partitioning integer programming implemented for Cm* [Raskin 78] uses an enumeration algorithm that performs an n-ary tree search in a large, relatively sparse binary matrix for a minimum-cost solution. The matrix is two-dimensional; its size is usually on the order of hundreds by thousands. The problem is to solve

$$\min(\mathbf{c} \cdot \mathbf{x} \mid A\,\mathbf{x} = \mathbf{e}, x_j = 0 \text{ or } 1 \text{ for } 0 \leq j \leq N)$$

where A is an M-by-N binary matrix,
c is a vector of length N,
and **e** is the identity vector of length M.

As an example, consider the airline-crew scheduling problem. The rows of the A matrix correspond to a set of flight legs to be covered during a specified period, and the columns of A correspond to a possible sequence of tours of flight legs made by one crew; **c** is a vector containing the cost of each tour. A feasible solution consists of a set of tours that satisfy all the flight legs (one and only one crew makes a flight leg). The algorithm seeks the solution with the lowest cost.

The multiprocessor version of this algorithm has a negligible algorithm penalty and hence a theoretical possibility of linear speedup. Raskin used a master/slave implementation, with the master initializing a shared stack of possible search-path solutions. Each process ran on a dedicated processor. The slave processes popped the stack each time they needed more work to do. A global variable maintained the cost of the best solution encountered so far. Each process compared its current cost value with this variable, backtracking in the search when the global cost was lower.

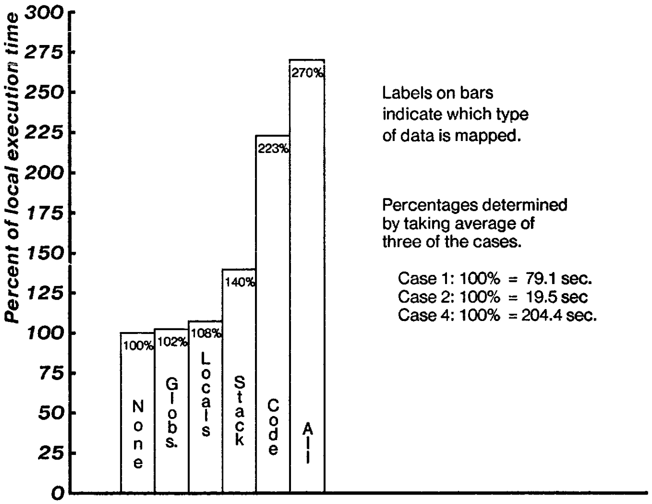
Results. The algorithm was run on five different sets of data (the five "cases"), again with a maximum of eight processors in one cluster. Two versions of the algorithm were investigated. In the first version, the processors communicated by shared memory, making direct memory references to, for example, the shared stack. The second version simulated a network, with the processors communicating via messages, which were copied from the local memory of the sender to the local memory of the receiver ("pass-by-value").

EXPERIMENTS USING SHARED MEMORY. As in the two previous experiments (Sections A.1 and A.2), Raskin studied the penalty that mapped references imposed on the performance of the uniprocessor version. Three of the cases (cases 1, 2, and 4) were measured, and results are presented in Figure A-8. The behavior of the three cases was so similar (the penalty of mapping a particular class of references never varied by more than 3 percent from case to case) that only one set of bars is shown, representing the *average* overhead in the three cases. An interesting anomaly in the speedup is the fact that case 1 achieved greater than linear speedup (see Section 11.2). The speedup curves are plotted in Figure A-9.

MULTIPROCESSOR/NETWORK COMPARISON. Cm* can easily be configured to emulate a computer network (see Section 12.3) if nonlocal memory references are disallowed (for example, by removing the associated microcode from the Kmap) and all interprocessor communication is performed by message passing. Raskin carried out further experiments with the integer-programming algorithm, varying several parameters:

- Configuration —multiprocessor vs. network.
- Acknowledge mechanism —simulated hardware vs. software.

Figure A-8 The Cost of Mapped Integer-Programming References

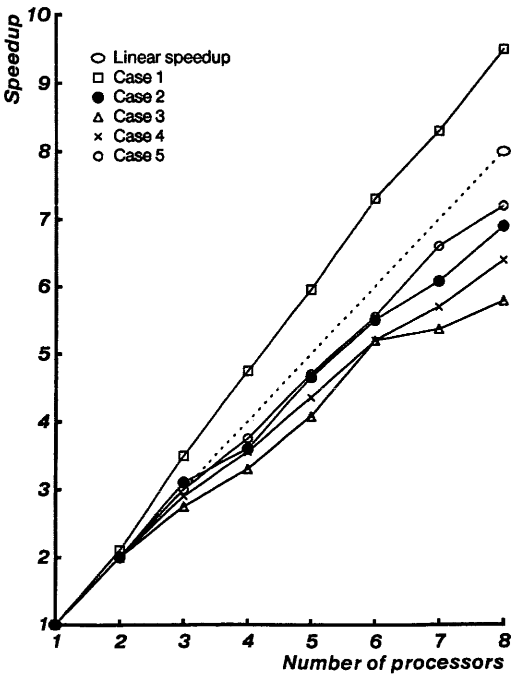


Labels on bars indicate which type of data is mapped.

Percentages determined by taking average of three of the cases.

Case 1: 100% = 79.1 sec.
Case 2: 100% = 19.5 sec
Case 4: 100% = 204.4 sec.

Figure A-9 Speedup of the Multiprocessor Integer-Programming Algorithm



- Network bandwidth—from a minimum of 5K bits/second to a maximum of approximately 1.3M bits/second.

When shared memory was used in the multiprocessor configuration, the Smap microcode (Appendix D) was used; in the absence of contention, intracluster references took 8.3 µs. and intercluster references 26.2 µs. In the network configuration, the shortest possible packet, consisting of three words, could be transmitted in 85 µs., and each additional word took about 12 µs.

To transfer messages reliably, it is necessary to inform the sending processor whether the message has been successfully transmitted so that it may retransmit if necessary. To simulate hardware acknowledgment, the Kmap was programmed to set a flag bit in the first word of the sending processor's copy of the message after it had been successfully transmitted. To perform software acknowledgment, the receiving processor sent an *acknowledge* message back to the sending processor after successful receipt of a message. Including buffer allocation and deallocation and message generation or decoding, it took about 1.1 ms. to send a packet, 0.7 ms. to receive a message, and 0.6 ms. to acknowledge a packet.

The maximum bandwidth of the emulated network was about 1.3M bits/second. Lower bandwidths were simulated by causing the Kmap to delay a certain amount of time between successive word transfers.

The integer-programming application requires relatively little interprocess communication, and thus it performs quite well on the emulated network. In several cases, its performance with maximum network bandwidth is nearly indistinguishable from that of the multiprocessor configuration (Figures A-10 and A-11). Also, because of the light message traffic, the results did not perceptibly depend on which acknowledge mechanism was used. As the communication bandwidth was decreased below 50K bits/second, however, performance did noticeably deteriorate. Figure A-12 presents the results for case 1.

A.4. The Speech-Recognition System Harpy

Algorithm name: The speech-recognition system Harpy.
Cm configuration*: Eight Cm's, one cluster.
Operating system: Smap microcode
Other software in environment: None
Experimenter: Peter Feiler, 1977.
Reference: [Fuller *et al.* 77]

Harpy [Lowerre 76] is a speech-recognition system developed at Carnegie-Mellon University (CMU). Its knowledge of speech is represented in the form of a weighted directed graph. Each node in the graph represents a phoneme, and each weighted arc represents a legal transition from one phoneme to another, with the weight representing the probability of the transition. Harpy has recognized speech in

Figure A-10

Comparison of Integer Programming on Network and Multiprocessor Configurations, Cases 1 and 2

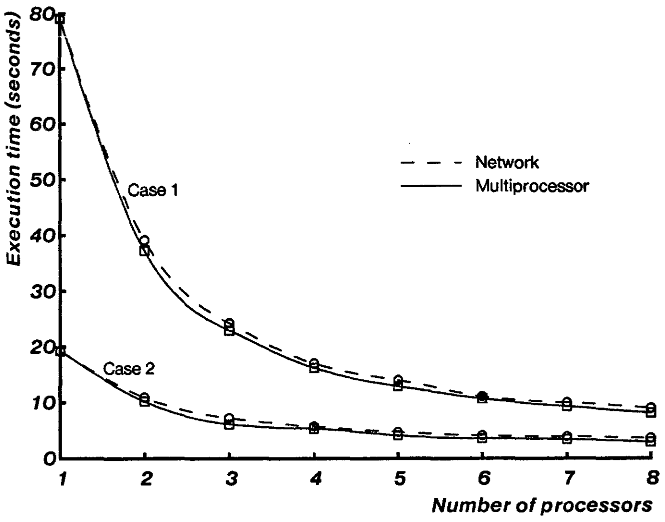


Figure A-11

Comparison of Integer Programming on Network and Multiprocessor Configurations, Cases 3 through 5

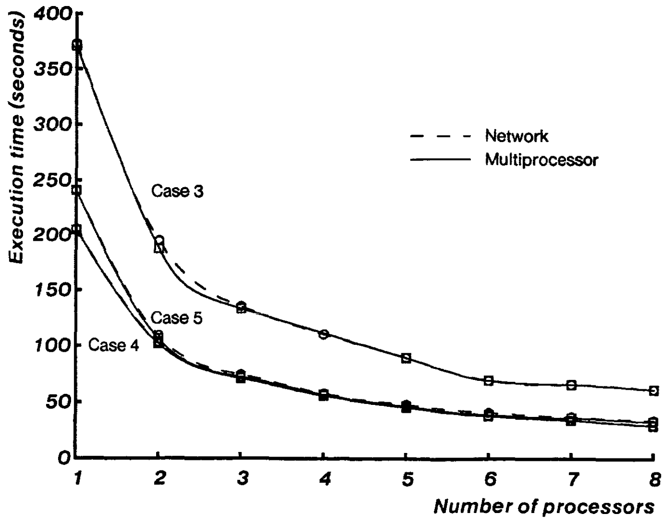
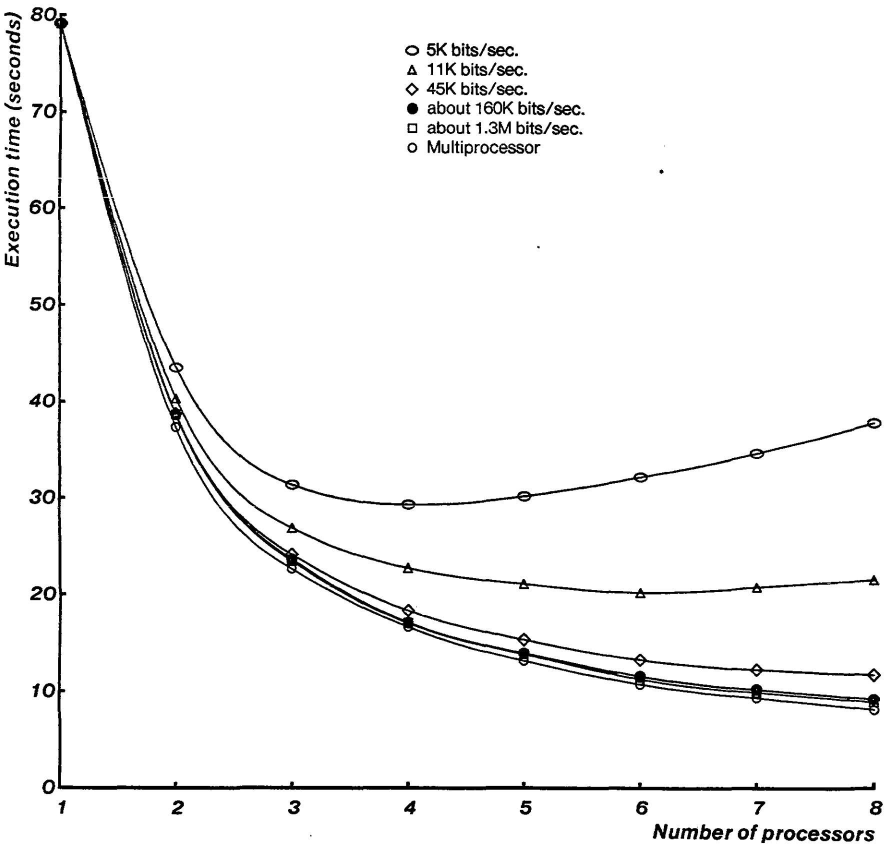


Figure A-12 Effect of Bandwidth on Network Version of Integer Programming



several different application areas, or *task domains*, among which are information retrieval and voice-activated numerical calculations. Only this latter task was implemented on Cm*; it is known as DESCAL (desk calculator). Its vocabulary is 32 words, and it recognizes speech utterances of the form "Alpha gets four times gamma." The task domain is also structured in the form of a weighted directed graph. Both graphs are constructed by a preprocessor program, which runs on a uniprocessor; it is not part of the multiprocessor implementation. The preprocessor determines the arcs and their weights based on the rules of English syntax and its knowledge of the probability of particular utterances in the task domain. For DESCAL, the graph consists of about 1,000 nodes, and the average branching factor is approximately four (the average node has an average of four immediate neighbors).

The *beam search* algorithm employed by Harpy lends itself to parallelization, as it searches several paths through the graph simultaneously, keeping track of the one with the highest probability. If more than one path reaches a terminal node of the graph, the one with the highest probability is selected as the meaning of the utterance.

Harpy was originally implemented on a DEC-10 uniprocessor. The first parallel implementation was built for the C.mmp multiprocessor [Wulf *et al.* 81]. It was later modified for Cm* by Feiler, who used a master/slave implementation in which the master process receives data from another computer (most experiments were performed on prerecorded data) and prepares it for the slaves. Only the slaves participate in the search. In addition, one Cm is dedicated to performing utility routines, so a maximum of six slave processes can be used.

Results. Feiler ran Harpy with one to six slave processors. Speedup varied from 1.87 with two processors to 3.44 with five and 3.60 with six. From experiments on C.mmp, speedup was not expected to increase beyond eight to ten processors. Feiler also experimented with mapped code and data references by the slave processes (see Section A.1). A mapped stack cost almost 14 percent, and mapped code cost about 150 percent.

Further experiments measured the effects of different synchronization mechanisms. The Smap microcode, like the other Cm* microcodes, provides an ***Indivisible Decrement*** operation. This is useful to synchronize updates to, say, a shared stack but is not the best way to implement private semaphores. A process that is busy-waiting on such a semaphore invokes Kmap operations very frequently, which can slow down the Kmap in servicing other requests. A better solution is for the process to busy-wait on a local memory location, as illustrated in Figure A-13.

When synchronized access is required to a large data structure, such as the graphs Harpy uses, one must decide how much of the data structure to lock.Locking a single node is time-efficient because there is only a small probability of a process blocking while waiting to access the node, but it is space-inefficient because of the amount of memory devoted to locks. Locking the entire graph is space-efficient but time-inefficient because only one process at a time can access the graph. In the DESCAL experiment, however, the difference is hardly noticeable until six slaves are used (Figure A-14).

A.5. Harpy on a Simulated Computer Network

Algorithm name: The speech-recognition system Harpy.
Cm configuration*: Eight Cm's, one cluster.
Operating system: Smap microcode.
Other software in environment: None.
Experimenter: Levy Raskin, 1978.
Reference: [Raskin 78].

Raskin experimented with Harpy (see Section A.4), comparing its performance on the Cm* multiprocessor to its performance on a network simulated using Cm* (see

Figure A-13 Local Semaphores vs. Kmap-Implemented Semaphores in Harpy

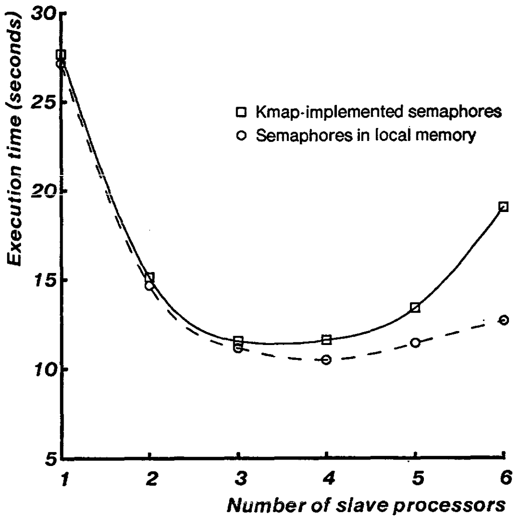
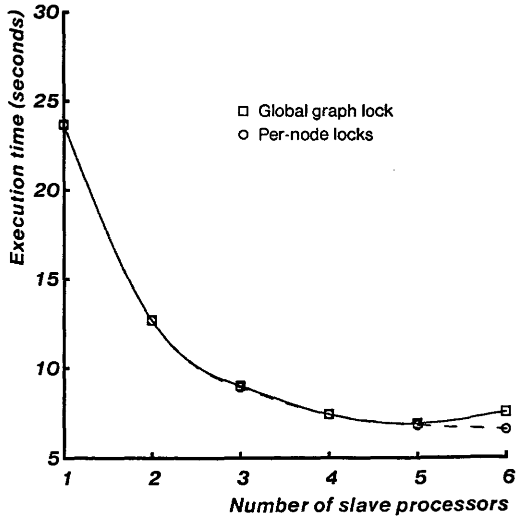


Figure A-14 Locking Nodes vs. Locking Graph in Harpy



Section A.3). Unlike the integer-programming application, Harpy requires a high degree of interprocess communication due to its low branching factor. A naive reimplementation of the multiprocessor algorithm for a network would have resulted in an unacceptably high message rate. Consequently, certain optimizations were made in the network version of the algorithm. All the read-only data structures were replicated in the local memory of each slave processor, the read / write data structures were made local to the processor that was expected to make heaviest use of them, and more of the reconstitution calculations were performed by the master so that some of its processing overlapped with that of the slaves. In the course of the experiments, the same three factors were varied as for the integer-programming algorithm:

- Configuration—multiprocessor vs. network.
- Acknowledge mechanism—simulated hardware vs. software.
- Network bandwidth.

Results. Because of Harpy's higher message traffic compared to the integer-programming implementation, the performance of both hardware- and software-acknowledge network algorithms suffers (Figure A-15). Neither version of Harpy shows a speedup with more than two slave processors. With one slave processor, both of the network algorithms outperform the multiprocessor algorithm. This is explained by the greater overlap between master and slave processors in the network version and by the more optimal placement of the data structures. (No experiments were carried out with data-structure placement on the multiprocessor version. For a detailed consideration of the effects of data-structure placement, see Section A.7.) Figure A-16 shows that message bandwidth is still not critical; it could be reduced by a factor of ten with little perceptible effect on execution time. Nonetheless, a higher bandwidth is required than for the integer-programming example.

A.6. Fast Fourier Transform Experiments Using Algol 68

Algorithm name: Fast Fourier Transform (FFT).
Cm configuration*: Ten Cm's, one cluster.
Operating system: Special-purpose kernel.
Other software in environment: Algol 68 run-time system.
Experimenters: Peter Hibbard, Andy Hisgen, Thomas Rodeheffer, 1977.
References: [Fuller *et al.* 77].

A substantial subset of Algol 68 was originally implemented on C.mmp [Wulf *et al.* 81]. It was derived from full Algol 68 by omitting infrequently used facilities and imposing restrictions to simplify compilation, but it was still somewhat more powerful than PL / I. To take advantage of the opportunities for parallelism presented by the multiprocessor on which it ran, it was extended to include several constructs for specifying concurrent execution and synchronization of processes.

Figure A-15 Network vs. Multiprocessor Implementation of Harpy

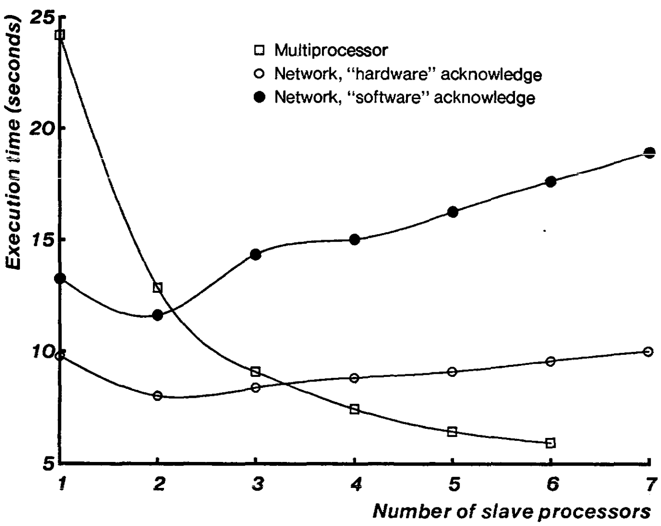


Figure A-16 Effect of Bandwidth on Network Version of Harpy

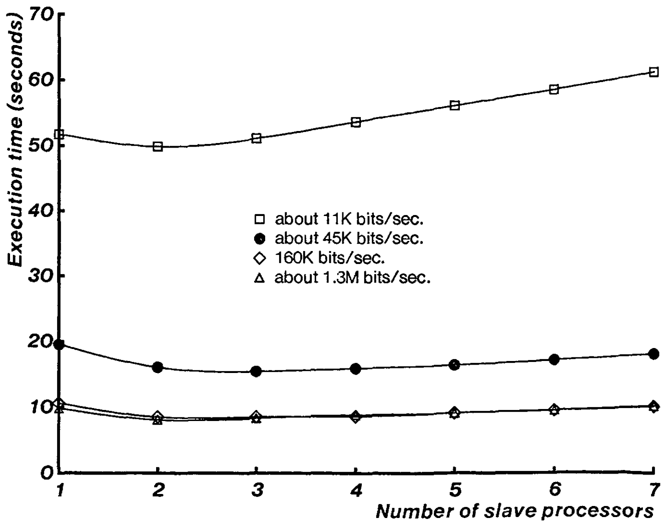
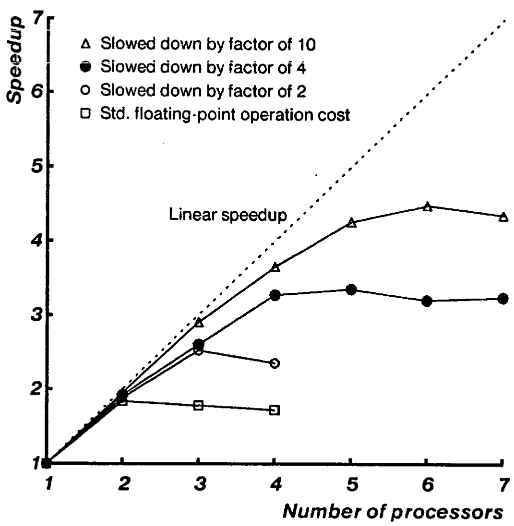


Figure A-17 Speedup of Fast Fourier Transform, Varying Pipelining Overhead



The compiler was then transported to Cm*, and a small special-purpose kernel was written to handle segment allocation, interrupts, and I/O and to collect performance statistics. The objective of the experiments was to study the automatic decomposition of programs into small subprocesses that could execute in parallel. To facilitate this, multiple parallel-instruction pipelines were implemented in software. The "instructions" in these pipelines were the primitive actions of the Algol 68 run-time system, such as floating-point operations, array indexing, and the assignment of large values. A master process placed the actions in the pipeline, and the actions were then executed by the slave processes.

One expects speedup to improve as the overhead of pipeline manipulation decreases. But how can this be tested experimentally? Given the constraints of the Cm* hardware, it is not possible to speed up the pipeline operations to an arbitrary degree. The same effect can be obtained, however, by slowing down the "instructions" in the pipeline as much as desired. Hence, to study the effects of pipelining overhead on speedup, an FFT program was run, with floating-point operations slowed down successively by factors of one (normal speed), two, four, and ten. Figure A-17 shows how speedup varied as pipeline overheads decreased.

A.7. Multicluster Partial Differential Equation Solver

Algorithm name: Partial differential equations (PDE).

Cm configuration*: 50 Cm's, 5 clusters.

Operating system: Smap microcode, StarOS microcode, Medusa microcode.

Other software in environment: NEST.
Experimenter: Jarek Deminet, 1979–80.
References: [Jones and Gehringer 80, Deminet 82].

Deminet modified Raskin's partial differential equation algorithm (see Section A.1) for the multicluster Cm*. All experiments were conducted using method 4 (purely asynchronous), since it was the best-performing of Raskin's implementations. Deminet used three different Cm* microcodes and considered four main issues (which will be listed later). The three microcodes were Smap (Appendix D), STAROS (Section 6.1), and MEDUSA (Section 5.1). These experiments used STAROS and MEDUSA microcode but not the entire STAROS and MEDUSA operating systems. Deminet used none of the software portion of STAROS and only part of the software portion of MEDUSA. Instead, he wrote a small kernel called NEST, which handled communication with the master and slave processors and with the user (see Section 9.1). NEST was capable of running with three different procedure libraries, one each for the STAROS, MEDUSA, and Smap microcodes.

Most of Deminet's experiments used NEST, but a few of the early ones used the "standalone" code inherited from Raskin. The standalone code was more difficult to use because it provided no means of handling queries from the user while the experiment was running. It was also less efficient because the master process was required to field clock interrupts and increment a counter that kept track of execution time, in addition to its regular duties of taking part in the experiment.

Deminet investigated these four issues:

1. The influence of grid size on execution time and speedup.
2. The time "wasted" by processes that had to make many remote references.
3. The placement of the processes that had the most work to do.
4. The distribution of data among several clusters.

Results. THE EFFECT OF GRID SIZE. The experiment was run with varying numbers of processors for grid sizes of 20-by-20, 40-by-40, and 150-by-150, using the Smap microcode. A 20-by-20 grid was the largest that could be accommodated in the standalone version because the grid had to share a 64K address space with the code for the master and the slaves and with the downloading software. The NEST version was used with the larger grid sizes. As shown in Figure A-18, greater speedup was obtained with the standalone version than with the NEST version. It is nearly linear, with a coefficient of 0.77. The performance of the NEST version seemed to deteriorate after 25 processors were reached. With the NEST version, the larger grid size continually showed more speedup than the smaller grid size.

Ironically, the superior speedup of the standalone version is a result of its inefficiency. In the more efficient NEST version, each processor had less work to do, although the grid was referenced the same number of times as by the standalone version. This caused greater memory contention and communication overhead, which eventually led to an increase in run time as processors were added. Figure A-19 compares performance of the standalone and NEST versions when run on grids

Figure A-18 PDE—Grid Size vs. Speedup Using Smap Microcode

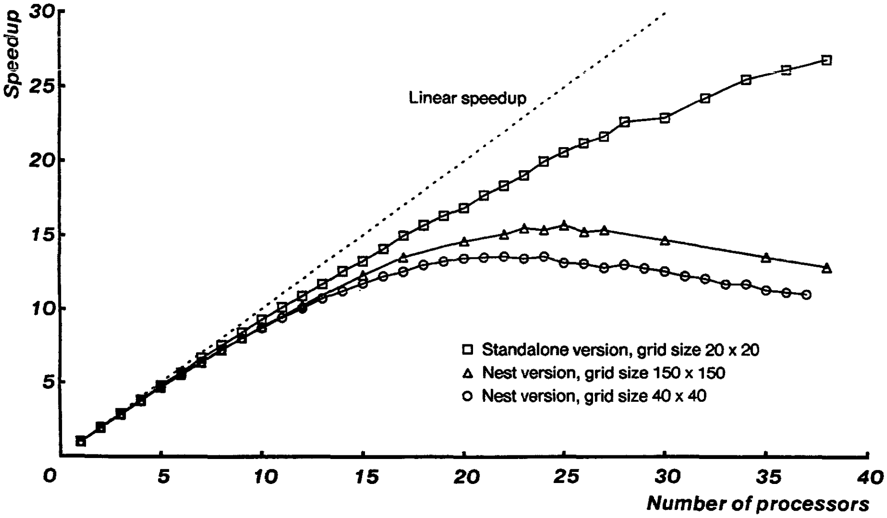
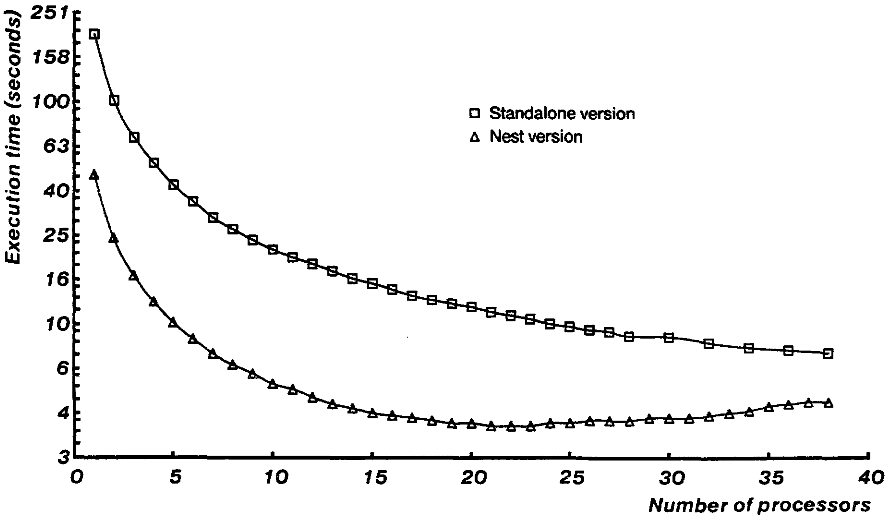


Figure A-19 PDE—Execution Time of NEST and Standalone Versions



of the same size. Note that the vertical scale is logarithmic. The NEST version is up to four times as fast as the standalone version. This is another illustration of the danger of equating speedup with performance.

TIME "WASTED" BY SLOW PROCESSES. The fastest process in the multiprocessor PDE is the process that runs in the Cm containing the global data. As the number of processes is increased, this process tended to show nearly linear speedup;[2] for the Smap version, its speedup was proportional to the number of processors with a coefficient of approximately 0.87. The speedup of the slowest process was at least 90 percent as great as the speedup of the fastest process, as long as only one cluster was in use. As soon as a cluster boundary was crossed, the speedup of the slowest process dropped (due to the performance overhead of slow intercluster references), then increased slightly as more processors were added. Eventually, Kmap contention became significant, and the speedup of the slowest process gradually decreased. The effect was noted using all three microcodes. Its magnitude with a large configuration is shown in Table A-2.

Table A-2 Speed Ratio of Slowest to Fastest Process with a Large Configuration

Microcode	Number of processes	Speed ratio of slowest to fastest
Smap	38	30%
STAROS	36	5%
MEDUSA	37	15%

IMPROVED PROCESS SELECTION. The processes that operate on the outer boundaries of the grid finish in fewer iterations than the processes in the center of the grid. Suppose we placed the processes with the most work—those near the center of the grid—in the same cluster as the grid. These processes would make more references, but this would be offset by the fact that the references take less time. There would be no "worst-case" processes with a large number of slow intercluster references. The task force finishes when its last process terminates, so its execution time would be reduced. This strategy is called *improved process selection*.

Figure A-20 shows that improved process selection made a significant difference when the number of processors was greater than about 16—in other words, when more than two clusters were in use. (These results were obtained using the Smap microcode.) At 35 processes, the improvement was about 20 percent.

[2] "Speedup" is computed as the ratio of the time for one iteration of the uniprocess task force to the time for one iteration of the fastest process in the multiprocess task force.

Figure A-20 PDE—The Effect of Improved Process Selection, Using Smap Microcode

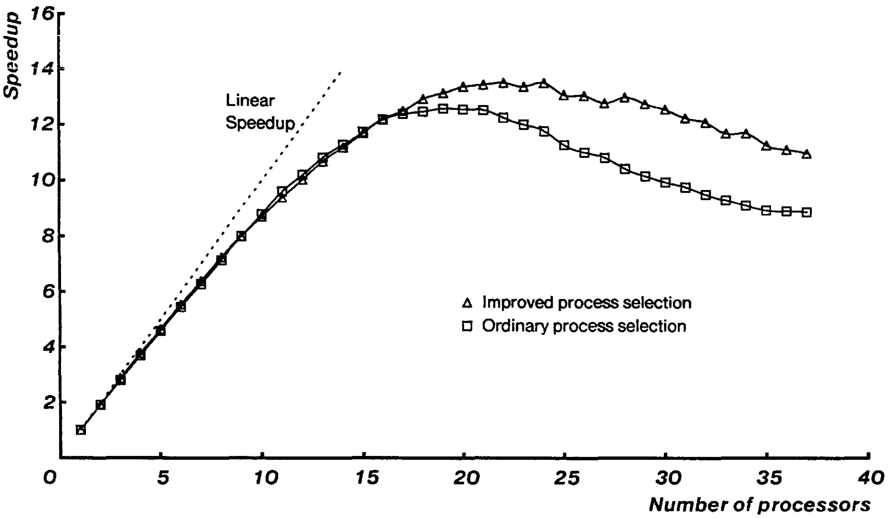


Figure A-21 presents the results of the same experiments but using microcode. The speedup is linear as long as the configuration is unicluster but reaches a maximum at ten processors and drops sharply thereafter. The StarOS microcode provides more functionality than the Smap microcode, and as a result, many of its operations, including intercluster references, take longer. Since these experiments were performed, however, this overhead has been reduced by several optimizations. Improved process selection does make a difference, especially in the region of decreasing speedup. This constitutes additional evidence that the cost of intercluster references is a major reason for StarOS's lower speedup, relative to the other microcodes.

DISTRIBUTED DATA. Another way to improve the performance of the task force is to cut down the number of intercluster references by distributing portions of the grid to all clusters used in the experiment. In Deminet's experiments, a 150-by-150 grid was distributed evenly among these clusters, and processes were assigned to maximize intracluster locality of references.

As illustrated in Figure A-22, the distributed data result was noticeably better at about 16 processors but then dipped between 20 and 21 processors, just as the experiment entered its third cluster.[3] The dip is called the *crossover phenomenon*. It is caused by the fact that the data is uniformly distributed between clusters but the

[3] This experiment used a slightly different Cm* configuration from the previous one, with more processors per cluster. The centralized data experiments were also made with fewer processors per cluster, so the two curves on this graph are not strictly comparable. Nonetheless, at a gross level, the impression they give is accurate.

Figure A-21 PDE—The Effect of Improved Process Selection, Using StarOS Microcode

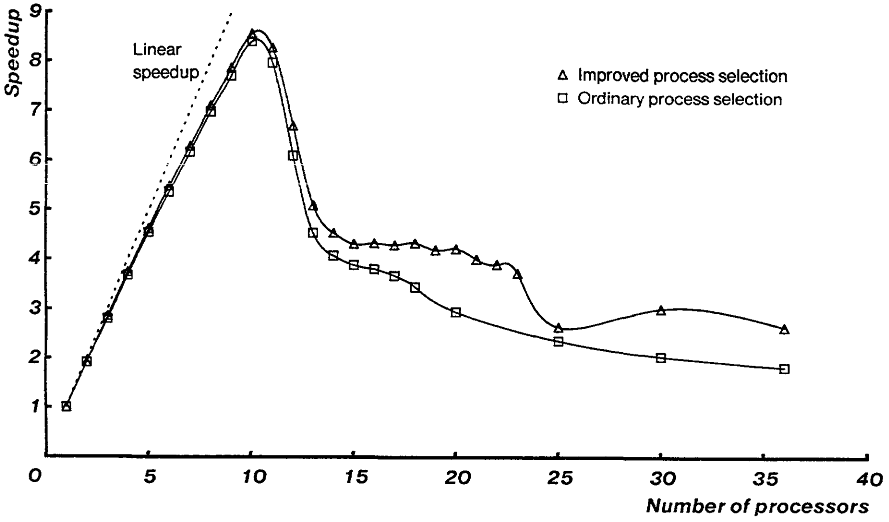


Figure A-22 PDE—Effects of Distributed Data, Using Smap Microcode

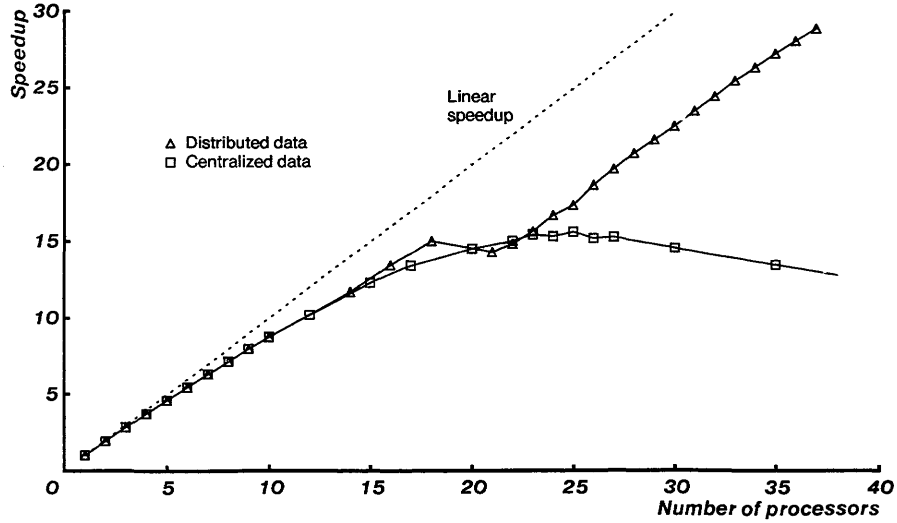
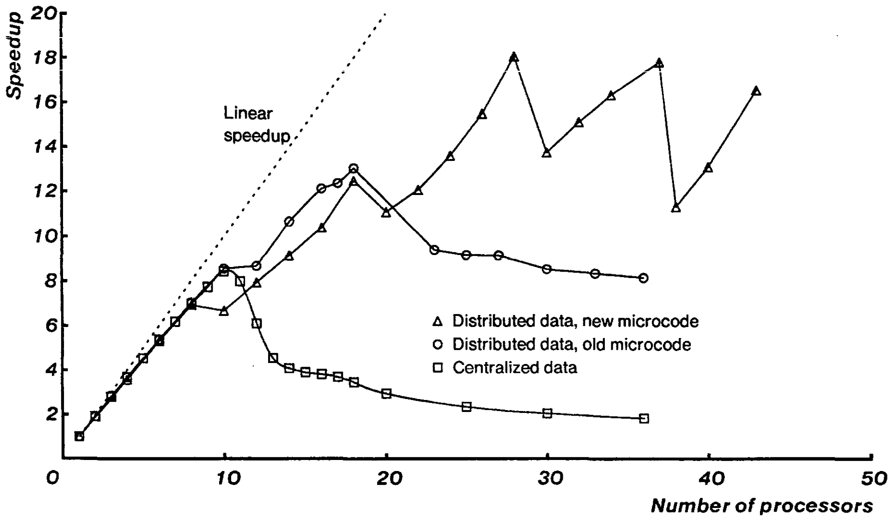


Figure A-23 PDE—Effects of Distributed Data, Using StarOS Microcode



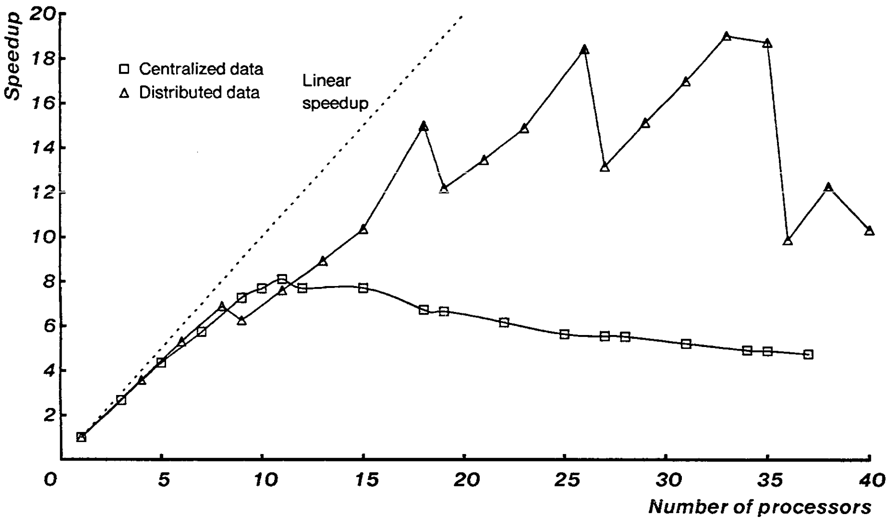
processes are not, causing a larger than usual number of intercluster references. Also note that when data is distributed, the speedup curve continues to increase with greater numbers of processors; it does not reach a maximum and then decline, as with the centralized-data experiments, which are more subject to the effects of contention. One explanation is the fact that the slowest process in the 37-process distributed-data task force runs at about 68 percent of the speed (iterations per unit time) of the fastest one, compared with less than 30 percent in the centralized-data version.

Results with StarOS microcode also show that distributing data helps speedup, although once again, maximum speedup was less than with the Smap microcode. The StarOS microcode was under development during the experimentation period, and Figure A-23 graphs results for two different versions. The "old" microcode is the same version that was used for the ordinary/improved processor-selection experiments. The "new" version incorporated several improvements, notably optimizations of intracluster and intercluster memory references.

The maximum speedup with centralized data (from Figure A-21) was less than 9, whereas with distributed data, it grew to a maximum of about 13 at 18 processors with the old microcode and 18.1 at 28 processors with the new microcode. At that point, the slowest process ran with about 66 percent of its potential speed. The crossover phenomenon is more pronounced here, probably because StarOS intercluster references are about four times slower than intracluster references, compared to three times slower with Smap.

The crossover phenomenon is also conspicuous in the experiments that used Medusa microcode. The best speedup is 19, compared to 18 with StarOS and 26 with Smap (Figure A-24).

Figure A-24 PDE—Effects of Distributed Data, Using MEDUSA Microcode



A.8. Multicluster Quicksort

Algorithm name: Quicksort.

Cm configuration*: 50 Cm's, 5 clusters.

Operating system: Smap microcode, STAROS microcode, MEDUSA microcode.

Other software in environment: NEST.

Experimenter: Jarek Deminet, 1979–80.

References: [Jones and Gehringer 80, Deminet 82].

The quicksort implementation that Deminet used is a modification of Raskin's (see Section A.2) for the multicluster Cm*. Deminet altered the algorithm to cut down on references to the shared stack. Raskin's version had continued partitioning a subset until it had only one element. Deminet's version stopped the partitioning when some *threshold* value—usually ten elements—was reached. If the size of a subset was under the threshold, it was immediately sorted using an insertion sort instead of being partitioned or pushed onto the stack. Deminet studied how the threshold and the size of the array to be sorted influenced execution time and speedup. His experiments were performed on all three microcodes: Smap, STAROS, and MEDUSA.

Results. THE INFLUENCE OF THE THRESHOLD. Using the Smap microcode, the quicksort program was run on a 20,480-element array, using thresholds of 1 to 40. The best speedup was usually achieved with a threshold of 10, but speedup was far below the theoretical limit, especially when more than 8 processors were in use.

Contrast this with the distributed PDE (see Section A.7). In that experiment, adding additional processors always seemed to decrease execution time. This is not true for the quicksort. Performance improved only until 8 to 10 processors were being used (Figure A-25); this is called the *critical point*. With more processors, performance deteriorated. Why did this happen? At first glance, one might assume that it was due to contention for the lock on the global stack, which contains descriptors for subsets to be sorted. If this were true, the critical point should have grown lower as the threshold was decreased, for subsets must then be partitioned more often before the insertion sort is used; consequently, lock operations become more frequent. However, the critical point reached its maximum value of 10 processors with a threshold of 5. When the threshold was increased to 10, the critical point *decreased* to 9 processors; for a threshold of 20, it fell to 8 processors. That indicates that contention for shared memory was a more serious bottleneck than contention for the lock. As the threshold becomes higher, larger subsets are sorted by insertion, and this increases the number of simple memory references to the shared array.

Results for the StarOS (Figure A-26) and Medusa (Figure A-27) versions are similar. In all three cases, maximum speedup was obtained with a threshold of 10. Again, maximum speedup with the Smap microcode (3.86) was slightly higher than with StarOS (3.19) or Medusa (3.25); this can be attributed to the greater functionality provided by the latter two microcodes. Rarely was there a significant performance drop when crossing a cluster boundary. In the Medusa experiments, the critical point for a threshold of 5 was lower than for a threshold of 20. This is the opposite observation from that noted with the Smap microcode, and though the evidence is not striking, it may indicate that contention for the lock is more of a bottleneck than contention for shared memory in the Medusa version.

THE SIZE OF THE SORTED ARRAY. The speedup equation (see Section A.2) predicts that as the size of the array to be sorted increases, so will the speedup, and Deminet's experiments bore this out. Of the three data sizes, the largest (20,480 elements) yielded the greatest speedup, while the smallest (5,000 elements) provided the least (Figure A-28). The results for 5,000 elements are quite irregular, since the time of that experiment was very short, and a small absolute difference could cause a large relative difference.

A.9. Railway-Network Simulation

Algorithm name: Railway-network simulation (Net).
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: Smap microcode, StarOS microcode.
Other software in environment: Nest.
Experimenter: Jarek Deminet, 1979–80.
References: [Jones and Gehringer 80, Deminet 82].

Figure A-25 Speedup of Quicksort with Different Threshold Values, Using Smap Microcode

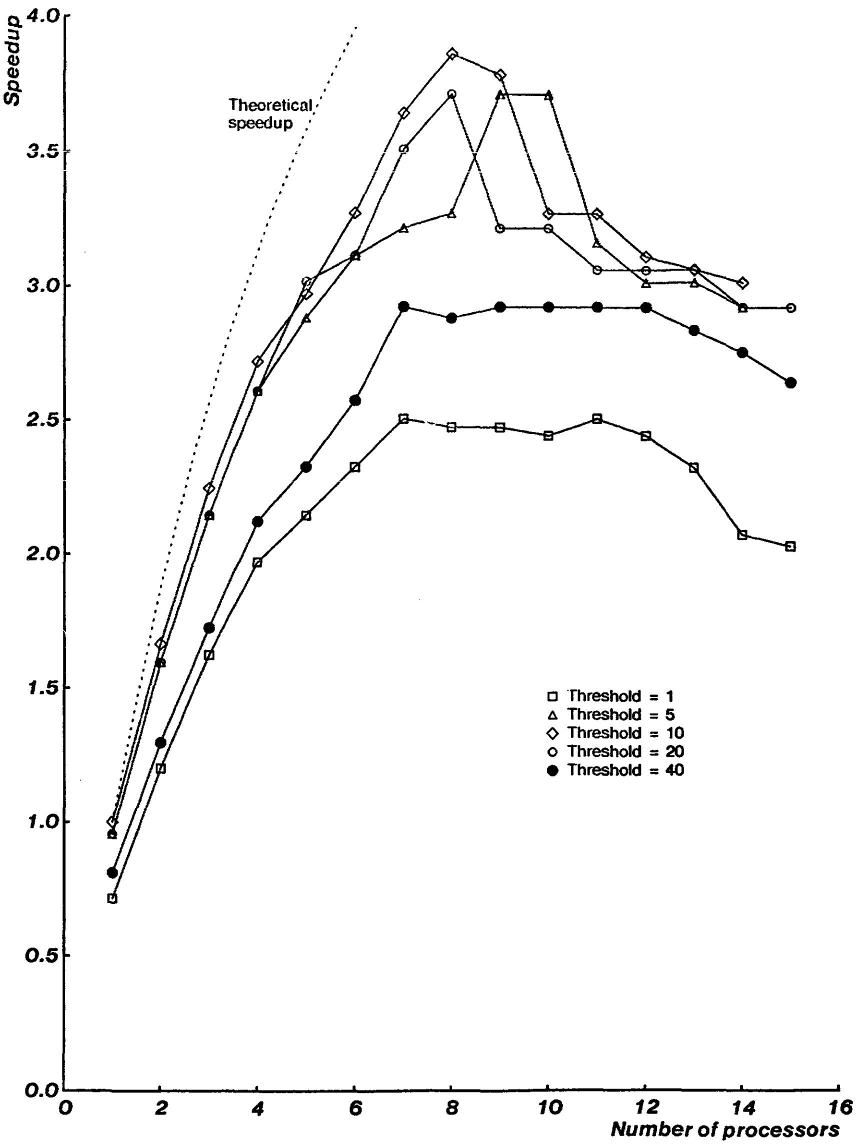


Figure A-26 Speedup of Quicksort with Different Threshold Values, Using StarOS Microcode

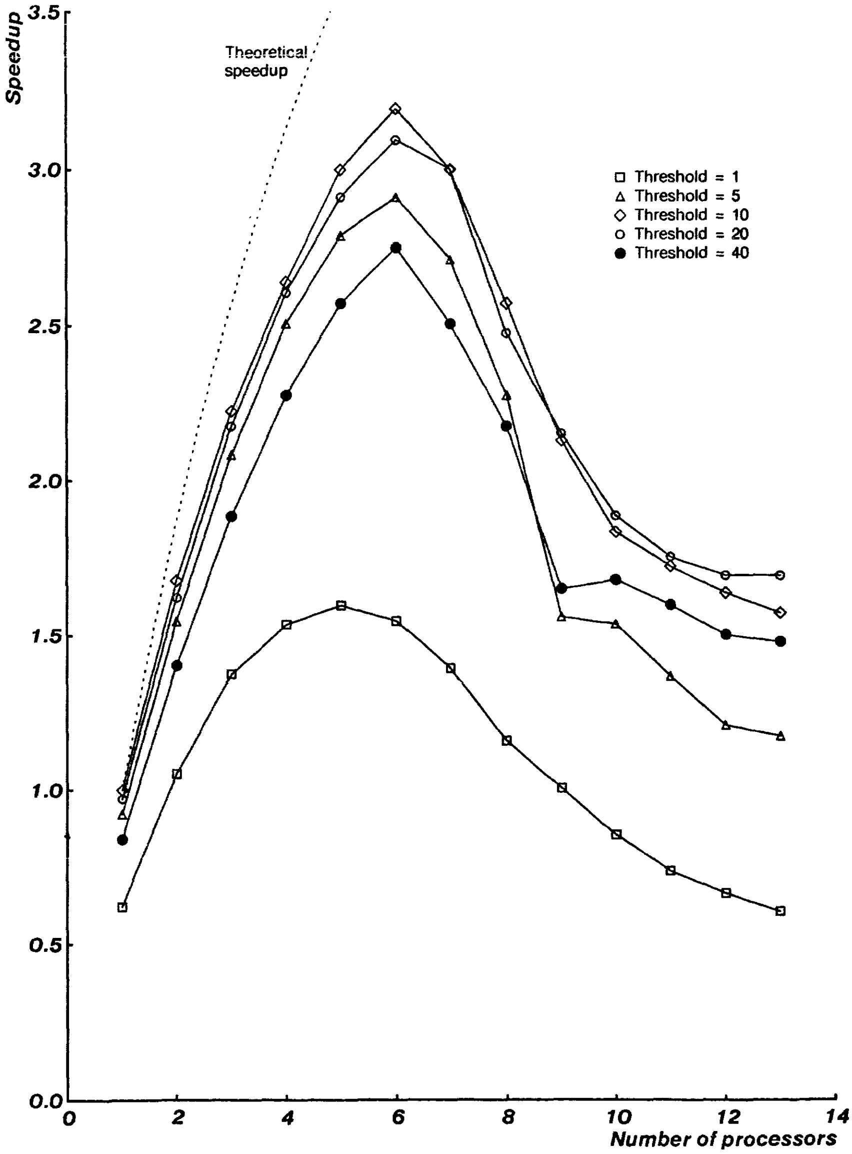


Figure A-27 Speedup of Quicksort with Different Threshold Values, Using MEDUSA Microcode

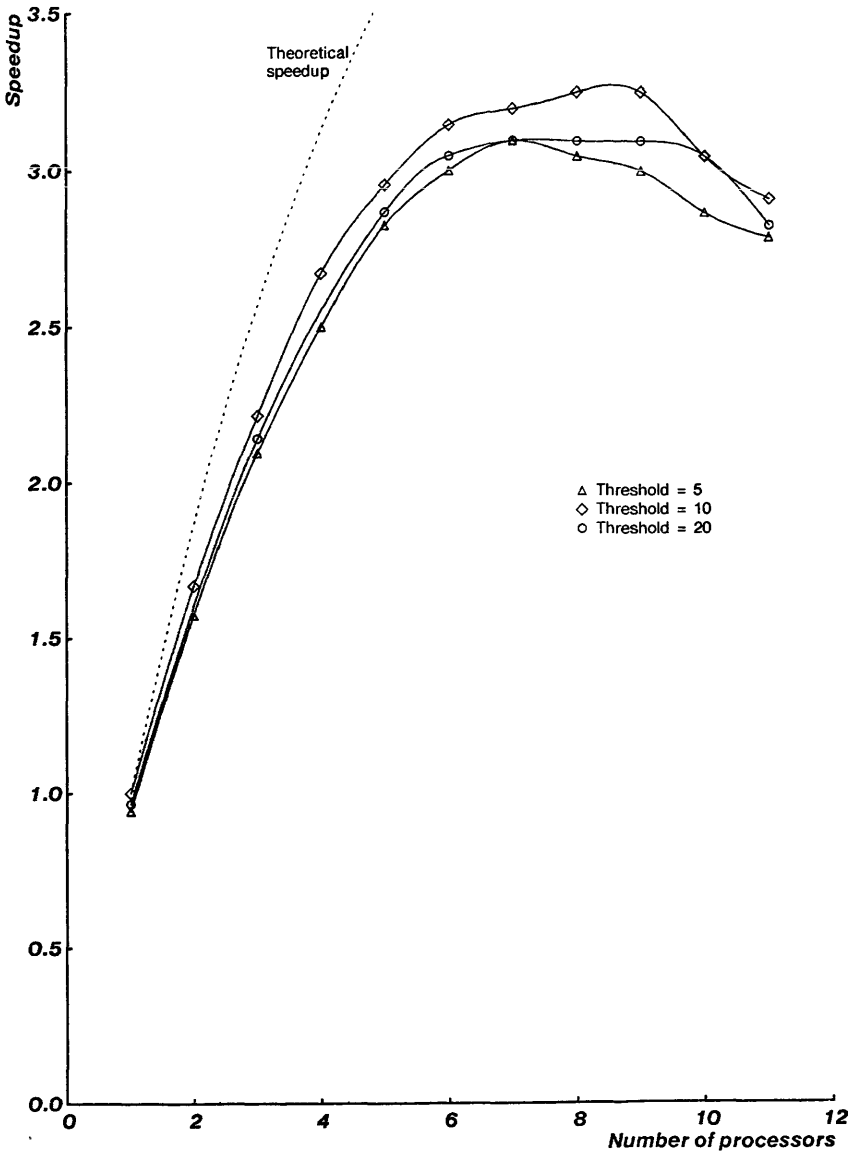
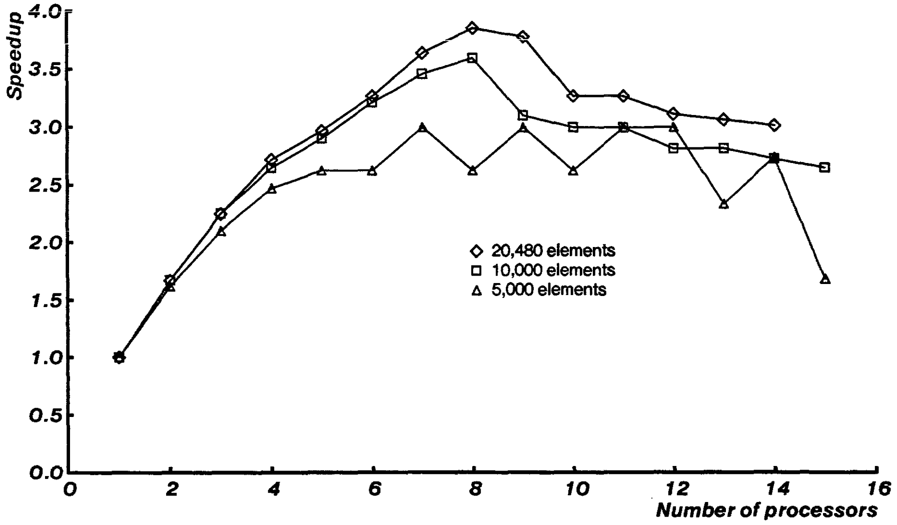


Figure A-28 Speedup of Quicksort for Different Data Sizes, Threshold = 10

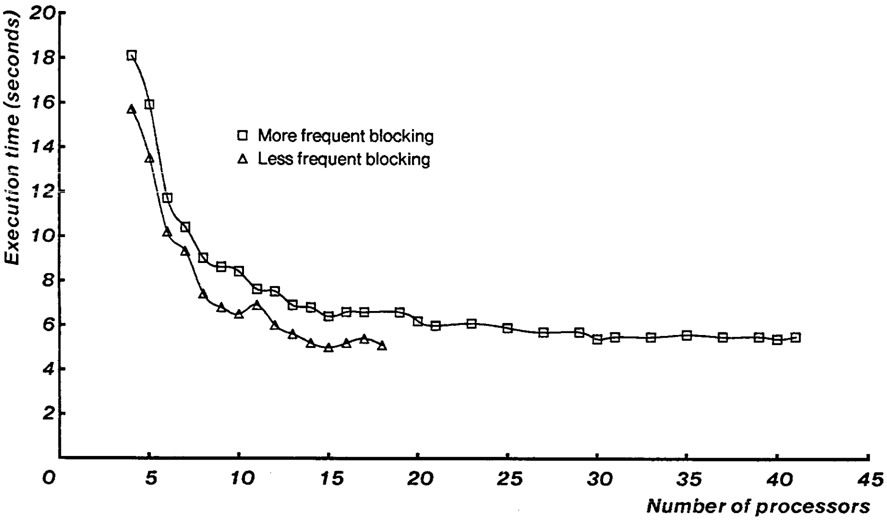


The simulation program is implemented as a task force, comprising 63 processes, each of which represents a *station*. Two stations may be connected by a unidirectional *track*. For a given station *A*, a set of *previous stations* includes each station *B* for which there is a track from *B* to *A*. The stations exchange messages representing *trains*. The route of each train is an attribute of the train, determined when the train is created. Each station serves the trains in the order in which they arrive.

Each process maintains its own simulated time. At any given moment, the simulated time probably will be different in different processes; thus the simulated time of sending a train from one station to another is unrelated to the real time when the message representing the train was created. Even the real-time order of events may be different from the simulated-time order. Suppose, for example, that station *A* sends a message to station *C* at real time 5. At this moment station *A*'s clock may show simulated time 50. Station *B* may send a message to *C* at real time 7, but its clock may then show simulated time 40. The second message should be serviced by *C* before the first one, since only the simulated time is relevant. Consequently, a station-process must block until it knows the simulated time of the arrival of the next train from each station.

Several station processes may run on each processor. There exists one additional process, the *reporter*, which records data sent to it by the stations. The reporter must be running at all times. If it were multiplexed, all other processes

Figure A-29 Execution Time of Railway-Network Simulation, Using Smap Microcode

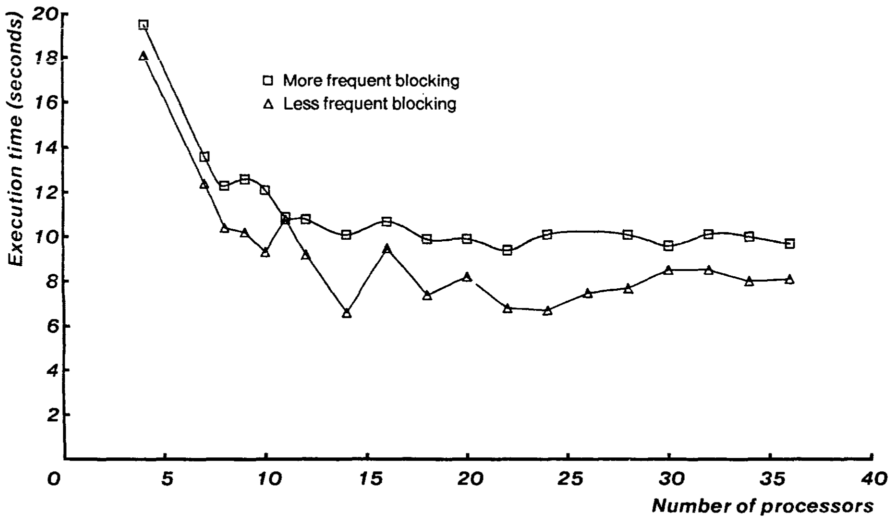


would be blocked. As a result, this application may not be run on a single processor. In fact, this task force will fit in a minimum of four Cm's.

Results. It is very difficult to estimate speedup for this algorithm because running time depends not only on the number of processors but also on the distribution of processes among them. In the implemented version, processes may not move from one processor to another. If they did, it probably would be necessary to move their private data to avoid the penalty of remote references. To move data would itself impose an overhead. Thus the run time of the experiment depends heavily on the initial assignment of processes to processors. Figure A-29 graphs the run time versus number of processors, using the Smap microcode. The curves plot the results for two different versions. The versions differ in one parameter that determines the travel time of trains between stations. It influences the average number of processes blocked at any given time. The region at the left of the graph is steeply decreasing because the average number of runnable processes is greater than the number of processors. Farther to the right, the curves exhibit apparently random fluctuations, perhaps due to variations in the suitability of the initial assignment of processes to processors. As expected, the version with less-frequent blocking outperforms the other version.

Figure A-30, which shows the results using the StarOS microcode, is less regular than the previous one. One explanation may be that the relative position of two communicating processes may influence performance more heavily with the StarOS microcode, whose intercluster references cost more.

Figure A-30 Execution Time of Railway-Network Simulation, Using StarOS Microcode



A.10. Power-Systems Simulation

Algorithm name: Power-systems simulation.
Cm configuration*: Ten Cm's, one cluster.
Operating system: StarOS (version 1).
Other software in environment: None.
Experimenters: R. C. Dugan, Ivor Durham, and Sarosh Talukdar, 1979.
References: [Dugan *et al.* 79, Durham *et al.* 79].

The simulation of large electrical networks is the subject of extensive work in the power-systems industry. The Cm* implementation utilizes the *network model*, which exhibits inherent parallelism in stages that account for more than half of its time. An electrical network is modeled hierarchically; it is composed of a set of devices, each of which may be made up of more primitive devices. A nonprimitive device is called a *macrodevice*. Any primitive device or macrodevice may be characterized by the behavior of voltage and current values at its terminals. The mathematical model for a device is called its *macromodel*. Experience with an earlier uniprocessor implementation [Talukdar 79] indicated that more than 50 percent, and up to 97 percent, of the time was spent solving the macromodels.

The algorithm performs two steps iteratively. Phase I solves the macromodels. This phase is susceptible to parallelism. Each Phase I processor repeatedly extracts an unprocessed device from the *pool* of unprocessed devices. The new voltage and current values for the device's output terminals are computed from the corresponding values for its input terminals, according to the macromodel. The resulting voltages and currents form part of a single linear system. When all the devices have been processed, Phase II begins. It solves the linear system. The Phase I / Phase II cycle computes the results for a single *time step*. Then the cycle is performed again for the next time step and continued until the simulation is finished.

The experiment utilized a nine-processor Cm* cluster, with one process per processor. One of the processes was the *management process*, which coordinated the simulation. Among other duties, it was responsible for placing the devices back into the device pool at the start of each iteration. A *communication process* was used to ship data and results back and forth between Cm* and a PDP-10 computer. This allowed the user interface and report programs to take advantage of the editors and other utilities that comprise the more mature software environment of the PDP-10. The Phase II process, and up to six Phase I processes, occupied the remaining processors. While the Phase I processes were executing, the Phase II processor serviced another device pool containing the single voltage source required in the sample network. All interprocess communication was via shared variables rather than message passing.

The device pool was implemented as a set of StarOS basic objects with a counting lock. The management process "put" the devices into the pool by resetting the counting-lock value to the number of devices in the pool. A Phase I process selected a device from the pool by performing an ***Indivisible Decrement*** on the counting lock. The result was used to index into the set of capabilities for the basic objects representing the devices. When the lock value became zero, there were no more devices left to process.

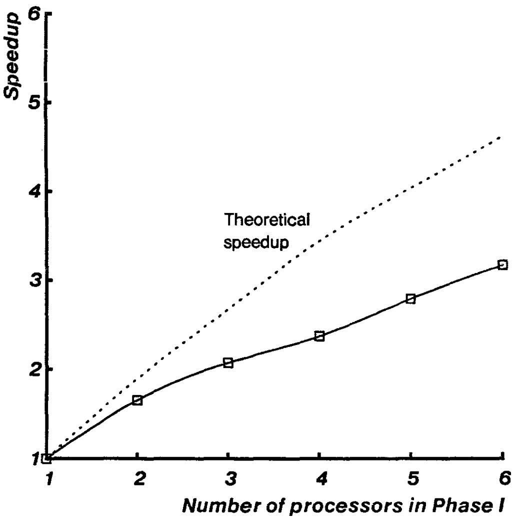
The theoretical speedup of this algorithm is less than linear because only one of its two phases is parallel. Assuming that the solution of the linear system in Phase II takes a constant amount of time and that there is a certain fixed overhead in each step of the iteration, the theoretical speedup is given by

$$E_N = \frac{T_{Phase\ I}}{N} + T_{Phase\ II} + T_{Overhead}$$

The theoretical speedup is then computed as E_1 / E_N, where E_1 is the time required to perform the computation with only one Phase I processor. A single-phase, 100 pi-section transmission line was simulated [Dugan *et al.* 79]. The actual speedup, as shown in Figure A-31, is much lower. This is due partially to memory contention, which is not accounted for by the model, and also to the fact that as the number of processors in Phase I increases, so does the frequency of nonlocal references to the device pool.

The execution time of the simulation on Cm* can be compared with the execution time of a strictly uniprocessor simulation run on the PDP-10. Ignoring I / O costs for both versions, Cm* could execute about 0.93 iterations per second with one Phase I processor, while the PDP-10 could execute about 4.36 iterations per second. With six Phase I processors in action, the rate rose to about 3.0 iterations per second.

Figure A-31 Speedup of Power-Systems Simulation



Considering the slowness of LSI-11 floating-point operations, the Cm* simulation performed quite well.

A.11. A New Implementation of Power-Systems Simulation

Algorithm name: Power-systems simulation.

Cm configuration*: 50 Cm's, 5 clusters (only 1 or 2 clusters used for these experiments).

Operating system: Smap microcode.

Other software in environment: NEST.

Experimenter: Mike Carey, 1980.

References: [Carey 80, Talukdar *et al.* 81, Talukdar *et al.* 82].

The original goal of the power-systems study was to determine how much the simulation could be speeded up using a multiprocessor. It partitioned the system into a fixed number of macrodevices. Since a given power system can be partitioned into different numbers of macrodevices, one might also study how the decomposition of the system affects the performance of the model. A reimplementation of the power-systems experiment made it possible to vary this parameter and others to study the effect on execution time. The new version was less sophisticated than the original in two respects:

1. In the original version, the Phase I processes selected macrodevices from the shared pool whenever they needed more work to perform. In the new version,

each Phase I process worked on one and only one macrodevice. Thus the number of macrodevices was equal to the number of Phase I processes and the number of Phase I processors.

2. To save time in Phase II, the original version used band-matrix techniques and LU decomposition, but the new version used full-matrix techniques and Gaussian elimination.

Carey [Carey 80] describes several experiments using the new system. We will consider two of them. The first experiment measured the time required to simulate a single time step, and the second measured the time consumed by a single Phase I iteration. Both these experiments varied two factors: the number of Phase I processes and the number and type of nonlocal references.

Each memory reference made by a process falls into one of four categories: code, process stack, private data, or global data (see Section A.1). Global data is always shared and hence almost always nonlocal. In these experiments, the process stack was always local. The private data—vectors and matrices—used by the Phase I processes could be placed either in the local Cm or in some other Cm, as could the procedure code.

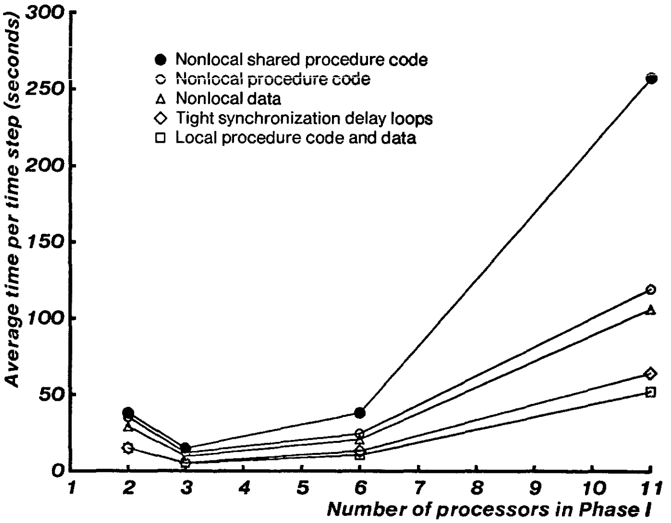
As other experiments have demonstrated (Sections A.1, A.2, and A.3), when more than three or four processors were used, contention for global data seriously impeded performance. During each iteration of the simulation, the Phase I processes busy-wait for the Phase II process to finish, and vice versa. This leads to high contention, which can be lessened by decreasing the frequency of reference to global data that records how many Phase I or Phase II processes have finished. This was done by inserting a delay loop of about 100 μs. between consecutive references to these global variables.

Carey studied five *configurations* of local and nonlocal references by varying these factors:

1. *Local process code and data.* Code and private data resided on the same Cm that was executing the Phase I process.
2. *Tight synchronization delay loops.* Code and private data were still local, but the 100-microsecond interreference delay was removed.
3. *Nonlocal data.* The delay loop was used, but private data was placed in a remote Cm.
4. *Nonlocal code.* The delay loop was used, and private data was local, but the Phase I processes executed nonlocal code.
5. *Nonlocal shared code.* Like configuration 4 except that the Phase I processes shared a single copy of the code.

The network simulated in the first experiment consisted of eleven devices. Ten were *pi-sections* (details may be found in [Carey 80]), and the eleventh was a nonlinear resistive load. In different runs, the number of Phase I processes was varied. One Phase I process was always dedicated to the nonlinear resistive load. The other Phase I processes serviced equal numbers of pi-sections; there were

Figure A-32 Execution Time per Timestep in Power-Systems Simulation



either one, two, five, or ten of them servicing, respectively, ten, five, two, or one device each. Thus the total number of Phase I processes ranged from two to eleven. The only other process in the system was the single Phase II process.

Not surprisingly, the results (Figure A-32) show that configuration 1 is the fastest for any number of Phase I processes. When the number of Phase I processors in the other configurations increases past three, contention exacts its toll. Configuration 2 is only 5 percent slower than configuration 1 with three Phase I processors, but it is 24 percent slower with six and 23 percent slower with eleven processors.

Configurations 3 and 4 exhibited mapping overhead. Configuration 3 had an advantage because private-data references were less frequent than code references. Its advantage over configuration 4 did not, however, grow as the number of processors was increased; in fact, it fell from a high of 25 percent at three processors to a low of 12 percent at eleven. The overhead of mapped code (configuration 4 compared to configuration 1) also declined slightly as the number of Phase I processors increased, from 136 percent with two processors to 88 percent at five and 86 percent at eleven.

Configuration 5 shows the effects of both mapping and contention overhead. The exceptionally large rise between six and eleven processors may be attributed to two factors. The first is intercluster references. Given the Cm* configuration at the time of the experiments, not all eleven processes would fit into the same cluster. Some references to the code had to be intercluster, increasing the mapping overhead. The second factor is contention for the Kmap. When the number of processors seeking Kmap service rises above seven, a context cannot be allocated immediately (see Section 2.2.2), introducing a further source of delay.This effect is much less visible without shared memory because memory references complete much faster; therefore, contexts become available much faster and the system is unlikely to run out.

For all configurations, the optimal number of processors is three. This is due to the fact that full-matrix techniques were employed in Phase II, causing the complexity of this phase to be $O(n^3)$ in the number of macrodevices. In these experiments, each pi-section Phase I process dealt with the same number of devices, so the complexity of Phase II grew with the cube of the number of pi-section processors. Phase II execution time quickly dominated the overall simulation time, which began to increase when the number of Phase I processes was increased past three. By contrast, in the earlier implementation (see Section A.10), the number of devices in the system was constant, and so was the execution time of Phase II.

The second experiment introduced another variable—the distribution of processes and data among clusters. In some runs, the code or data of Phase I processes was placed in a different cluster from the Cm on which the processes were running. Each of the Phase I processes, except the nonlinear-load process, simulated an eight pi-section macrodevice. In Figure A-33, four of the curves are coincident, namely the curves for configurations 1 and 2 with intracluster or intercluster references. This is because each Phase I iteration, which takes several seconds, makes only four references to global data. In these configurations, only the global data was ever intercluster, and the reference rate to it was so low that it made no perceptible difference in execution time per iteration. Similarly, the low reference rate to global data rendered insignificant the presence or absence of the 100-microsecond interreference delay.

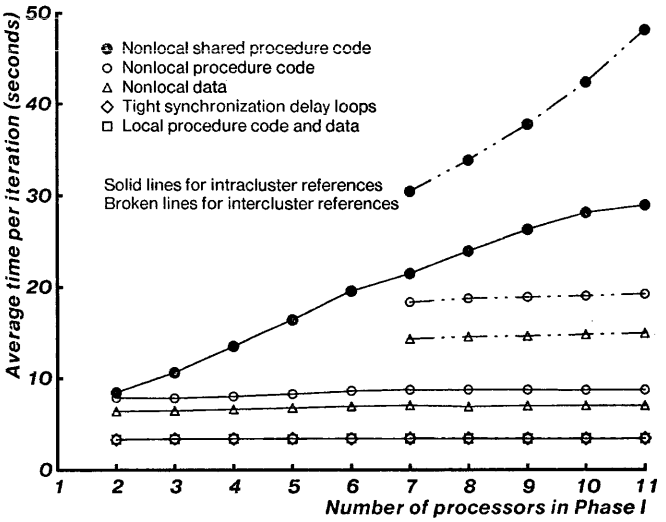
In the intercluster experiments with configurations 3, 4, and 5, not only the global data but also the private data and / or procedure code was in a different cluster from some of the processes. The curves in Figure A-33 marked "intercluster references" plot the execution time for those processes. Here, the intercluster memory-reference rate is high enough to induce significant performance degradation.

Except for the two shared-code curves, all the curves in this figure are essentially flat. This indicates that time per iteration is independent of the number of Phase I processors, which is true because the processors do not make enough references to global data to interfere with each other. However, with shared code, the iteration time rises as the number of processors is increased. At ten processors, it is 8.6 times as high as with local code (configuration 1) when the shared code is cluster-local and 14.3 times as high when the shared code is intercluster.

A.12. AMPL Partial Differential Equation Solver

Algorithm name: Partial differential equations (PDE).
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: MEDUSA.
Other software in environment: AMPL run-time system.
Experimenter: Roger Dannenberg, 1980–81.
References: [Dannenberg 81].

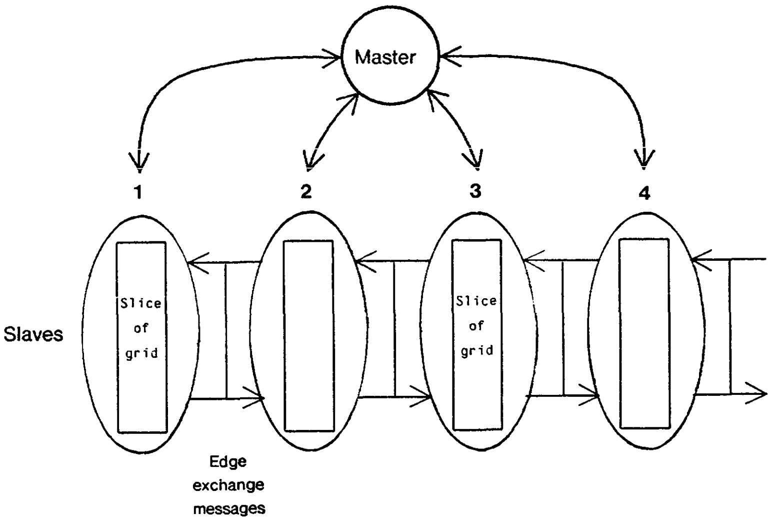
Figure A-33 Time per Phase I Iteration in Power-Systems Simulation



Dannenberg's partial differential equation (PDE) solver was written in Ampl (Section 8.2). It uses the finite difference method to solve Laplace's PDE with specified boundary conditions. Two versions of this algorithm were implemented. The first uses synchronous communication. The grid is divided into slices, and a slave process is created to compute the solution for each slice (see Figure A-34). (This organization is similar to, but not quite the same as, the one that Raskin [Section A.1] and Deminet [Section A.7] used.) The slave processes share the values on the edges of the slices. After finishing its current iteration, a processor communicates the values on the edge to its neighbor. Synchronization is maintained only by waiting on edges, so neighboring slaves are synchronized to within only one iteration. In a program with *n* slaves, if the first slave is computing iteration *i*, then the last slave could be computing any iteration from *i* – (*n* – 1) to *i* + (*n* – 1). To help detect when convergence has been achieved, each slave sends a status message to a master process after each iteration.

Dannenberg varied both the number and the placement of slave processes. In one set of experiments, the number of slaves was varied, and all the slaves ran on a single CP/AP pair (page 184). The measurements reveal the cost of dividing the grid among communicating processes while keeping the processing power constant. In the other set of experiments, the number of CP/AP pairs was varied, but each pair contained a single slave. This permitted measurement of the speedup obtained by adding additional processors. In all cases, the grid size was 34 by 34. Figure A-35(a) (white bars) displays the elapsed time to execute the experiments, and Figure A-35(b) shows how much CPU time was used by all the processors. These

Figure A-34 Structure of Dannenberg's PDE Program



values do not include the time to create processes or initialize the grid. Total CPU time increases slightly as processors are added; a maximum speedup of 3.1 is achieved with four processors.

An extra CP / AP pair was used for the master process, which creates the slaves and detects when the computation has converged. Keeping the master in a separate processor pair facilitated measuring the processing time taken by the master. In the worst case (with four slaves), the master uses only 3 percent of the total CPU time.

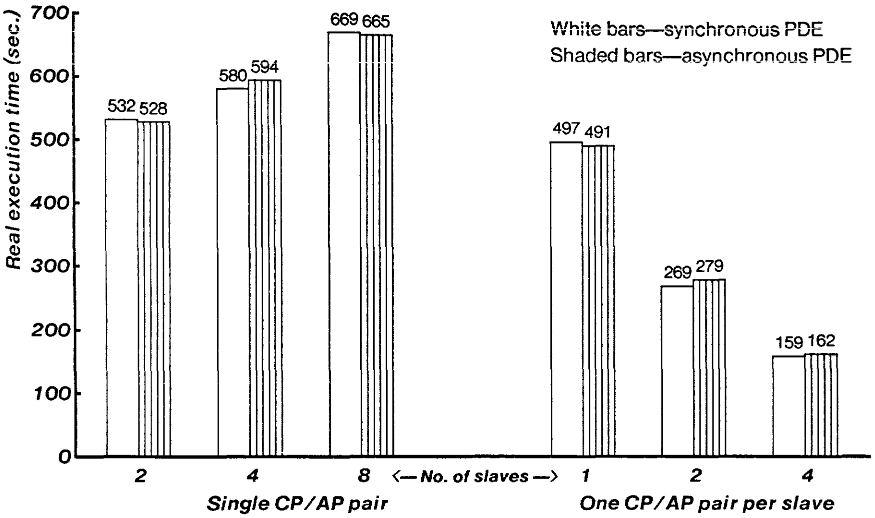
The white bars in Figure A-36 show how many iterations were needed for the computation to converge for various numbers of slaves. Since the computation is synchronous, the number of iterations is not affected by the placement of the slaves. The number generally increases as the number of slaves increases because the rate of convergence depends on the order in which values of the grid are updated.

The amount of data accessed within each slave can easily be compared to the amount of data transferred in messages. The four neighbors of the current point are read to compute its new value, and the current point itself is read to test for convergence. Thus at any interior grid point, each iteration reads five data points from the grid and writes one. Hence there are a total of

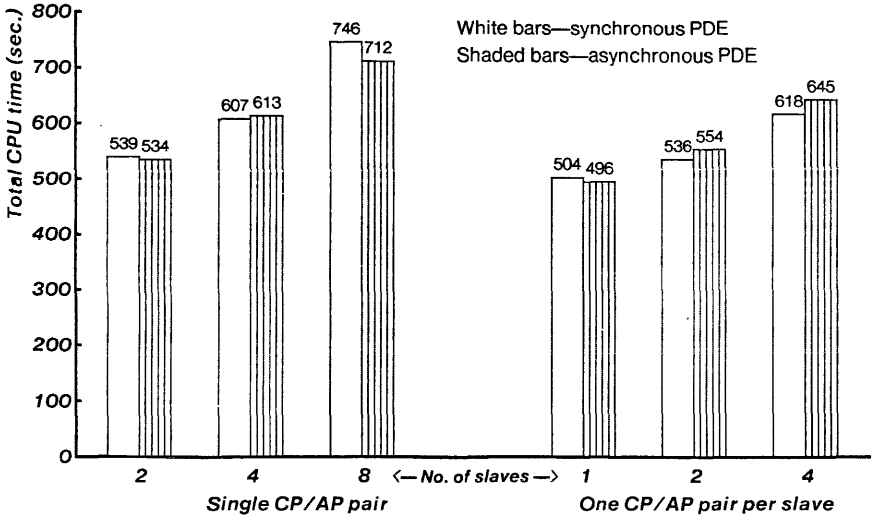
$$(5 + 1) \times 32 \times 32 = 6{,}144$$

local data accesses per iteration. Table A-3 expresses the number of edge points passed in messages as a percentage of this value. Message overhead increases as

Figure A-35 Performance of AMPL PDE

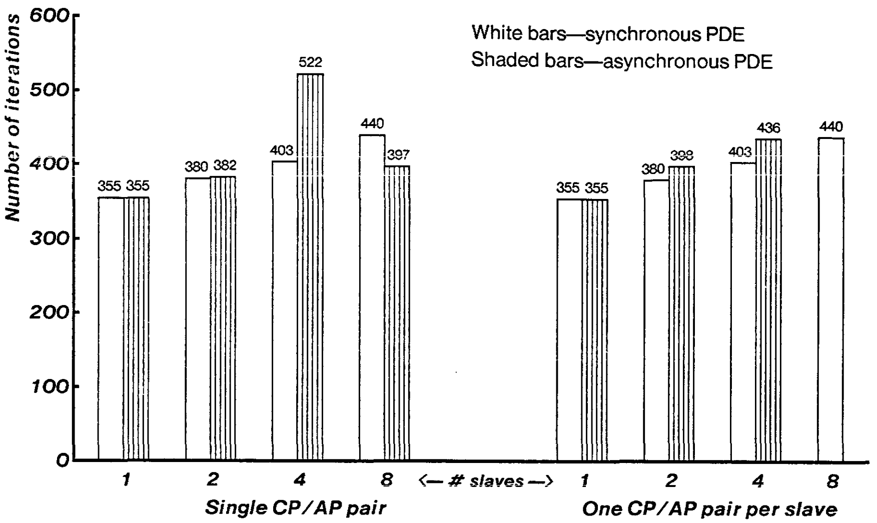


(a): Elapsed Time for AMPL PDE Execution



(b): CPU Time Used in AMPL PDE Execution

Figure A-36 Number of Iterations for the Ampl PDE to Converge



the partition size decreases. In other words, overhead increases with the number of slaves.

To gauge the effect of code quality on the results, the inner loop of the slave program was manually optimized by modifying the Bliss-11 code produced by the Ampl compiler. Subscript range checking was removed, and strength reduction was performed to eliminate multiplications in subscript calculations. Both these transformations could be performed by a modern optimizing compiler. For the version with two slaves and a single CP/AP pair, the execution time dropped from 531 to 75 seconds, a sevenfold improvement. A small amount of time also could have been saved by eliminating debugging statements, error-detection code, and instrumentation, all of which are outside the inner loop.

In Dannenberg's asynchronous version of the PDE, each slave checks its ports for new edge values at the beginning of an iteration. If a message has arrived, the slave uses it to update its grid. If no new edge values are present, the slave

Table A-3 Relative Message Overhead in Ampl Synchronous PDE

Number of slaves	Message overhead
2	1.2%
4	3.4
8	8.0

performs an iteration using the old values instead of waiting. The shaded bars in Figures A-35 and A-36 illustrate the behavior of the asynchronous PDE. One egregious point in Figure A-36 is immediately apparent: When the four-slave version is run on a single processor, an average of 522 iterations are required, compared to an average of 397 iterations if the number of slaves is doubled or 436 iterations if the four slaves are dispersed on separate processors.

The explanation for this behavior is simple. A FIFO ready-to-run queue is used to schedule processes. A process is selected from the queue only when another process is suspended after sending a message, initiating a *create* operation, or trying to accept a message from an empty port. The slaves on either edge of the grid have only one neighbor and send only one message after each iteration. Each slave in the interior of the grid has two neighbors and sends two messages per iteration. The interior slaves are therefore suspended twice as often as the two edge slaves and thus execute about half as many iterations when all processes reside on a single CP/AP pair. The effect is maximal with four slaves, since half are edge and half are interior slaves.

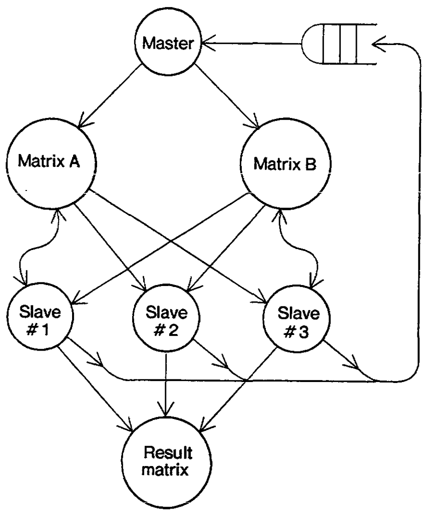
It is surprising to observe that the synchronous PDE generally outperforms the asynchronous PDE in execution time and speedup, although the margin is small. In all previous PDE experiments, the asynchronous PDE has proven more efficient by a large margin. At first glance, one might assume the anomaly is due to the extra iterations, and hence extra messages, required by the asynchronous version. The cost of message passing, after all, is much higher than the cost of lock operations used in the other PDE implementations. But message passing does not contribute much to the overall execution time; even for the four-slave system, it amounts to only about 20 seconds total of execution time. Rather, the explanation lies in noting that the "synchronous" PDE is not quite as synchronous as Raskin and Deminet's; it requires a processor to synchronize with only one or two neighbors, not with all other processors after each iteration, as in the synchronous Jacobi method used by the other experimenters. However, Dannenberg's "asynchronous" implementation is essentially the asynchronous Jacobi method—the second slowest—of Raskin and Deminet. Since the difference in synchronism is slight, the direct and indirect (e.g., copying edges) cost of the extra messages sent by the asynchronous version may be enough to tip the balance against it.

A.13. Matrix Multiplication in AMPL

Algorithm name: Matrix multiplication.
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: MEDUSA.
Other software in environment: AMPL run-time system.
Experimenter: Roger Dannenberg, 1980–81.
References: [Dannenberg 81].

A parallel matrix-multiplication algorithm was implemented in AMPL (Figure A-37). The algorithm uses several different kinds of processes:

Figure A-37 Structure of Dannenberg's Matrix-Multiplication Program



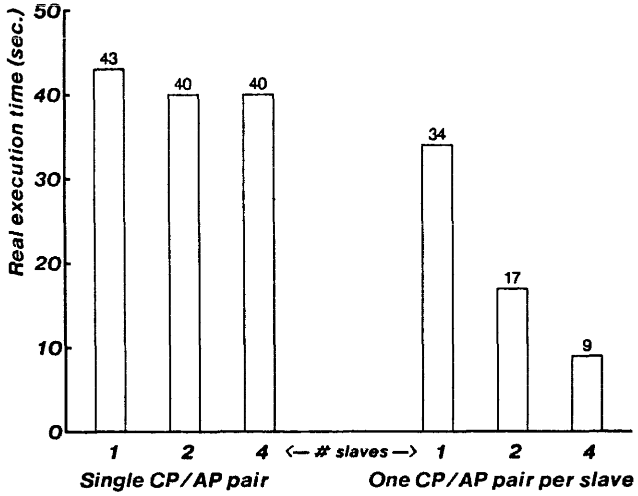
- Two *matrix* processes initially hold the two matrices to be multiplied. They divide the matrices into some number of submatrices (16 in these experiments). On command, the matrix processes send a submatrix to a *slave* process.
- A *master* process commands the two matrix processes to send submatrices to a slave, whenever the slave is ready for more work.
- The *slave* processes multiply the submatrices they have been sent. When a slave has finished, it sends the result submatrix to a *result* process and sends its name to the master, which then assigns it more work.
- The *result* process accepts submatrices from the slaves and assembles and sums them to form the complete product matrix.

Notice that the slaves' work assignments are nondeterministic and depend on their relative speeds. In these experiments, the number of slaves was varied from one to four and the matrix size was held constant at 40 by 40. The same sets of experiments were run as for the AMPL PDE. In one set, the number of slaves was varied from one to four, while all processes—master, slave, and otherwise—ran on a single CP/AP pair (see page 184). In the other set, the number of processors varied from one to four, with each CP/AP pair containing a single process. In this set of experiments, additional processors were used for the master, matrix, and result processes. In all cases, measurements were started after all processes were created and ended after the complete product matrix had been formed.

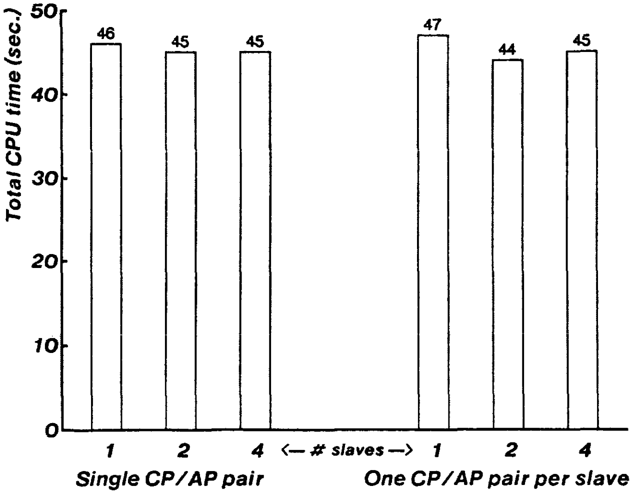
Figure A-38 shows real execution time and CPU time for both sets of experiments. Included in these values are about 10 seconds of CP and AP time consumed

Figure A-38

Performance of AMPL Matrix-Multiplication Algorithm



(a): Elapsed Time for AMPL Matrix Multiplication



(b): CPU Time Used in AMPL Matrix Multiplication by the master and result processes and 2.3 seconds used by the two matrix processes. Buffering in the result process allows the slaves to run without blocking, even with high utilization of the CP/AP pair on which the result process executes. As the number of slaves grows large, the execution time of the result process would come to dominate the overall performance; it could also be decomposed to compute its sums in parallel.

To compute the ratio of data sent in memory to data accessed from local memory, notice that each element in the result matrix is computed from the product of two vectors of length 40. Consequently, 80 elements are fetched for each one stored. The total number of accesses is

$$(80 + 1) \times 40 \times 40 = 129{,}600.$$

The total number of message bytes, as computed in Table A-4, is 19,968, independent of the number of slaves. Thus data sent in messages is 15 percent of the number of array accesses. In the AMPL implementation of matrix multiplication, a few more local accesses are performed because vectors are not multiplied all at once, and many more accesses are made during subscript calculation.

A.14. Simulation of Molecular Motion

Algorithm names: Metropolis, molecular dynamics.
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: MEDUSA.
Other software in environment: None.
Experimenters: Neil Ostlund, Peter Hibbard, and Bob Whiteside, 1980–84.
References: [Jones and Gehringer 80, Ostlund *et al.* 82a, Ostlund *et al.* 82b].

Given the microscopic interactions between particles, we want to predict the static and dynamic properties of a collection of such particles. Macroscopic quantities are obtained by an averaging according to one of two methods—ensemble averaging or time averaging.

The Metropolis method [Metropolis *et al.* 53] employs ensemble averaging. A Metropolis "Monte Carlo" calculation is made up of a large number of passes, during which the average value of some property is computed. (We will ignore the computation of the property and concentrate on the process of ensemble averaging.) During each pass, an attempt is made to move each particle in the collection. When a particle is moved, its coordinates are randomly perturbed, producing a new configuration. The new configuration is then either "accepted" or "rejected," with a probability based on the change in energy of the system caused by the move. If the new configuration is accepted, then the new coordinates are used during each subsequent attempt to move a particle; otherwise, the old coordinates are used.

The bottleneck of the calculation for each move is the computation of the binding energy for each particle, which involves $O(N)$ calculations. The parallel algorithm uses K processors in an attempt to reduce the complexity of this step to $O(N/K)$.

Table A-4 Number of Words Transmitted in Messages during AMPL Matrix Multiplication

One message from a matrix process to a slave, or a slave to the result process, contains		
a 10-by-10 submatrix		100 words
For each submatrix multiplication, the following information is sent:		
• three messages, two matrix → slave and one slave → result		300 words
• a slave-port reference (**refport**) to matrix process *A*		2
• a slave-port reference (**refport**) to matrix process *B*		2
• a four-word command to the slave		
a **refport** telling where to send results	2	
two 16-bit integers giving row and column	2	
Subtotal		4
• a slave-port reference back to the master when the slave is done		2
• row and column numbers to result process		2
Subtotal		312
There are 4 × 4 × 4 = 64 submatrix multiplications altogether: *Total*		64 x 312 = 19,968 words

The interactions can be evaluated in parallel without interprocessor communication, but the contributions calculated by each processor must be added together. The complexity of this step is $O(N)$. In addition, the computation must be synchronized at each move.

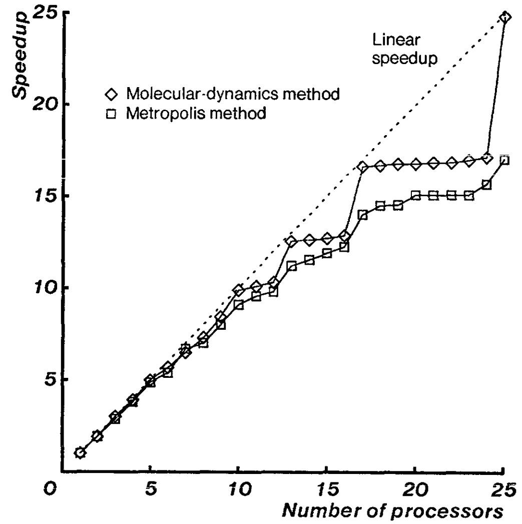
The molecular-dynamics algorithm [Verlet 67] uses time averaging. First, an initial set of velocities for the particles is calculated. Given an initial set of coordinates, the velocities can be used to predict a set of coordinates at a later time. A summation is again performed to find the binding energy, but with this algorithm, the summation can be reordered to allow a processor to sum its subset of binding energies *locally*, with the global summation being required only once at the end of the computation.

These simulations are described by Ostlund, Hibbard, and Whiteside [Ostlund *et al.* 82a]. This problem is representative of the general problems involved in the theoretical study of molecular motion. The results shown in Figure A-39 are for a system of 50 particles.

The molecular-dynamics algorithm shows better speedup than the Metropolis algorithm. One factor is that the molecular-dynamics algorithm avoids performing sums of shared variables. In addition, since particles move simultaneously rather than one at a time, the $O(N^2)$ serial computation of the binding energy is converted to an $O(N^2/K)$ parallel computation using K processors. A processor computes $O(N^2/K)$ interactions between synchronizations, instead of $O(N/K)$ interactions as in the Metropolis algorithm.

Both graphs exhibit a zigzag pattern because the particles are parceled out among the processors as evenly as possible (there is still only one process per

Figure A-39 Speedup of Metropolis vs. Molecular Dynamics Algorithm

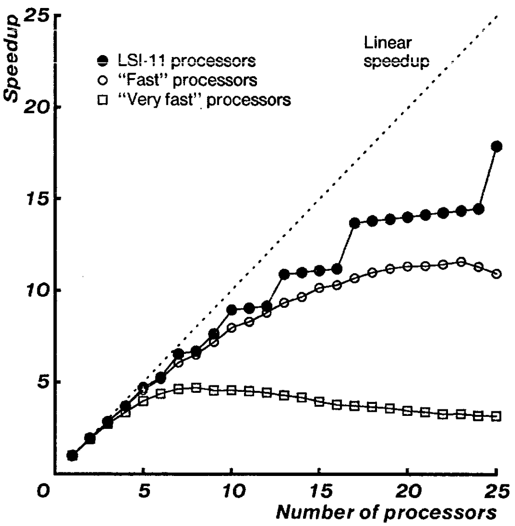


processor, however). Each time there is a decrease in the number of particles handled by the "busiest" processor, speedup increases markedly. Note, for example, how speedup "jumps" between 24 to 25 processors; in the latter case, no processor need handle more than two particles. The 25-processor system is an example of matching parallelism, as described on page 246.

Both these algorithms for molecular motion are synchronous, requiring lockstep iteration; in contrast to the PDE, no asynchronous molecular dynamics equation is known. Evidently, the synchronization penalty would be large, especially for the Metropolis algorithm. Relatively little computation is performed between synchronization points. Also, before each synchronization point, each process requires exclusive access to the global summation variable, hence the processes must contend for its lock. Lastly, there is a serial step after each attempt to move a particle, as the master decides whether to accept the move and perturbs the coordinates of another particle to generate the next configuration.

In practice, these factors seem to cause little degradation in comparison with other parallel algorithms, such as those described in Sections A.2, A.5, A.7, and A.11. These algorithms exhibit declining speedup after some critical point, while the Metropolis algorithm continues to gain speed as more processors are added. More amazingly, the molecular-dynamics algorithm, which suffers the same synchronization and contention penalties, albeit somewhat less often, shows more nearly linear speedup than any other algorithm implemented on Cm* (except the "lucky" run of the integer-programming problem; see Section A.3). How can this be?

Figure A-40 Effect of Processor Speed on Speedup of Metropolis Algorithm



The explanation is actually quite simple. The binding-energy calculations are performed in double-precision floating-point arithmetic, which on the LSI-11 is implemented by software procedures. The tremendous overhead of software simulation tends to mask the overhead of synchronization and the serial phase. Firmware or hardware floating-point arithmetic would speed up these operations by one or two orders of magnitude. It is not very difficult to simulate faster floating-point operations. Because we are interested in the performance, rather than the results, of the Metropolis algorithm, the binding-energy computation can simply be replaced by a delay loop. By changing the length of the delay loop, then, it is possible to simulate faster or slower floating-point arithmetic.

Figure A-40 illustrates the impact of simulating faster floating-point operations.[4] The "fast" curve reports the results of experiments in which simulated floating-point arithmetic was speeded up by a factor of 10. Here the critical point (maximum speedup) occurs at 23 processors. Using "very fast" simulated arithmetic—a factor of 100 faster than the LSI-11—the critical point is reached at just 8 processors. All results subsequently presented in this section have been derived using "fast" processors.

Since synchronization overhead becomes more conspicuous with increasing processor speed, it is desirable to reformulate the algorithm to relax the synchronization requirements. Recall that during each pass, an attempt is made to move each particle in the collection. Lockstep iteration occurs because no processor is allowed

[4] A different version of code was used in this experiment from that used for Figure A-39; hence the two curves are slightly different.

to begin working on the $(i+1)$st move until all processors have completed work on the ith move. One way of visualizing this approach is to suppose that the coordinates of the trial move have been deposited in a single *buffer*, that all processors read from this buffer until they are finished, and that the master then decides whether to accept or reject the trial move.

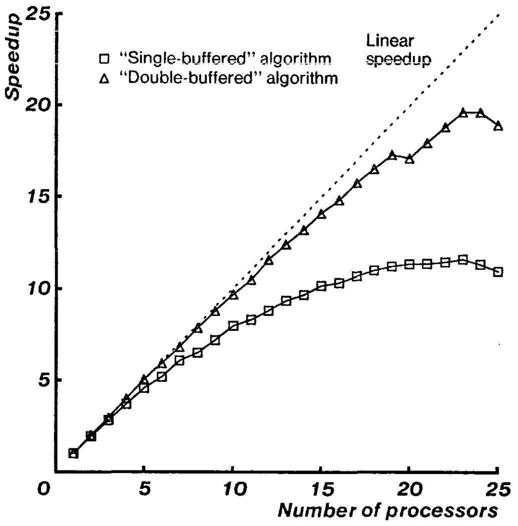
Suppose, instead, that we modify the algorithm to use two buffers. During the attempt to move the ith particle, the first of these buffers, called the *current* buffer, contains the trial coordinates of the ith move, just as in the previous version of the algorithm. The *other* buffer holds the trial coordinates for the $(i+1)$st move, except that the position of the ith particle is temporarily marked "unknown." When a processor finishes computing the binding energy for its particle, it switches to the other buffer and starts computing the binding energy for a particle in the $(i+1)$st trial configuration, subject only to the restriction that it must skip the contribution of the ith particle for a while because its coordinates are not known until all processors have finished working on the ith trial configuration.

Some processors happen to be slightly faster than others, but eventually all processors finish computing their binding energy for the ith trial configuration and leave the current buffer. When this happens, the last processor "turns out the lights" by (a) making the accept/reject decision for the ith trial move and updating the coordinates appropriately, (b) generating the coordinates for the $(i+2)$nd trial move, and (c) making the *current* buffer the *other* buffer, and vice versa. Those processors that were waiting to learn the coordinates of the ith particle can now proceed to complete their calculations for the $(i+1)$st configuration and move on to the $(i+2)$nd configuration.

Notice how the synchronization requirements have been relaxed: The "fastest" processor, when it is nearly finished with the $(i+1)$st configuration, can be nearly two configurations ahead of the "slowest" processor, which may be just starting on the ith configuration. Before actually finishing work on the $(i+1)$st configuration, however, the fastest processor must wait for the slowest processor to complete the ith configuration. Figure A-41 shows that the double-buffered algorithm has almost linear speedup out to about 15 processors and significantly outperforms its single-buffered counterpart thereafter. Several factors contribute to its improved performance. First, the lack of a global synchronization point decreases contention for the lock on the global summation variable. Contention is further reduced because the processors get "skewed" the first time they contend for the lock and remain skewed instead of being "realigned" at the global synchronization point. This decreases the probability that two processors will subsequently request the lock at the same time. Finally, there is no serial phase to add the binding-energy contributions. Rather, as the last processor is performing this summation at the end of the ith configuration, the other processors can be proceeding to work on the $(i+1)$st configuration.

Systolic Algorithms. Experience with the Metropolis algorithm seems to suggest the following question: If two buffers are better than one, then are n buffers better than two (where n is the number of particles in the system)? Perhaps, if other modifications are made. Recall that in the computation of the $(i+1)$st trial move, the

Figure A-41 Speedup of Single- and Double-Buffered Algorithm

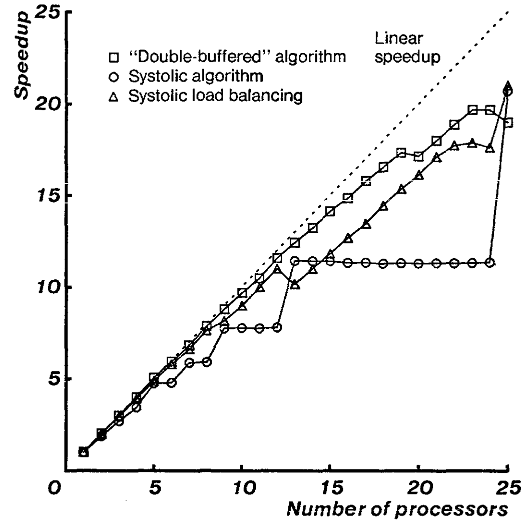


contribution of the *i*th particle had to be skipped for a while until its coordinates became known. In the hypothetical *n*-buffer algorithm, in the computation of the *n*th trial move, the contribution of the *n* – 1 other particles would have to be "skipped" until their coordinates became known.

This algorithm will not be explored in detail. Instead, we will give an inexact, but intuitive, explanation of how this idea can be converted into a working algorithm. With the *n*-buffer algorithm, the processors working in the first buffer may complete their work without knowing the final decision on any of the new coordinates. Processors working in the second buffer must know the new coordinates of the first particle; processors in the third buffer must know the new coordinates of the first two particles, and so forth. Imagine that we had *n* processors and could arrange to have one of them working in each buffer. They could receive the new coordinates of particles one by one, as they become available. Intuitively, this seems to suggest an algorithm where the coordinates of a particle either are broadcast or move down a pipeline. Such an algorithm is known as a *systolic* algorithm, because data "pulses" from each processor to one of its neighbors at the end of each step in the computation. In the Cm* implementation, the coordinates are passed from one processor to another by means of MEDUSA messages.

The systolic Metropolis algorithm was implemented by Whiteside [Whiteside *et al.* 82, Whiteside *et al.* 83]. Its speedup for a system of 50 is shown in Figure A-42. Notice that it, like the molecular-dynamics algorithm (see Figure A-39), shows prominent steps at each point of matching parallelism. The reason is the same: When there are fewer processors than particles, some of the processors do com-

Figure A-42 Systolic vs. Nonsystolic Metropolis Algorithm



putations for one more particle than other processors. The execution time of the entire experiment is bounded by the speed of the "busier" processors. The systolic algorithm can be modified to diminish this effect. To balance the load among processors, at the end of each computation step, the busier processors "send" one of their particles to a neighboring processor. Like the communication of coordinates, this can be accomplished by message passing. Notice that the systolic load-balancing algorithm exhibits none of the steps of the original systolic algorithm. It does achieve generally lower speedup, however, perhaps due to the overhead of MEDUSA message operations relative to the shared-memory approach taken by the double-buffered algorithm.

A.15. Comparative Implementations of Ada Rendezvous

Algorithm name: Ada rendezvous.
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: MEDUSA.
Other software in environment: None.
Experimenters: Anita Jones and Anders Ardö, 1981–82.
Reference: [Jones and Ardö 82].

In principle, a *rendezvous*, as defined in Ada, is just a special case of message passing. The message-passing systems of STAROS and MEDUSA have the ability to *buffer* messages; a message can be sent by one process and later received by another process, with the sending process performing other work in the meantime. If the mailbox or pipe is full, however, the ***Send*** cannot be completed until space is freed for the new message. Ada rendezvous is similar, except that the buffer (mailbox or pipe) has a size of zero; there is nowhere for messages to wait before being received. Consequently, the ***Send*** cannot be completed until the instant that the ***Receive*** is performed. Of course, the ***Receive*** cannot complete until the ***Send*** is performed, so the sending and receiving processes must execute their message-communication statements simultaneously. This is the origin of the term "rendezvous."

Rendezvous can be implemented in several ways. The most straightforward method is via message passing, but because a rendezvous can communicate information in both directions, a single rendezvous may require two messages. In special cases, a rendezvous also can be implemented using shared memory. To investigate these alternatives, we will consider an example.

The example is a task force organized according to the "server" paradigm. A set of identical *worker* processes are responsible for performing variable-length work requests. We will refer to the work requests as "tasks," corresponding to the usage of the term on page 244. Each time a process finishes performing a task, it begins work on a new task, until there are no more tasks to do. The number of tasks, and the length of each task, may be dynamically generated or known *a priori*. If the tasks are dynamically generated, there must be some way of telling a worker which task it is to perform next. This is known as an *assignment* of work.

In Ada, the obvious implementation is to use a *master* process to assign tasks to the workers. If there are *n* tasks altogether, the code for the master is that shown in Figure A-43. A worker executes the code shown in Figure A-44.

Figure A-43

Master Process Code

```
for WorkID in 1 . . n loop
        accept Assign (Id : out integer ) do
            delay Ta ;                          -- simulate an assignment cost of Ta
            Id := WorkId ;
        end Assign ;
end loop;
for i in 1 . . NumWorkers loop
        accept Assign (Id : out integer ) do
            Id := NoWork ;
        end Assign ;
end loop;
```

Figure A-44

Worker Process Code

```
loop
        Master.Assign (MyWorkId );
        exit when MyWorkId = NoWork ;
           delay T (MyWorkId );                -- simulate processing cost
end loop;
```

Jones and Ardö studied the performance of four implementations of the rendezvous in this program. In all cases, each processor executed only a single process.

Message-passing implementation. A worker sends its process number to the master when it is able to take on a new task, and the master responds by sending the worker a message containing the task number. When there are no more new tasks to start, the master instead sends a message containing a special task number that means "no more work." The messages are conveyed through MEDUSA pipes. Since there is only one process per processor, the ***Receive*** operations can specify a very large pause time, so context-swap overhead is never incurred. This implementation strategy uses no shared memory and hence could be applied on a computer network as well as a multiprocessor.

Busy-waiting on shared variables. A worker waits on a shared variable until it is assigned work by the master process. The master polls the shared variables until it finds a worker that needs more work. It then assigns a task to the worker and communicates the identity of the newly assigned task via another shared variable. Thus there are two arrays of shared variables. One array records, for each worker, whether that worker is waiting for work. The other array records, for each worker, the task number that the worker has been assigned. Since only the master can write into the first array and only one worker writes into any element of the second array, no locks are needed on either array. If there were more than one process per processor, this implementation would be very inefficient because a busy-waiting process often would prevent another process from executing.

In-line embedded code. Instead of having a master process assign tasks to the workers, the workers can do the assignment themselves by procedure calls. The number of the last task to be assigned is held in a global variable, which is protected by a lock. A worker that needs a new task calls a procedure that locks this variable, then reads and increments it. The MEDUSA ***Indivisible Increment*** instruction can be used for this purpose. Although there is no master, one worker is distinguished in that it initializes the counter and starts the other workers by unlocking the global counter variable. This strategy correctly implements the semantics of Ada rendezvous in the example program, even though it might be difficult for a compiler to notice the optimization.

Static assignment. If the lengths of all tasks are known before execution begins, the tasks can be divided among the workers all at once—in other words, statically. Each worker merely executes its set of tasks, without communicating with a master or any other worker; synchronization is needed only to start and stop the task force. Clearly, this is the most efficient implementation when it is possible.

Results. When a task is assigned to a worker, the worker may have to obtain access to a set of data that defines it. We will call this the "assignment cost" and distinguish it from the cost of implementing the rendezvous itself. The performance of the four strategies depends on the assignment cost and the lengths of the tasks. Notice also that the in-line and static assignment strategies, which lack a master process, have a built-in advantage over message passing and busy-waiting: With *n* processors, they can accommodate *n* workers instead of *n* – 1.

The first experiment assumed that the assignment cost and the task lengths were both zero. It therefore measured only communication and synchronization costs. The results are given in Figure A-45. The message-passing implementation is the slowest of the four for small numbers of processors. Its execution time is dominated by the approximately 1.2 ms. that are needed to perform the four pipe operations that convey the two messages. Some overlap obviously occurs because the cumulative time is less than the sum of the times for the four pipe operations (Section 7.4.2). From the fact that performance ˥egrades very slowly as processes are added, we can deduce that Kmap contention is not a serious factor.

The in-line code approach suffers contention overhead, as indicated by its worsening performance as more processors are added. Several kinds of contention are responsible—contention for the memory word that is used as a lock and contention for Kmap contexts, for example. As the lock operations go cross-cluster at about 12 processors, the slope of the curve increases, reflecting the higher intercluster access times. In contrast, the busy-waiting solution does not experience contention; its performance actually improves with greater numbers of processes, up to about four or five, as loop control becomes less significant with respect to the body of the loop

Figure A-45 Synchronization and Communication Costs in Various Rendezvous Implementations

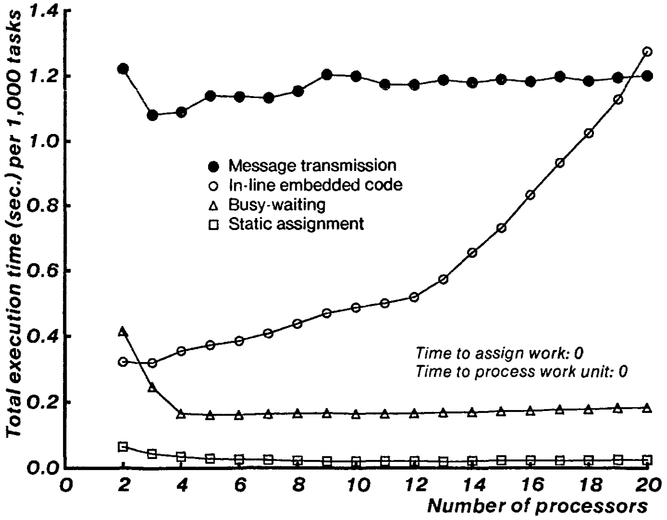
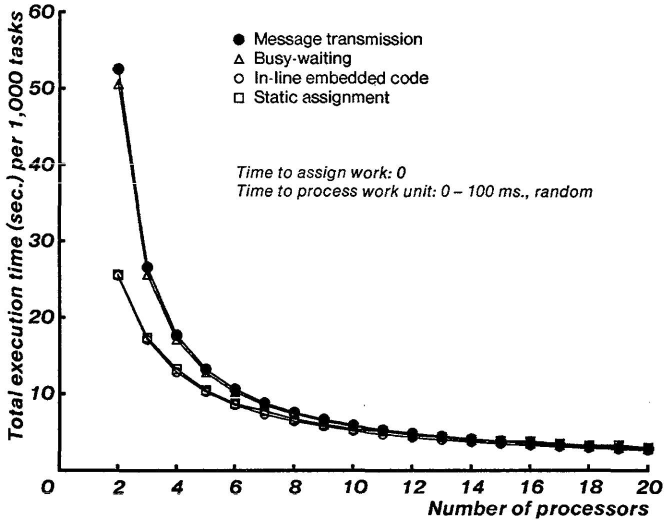


Figure A-46 Performance of Rendezvous Implementations when Task-Processing Dominates



that polls the worker processes. Static allocation is the cheapest strategy, as expected. After about seven processors are reached, it is constantly cheaper than the busy-waiting approach by a factor of three.

Figure A-46 shows what happens when assignment cost is negligible (0.1 ms.) and processing cost (randomly varying between 0 and 100 ms.) dominates. There is little difference between the four implementations, except for the built-in advantage of the in-line and static-assignment strategies. This advantage is most pronounced when there are few processes. With two worker processes, for example, each has only half as much work as a single worker would have. By the time the number of workers reaches 20, the advantage nearly vanishes.

From this experiment, some general guidelines can be derived. First, the rendezvous implementation strategy has the greatest effect when rendezvous occurs frequently. If a task executes for, say, twenty times as long as it takes for a rendezvous to occur (an average 50 ms. versus 2.4 ms.), the synchronization strategy is of little import. Second, runaway contention is the most serious problem faced by a synchronization mechanism. Even a method that nominally requires much more execution time will perform better on large task forces if it is not subject to contention. Finally, when the number of processors is small, it is wasteful to dedicate one of them to a master process that performs no tasks. If the number of processors is larger—15 or more in this example—the presence of a master has little influence on performance.

A.16. Transaction-Processing System

Algorithm name : Transaction-processing system.
Cm configuration* : 50 Cm's, 5 clusters (only 3 clusters used).
Operating system : MEDUSA.
Other software in environment : None.
Experimenter : John Robinson, 1982.
Reference : [Robinson 82].

A transaction-processing system is a database system that, in principle, allows any of the users sharing the database to modify it. Examples include airline reservations systems, point-of-sale inventory-management systems, and program-documentation systems that update a database whenever a change is made to a module of the program. Among the key problems in designing such systems are how to distribute and cache data in the various levels of the memory hierarchy and how to synchronize accesses to the data to ensure that queries access a consistent copy of the data and that updates are indivisible.

Robinson implemented a transaction-processing system on Cm* under the MEDUSA operating system. He utilized two data-placement strategies and two main synchronization policies. The data-placement strategies were as follows:

- MEDUSA random-access files were used for the shared memory of the database.
- The database was placed in four otherwise unused Cm's, each with 128K bytes of memory, and accessed through the shared descriptor list. The goal was to simulate a database placed on fast disks rather than in primary memory. Reads and writes of 512-byte pages were performed using Kmap ***Block Move*** operations. The maximum block-move rate is approximately 300 blocks per second.

The synchronization, or *concurrency control* policy, must ensure that conflicting accesses to the database (for example, two concurrent *writes* of the same information or a *write* during a *read* of overlapping data) cannot take place. Many concurrency control policies are possible. Robinson considers two in detail: the *locking* policy and the *optimistic* policy.

- The locking policy stipulates that an arriving *read* or *write* request will wait for all conflicting transactions, and an arriving *read / write* request will wait until the completion of all transactions that are in progress.
- The optimistic policy never requires a transaction to wait; conflicting transactions "race" to the finish. The transaction that "wins" is allowed to finish, while all other conflicting transactions are aborted just before the point of conflict. For example, two concurrent write transactions are allowed to proceed until the second attempts to write data that has been written by the first; at this point, the second transaction is aborted.

The transaction-processing system consisted of up to 11 processes (MEDUSA activities): a master process, up to 8 *transaction processors*, a *concurrency control* process, and a *global memory manager*. Each process was allocated its own Cm, and the code, stack, and private data was always local to the Cm.

In a real transaction-processing system, transactions would be obtained from outside the system, usually from some sort of user interface. In this simulation, however, each transaction processor randomly generated the transactions it processed. These transactions consisted of insertions, deletions, and queries, which accessed random tuples in the database.

Results. Figure A-47 shows that as processors were added, the MEDUSA *File System* soon became a bottleneck. One process, the MEDUSA *File System* activity, was responsible for managing the disk. Only one disk access could be made at a time, so as soon as the *File System* process saturated, no further increase in throughput was possible. In fact, throughput decreased, probably because

- with the locking policy, processors waiting for I/O had locks on a considerable portion of the database, increasing the likelihood that a subsequent transaction would have to wait.
- with the optimistic policy, more transactions were outstanding, increasing the frequency of conflicts between transactions and hence the frequency with which transactions had to be aborted and restarted from the beginning.

Saturation would have occurred with even fewer processors, if the transaction

Figure A-47 Throughput of Transaction-Processing System

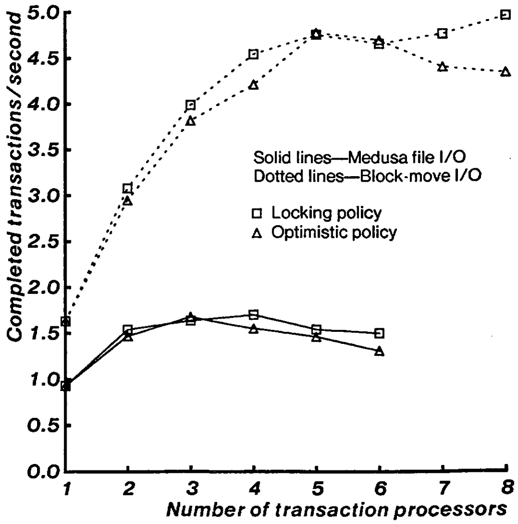
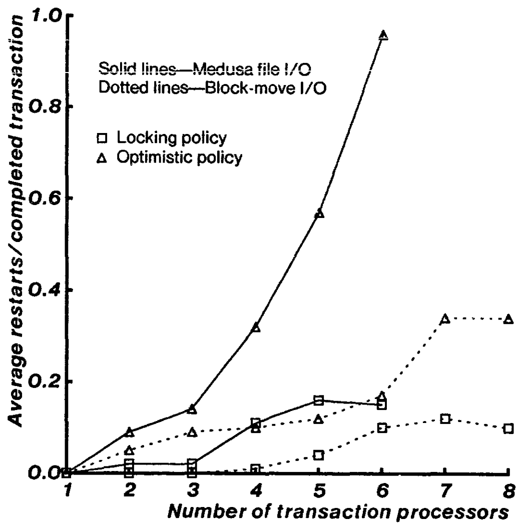


Figure A-48 Number of Aborted and Restarted Transactions



processors had not cached pages in local memory. Figure A-48 provides evidence for the high rate of conflicts under the optimistic policy. Note that even under the locking policy, transactions must be aborted occasionally to prevent deadlock.

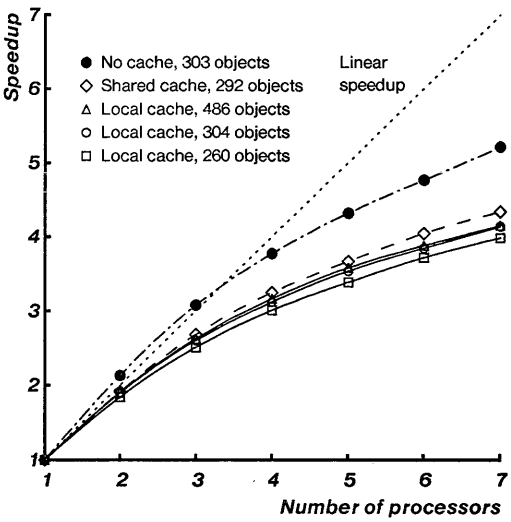
Contention for the file system is a serious problem because there is only one disk and one MEDUSA *File System* activity. When I/O is performed with Kmap ***Block Moves***, contention would be for the Kmap and for shared memory instead. Seven Kmap contexts can be allocated to ***Block Moves***, however, and there are 384 pages in the simulated database, so contention for these resources is much less likely.

Examining the other two curves of Figure A-47, it is evident that ***Block Move*** I/O yields much higher throughput and also that more transaction processors can be utilized effectively. As more processors are added, the increase in throughput slows, due to conflicts between transactions. Eventually, the increasing number of restarts dominates (Figure A-48) with the optimistic policy, and throughput decreases. With the locking policy, throughput continues to increase with up to eight transaction processors.

A.17. Parallel Garbage Collector

Algorithm name : STAROS *Garbage Collector*.
Cm configuration* : 50 Cm's, 5 clusters.
Operating system : STAROS.
Other software in environment : None.
Experimenter : Robert J. Chansler, Jr., 1982.
Reference : [Chansler 82].

Figure A-49 Speedup of the StarOS Garbage Collector



As described in Section 6.6, StarOS uses garbage collection to reclaim the memory occupied by inaccessible objects. The StarOS *Garbage Collector* runs in parallel with other activity in the system and, in fact, may itself be composed of several processes that run in parallel with each other. Like other parallel garbage collectors, the StarOS *Garbage Collector* must treat as accessible all objects created while garbage collection is in progress. Consequently, the longer it takes to collect garbage, the more overall work there is to perform. Conversely, as the *Garbage Collector* speeds up (for example, by employing multiple garbage-collection processes), the overall work is less. The *Garbage Collector*, then, has the unusual property that its theoretical speedup is greater than linear. Section 11.2 discusses this aspect in more detail.

Chansler measured the speedup of the *Garbage Collector* in several different experiments. The experiments are classified according to the type of cache used for the information on which objects have been marked. A field in the descriptor for an object, called the *color* field, tells whether it has been marked. As explained in Section 6.1.1, a StarOS process must read the descriptor for an object in order to access it. When the *Garbage Collector* needs to check the color of the object, it must read the descriptor again, unless it maintains a cache of object colors. If it chooses to maintain a cache, the cache may be either local to the garbage-collection process or shared among all garbage-collection processes.

Figure A-49 (which reproduces Figure 11-4) shows the results of one experiment each without a cache and with a shared cache and three experiments with a local cache. (The three experiments were made with three different StarOS workloads.) Notice that only the no-cache experiment ever shows greater than linear speedup, and then only with two or three processors. The absolute execution time of the *Garbage Collector* with this implementation, however, is about 25 percent longer than in any other experiment. With a cache, the *Garbage Collector* makes fewer overall references to shared data, but because a greater fraction of its references are to shared data, contention during these references is greater. The no-cache implementation has less contention and higher speedup. Like the "slow-processor" Metropolis algorithm, its inefficiency makes it more susceptible to speedup.

A.18. Traveling-Salesman Problem

Algorithm name: Traveling-salesman problem.
Cm configuration*: 50 Cm's, 5 clusters (only 4 clusters used).
Operating system: StarOS.
Other software in environment: None.
Experimenter: Joe Mohan, 1982.
Reference: [Mohan 83].

The traveling-salesman problem (TSP) is to find the minimum-cost "tour" that visits each element of a set of "cities," given a cost matrix that specifies the cost of moving from each city to any other city in the set. More precisely, given a complete graph $G=(V,E)$, where V is a set of n vertices and E is the set of m edges where $m=n(n-1)$, if c_{ij} is the cost associated with edge (i, j), $[c_{ij}]$ will be the cost matrix of the graph G. TSP is the problem of finding a Hamiltonian circuit of G with the minimum cost.

TSP has obvious applications to vehicle routing, and it underlies many other optimization problems such as sequencing and scheduling. While it is easy to state, it is hard to solve—in fact, it is NP-complete. When an exact solution is desired, some form of branch-and-bound algorithm usually is used. Typically, such algorithms perform much better than the worst case if good bounding heuristics are employed.

Mohan experimented with an algorithm based on the work of Little, Murty, Sweeny, and Karel [Little *et al.* 63]—called the LMSK algorithm for short. Following Mohan [Mohan 83], we describe the algorithm as follows:

The LMSK algorithm works by partitioning the set of all possible tours into progressively smaller subsets (which are represented on the nodes of a state-space tree), computing a lower bound on the cost of the best tour in each partition, and then expanding the state-space tree incrementally toward the goal node, using heuristics to guide its search toward the solution node. The *node-selection heuristic* chooses, from among all the leaf nodes of the current tree, that leaf node whose estimated lower-bound tour cost is the least. The *edge-selection heuristic* computes the increments in tour cost when different edges are excluded from the tour and chooses the edge that causes the maximum increment. The description of the algorithm below explains how the nodes and edges thus chosen are used.

Starting with a tree consisting of just the root node, which represents the set of all tours, the algorithm *repeatedly* executes these steps *in sequence*: It first chooses, from all the leaf nodes of the current tree, the node that is most likely to lead to the solution (using the node-selection heuristic) and designates it to be the next expansion node; it then chooses one or more edges (legs of a tour) using the edge-selection heuristic; it sets up the child nodes with all the different combinations of inclusion and exclusion of the selected edges; and finally, it computes for each child node the lower-bound cost of all tours in the subset defined by the child node. For any given node, each of its child nodes corresponds to that subset of the tours of the parent node, with the additional constraint of including or excluding one or more edges. The algorithm terminates when a leaf node is found that represents a single complete tour with a cost less than or equal to the lower-bound costs of all possible tours, which are represented by the leaf nodes of the current state-space tree.

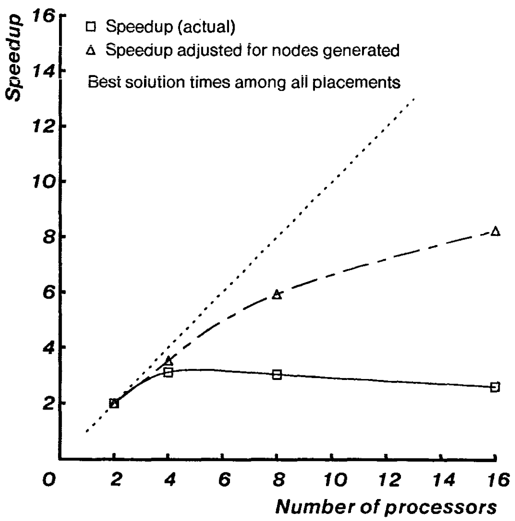
Here is a high-level representation of the algorithm:

```
repeat

   1. Select node in state-space tree with least lower-bound tour cost.
   2. Select one or more edges using edge-selection heuristic.
   3. For each child corresponding to one inclusion/exclusion combination of the
      selected edges,

         3.1 . Create a node object and link it to tree.
         3.2 . Derive cost matrix for child node.
         3.3 . Reduce matrix and find new lower bound for all tours defined by
               child node until a full tour, with cost less than or equal to the lower
               bounds on all possible tours, is obtained.
```

Mohan experimented with two different parallel decompositions of this algorithm. One implementation, known as TSP1, unfolded the *for* loop of step 3, parceling out different iterations to different processors. The other implementation, known as TSP2, unfolded the outer *repeat* loop. TSP1 can be characterized as a multiphase algorithm (see Section 11.4.1). The master process performs steps 1 and 2, then activates the slaves, one for each child node, which work concurrently on step 3. TSP2 is structured as an asynchronous collection of processes. Each process, during each iteration, selects one edge and sets up two child nodes—one of which includes the edge in the tour and one of which excludes it. The processes communicate via the state-space tree and global variables, synchronizing their accesses with spin locks (mutual-exclusion locks). In TSP1, processors are idle during the master phase and while the faster slave processes wait for the slower ones to finish. By contrast, TSP2 processors are idle only during startup transients (when there are more processes than nodes) and while waiting on one of the spin locks.

The number of slave processes in TSP1 is always a power of two, specifically $n = 2^k$, where k is the number of edges selected in step 2. In the experiments, n ranged from 2 to 16. Speedup was computed by comparing execution times to *twice* the execution time observed for $n = 2$. Each TSP2 process selected one edge; 1 to 16 processes were used, and speedup was computed in the normal fashion.

Figure A-50 Speedup of TSP1



Results. EFFECTIVENESS OF HEURISTICS. Figure A-50 plots the speedup of TSP1. Two curves are shown. The lower curve is the observed speedup; note that it declines as *n* increases above four. The dashed curve is an attempt to separate synchronization overhead (page 242) from parallelization overhead (page 248). As the number of processes grows, more computation is done between successive applications of the heuristics. The heuristics become less effective, so more nodes are generated and more total work done. We can say that the heuristic *granularity* has increased [Mohan *et al.* 85]. If the execution time is scaled by the number of nodes generated, using as a base the number of nodes generated for a parallelism of two, the dashed curve is obtained. More detail may be found in Section 11.5.3.

INTRACLUSTER AND SYSTEMWIDE CONTENTION. Since each TSP2 process selects only one edge, regardless of the number of processes in the system, the heuristic granularity remains constant, and the speedup continues to rise as more processors are added. Nonetheless, the speedup becomes more and more sublinear as the number of processors grows (Figure A-51). Perhaps the convexity of the speedup curve can be explained by contention.

TSP2 experiences contention for several resources. Some of these resources are replicated in each cluster (the Kmaps, the map buses, and the *Object Manager*); this is called *cluster-level contention*. If there is only a single instance of a resource for which a process contends, that process is said to experience *system-level contention*. System-unique resources include global data (for example, the state-space tree), spin locks, critical sections of procedures, and the intercluster bus. Mohan ran a set of experiments to estimate the effects of cluster-level contention by varying the number of processes but keeping the number of processes per cluster

Figure A-51 Speedup of TSP2

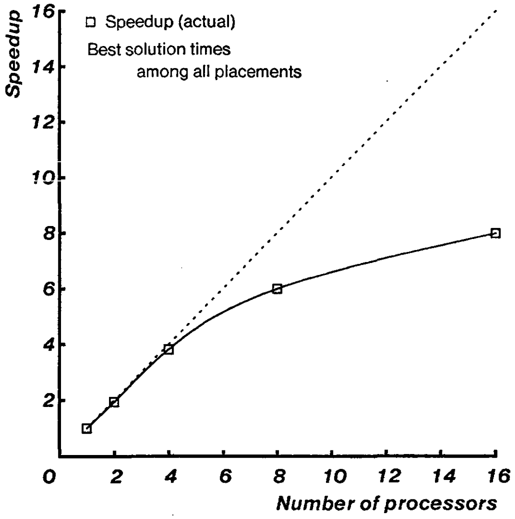


Figure A-52 Speedup of TSP2, Normalized for Cluster-Level Contention

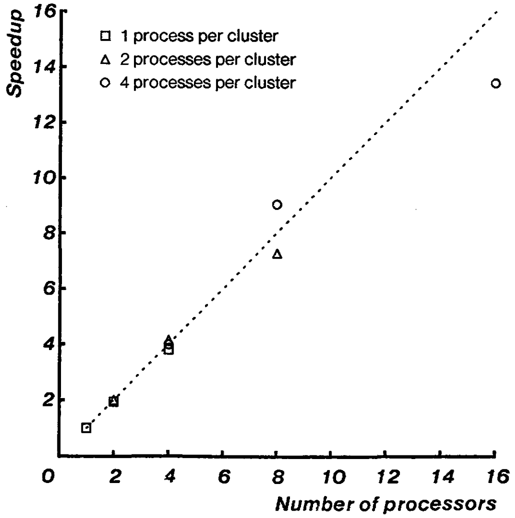
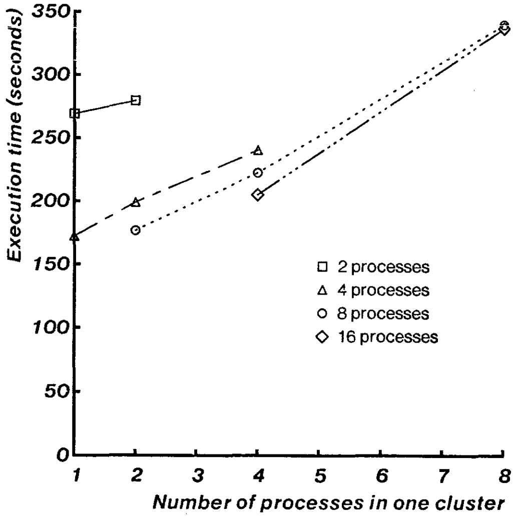


Figure A-53 Execution Time vs. Placement for TSP1



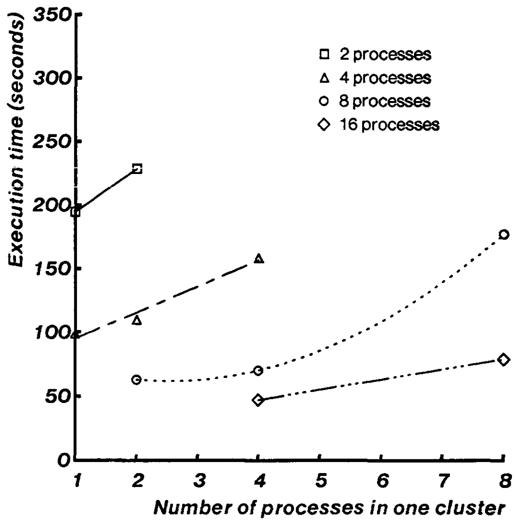
constant. Any degradation in speedup, then, should be primarily due to system-level contention.

The results, shown in Figure A-52,[5] indicate that speedup is nearly linear until parallelism reaches 16. This suggests that cluster-level contention is the primary source of degradation at a parallelism of 8 and below.

PLACEMENT OF PROCESSES. Like Deminet (see Section A.7), Mohan studied the distribution of processes among clusters. Unique among all the experimenters on Cm*, he studied the effect of distributing the *same number of processes* among *different numbers of clusters*. It is not clear *a priori* what the optimal number of processes per cluster is because two conflicting factors are at work. As the number of processes per cluster increases, the average reference time decreases as expensive intercluster references become less frequent, and cluster-level contention grows and leads to higher execution times. For TSP1 (Figure A-53), the best performance is obtained when processes are distributed among several clusters. In fact, execution time seems to increase linearly with the number of processors per cluster. TSP2 (Figure A-54) also seems to perform best with widely distributed processes. When both processes of a two-process task force are moved into a single cluster, there is a greater absolute increase in TSP2 execution time than in TSP1 execution time; this probably reflects the greater parallelism in TSP2, due to the lack of a serial

[5] Many of the speedup values in this figure appear higher than those in Figure A-51. This is because each one-cluster, *m*-processor-per-cluster configuration is arbitrarily assigned a speedup of *m*. (This is analogous to the way speedup is computed for TSP1, with a base case of two processors.) Computing other speedups with respect to this value yields different results from computing them in the normal fashion.

Figure A-54 Execution Time vs. Placement for TSP2



phase. The increase in execution times is not so dramatic for larger task forces, especially for the 16-process configuration. Nonetheless, in every case, the best-performing configuration is the one that uses the most clusters.

Although the experiments are not directly comparable, it is interesting to contrast this observation with Deminet's discovery of the crossover phenomenon (see Section A.7), where an increase in intercluster references tends to outweigh the beneficial effects of adding several additional processors. It may be that the PDE makes heavier use of intercluster references, or Deminet might have observed different results if he had always distributed his processes *evenly* among clusters. Future research will be required to shed more light on the effects of process placement.

EXECUTION TIME, SYNCHRONISM, AND THE SERIAL PHASE. A glance at Figures A-53 and A-54 reveals that TSP2 outperformed TSP1 for all numbers and configurations of processes. Several factors account for the difference. TSP1 encounters significantly more synchronization overhead because the slave processes work in a lockstep fashion under the control of the master process. TSP1 executes steps 1 and 2 serially, whereas TSP2 executes them in parallel. Finally, the synchronous nature of TSP1 leads its processes to request global resources at approximately the same time; this clustering increases contention and execution time.

TSP is a valuable case study because it demonstrates that an important class of search algorithms can be effectively parallelized. At first glance, search algorithms seem not to lend themselves to parallel processing because they make extensive use of global data without any predictable access patterns. Furthermore, search heuristics may be applied less frequently, and hence become less effective. TSP2 has overcome both these difficulties by a judicious choice of decomposition strategies.

A.19. Design-Rule Checking on MEDUSA

Algorithm name: Design-rule checking.
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: MEDUSA.
Other software in environment: None.
Experimenter: Francesco Gregoretti, 1983–84.
Reference: [Gregoretti and Segall 84].

A VLSI (very large scale integration) circuit design must conform to certain rules that specify the minimum allowable distance between adjacent circuit elements. Most circuit elements are represented as rectangles whose sides are parallel to the coordinate axes of the chip. The problem of deciding whether a pair of rectangles are "too close" can be solved by determining whether a pair of slightly larger rectangles overlap. The complexity of modern VLSI circuits demands computer assistance in this task. Indeed, the process is frequent and expensive enough to warrant multiprocessing as a means of reducing overall design time.

It is possible to define a circuit by listing all the rectangles that comprise it, but it is much more efficient to describe a large circuit in a hierarchical fashion. A memory chip, for instance, can be viewed at ascending levels of abstraction as a collection of rectangles, transistors, memory cells, and memory-cell rows. Caltech Intermediate Form, or CIF, is a layout language that describes a design as a set of symbols. Each symbol is made up of rectangles and instances of other symbols whose positions are given relative to the local origin of the symbol. To prevent infinite recursion, a symbol must be defined before it can be used.

During design-rule checking, errors are reported if two rectangles lie within the minimum permissible distance. Various component counts are also maintained, but the bulk of the work consists of finding overlapping elements.

A typical VLSI design consists of relatively few symbols but many symbol *instances*. Thus a sequential program to check a hierarchical description [Hon 83] generally proceeds by checking *symbol definitions* (rather than symbol instances) recursively. The components of an individual symbol are compared against each other. If a component is an instance of another symbol, that symbol is checked in the same way (unless it has already been checked). This process continues until all symbols in the design have been checked. Three mutually recursive procedures are used:

Check Symbol (*CS*) examines a single CIF symbol and counts the number of overlaps between its components. It calls *CS* recursively on all the symbol's components and then calls *CSS* (below) on each *pair* of component symbols to see whether they overlap.

Table A-5 Circuits Analyzed by Gregoretti's Design-Rule Checker

Circuit name	Number of rectangles	Number of rectangle intersections	Number of symbols	Regularity index
FIFO	71,761	218,432	22	93.2
Cherry	7,416	20,395	35	13.14
Pads	4,263	13,123	8	16.33
Test	2,811	6,070	23	2.77
Slice	525	1,324	9	1.075

Compare Symbol to Rectangle (*CSR*) compares a rectangle against a symbol instantiation and counts the overlaps between the rectangle and the component rectangles of the symbol.

Compare Symbols (*CSS*) compares two symbols and counts the number of times a component rectangle of one overlaps a rectangle of the other.

The algorithm is structured as a top-down tree search. It begins with a call to *CS* for the top-level symbol. Eventually, it reaches a point where it compares two primitive rectangles. Most of the research on this algorithm has to do with finding an efficient method for pruning the search tree.

One of the most effective ways to prune the search tree is by finding the *bounding box* of each symbol. The bounding box is the smallest rectangle that surrounds all the symbol's components. If the bounding boxes of two symbols do not overlap, then obviously *CSS* does not have to check for overlap of their components either. At the time the description of a symbol is read in, its bounding box can be calculated, and a call to *CS* can be generated for it; this makes it easy to avoid the need to redo these tasks.

The main difference between a sequential and a parallel design-rule checker is that the parallel algorithm must "parcel out" some of the *CS*, *CSR*, and *CSS* tasks to other processors. This is done by maintaining a common queue of tasks that must be performed.Instead of performing a recursive task itself, a processor may place the description of it in the common queue rather than execute it locally. Processors that "run out of" recursive calls to perform then obtain more work by removing the first task description from the queue.

Due to the high branching factor of the task tree, it is impractical to place all recursive task invocations in the queue, lest the queue grow to an unmanageable size. According to a strategy developed by Lane [Lane 84], a task is placed in the queue only if the length of the queue is less than some predetermined upper bound (somewhat larger than the number of processors).

Gregoretti input the description of five different NMOS integrated circuits to his design-rule checker. Table A-5 summarizes their characteristics. The regularity index is defined as the ratio of the number of rectangle instances in the entire circuit to the number of rectangle definitions in the hierarchical description. It gives an idea of how concisely the hierarchical description captures the layout.

Execution time to check each design is shown in Figure A-55 speedup is shown in Figure A-56. Linear speedup is approached only by the Cherry and FIFO layouts. The others seem to saturate at some point, after which their execution times oscillate. Evidently, this is because the Test, Pads, and Slice layouts have terminal symbols with a large number of primitive rectangles. The time to check or compare these large leaf symbols bounds the execution time from below. Nonetheless, the Pads layout, being more regular than Test or Slice, can be executed in an order of magnitude less time.

A little reflection suggests that the lower execution-time bound can be circumvented by breaking *CS* invocations into calls of *CSR* and *CSS* as follows: For every element *i* of the symbol,

- if element *i* is a rectangle, then queue a *CSR* task for the element and a hypothetical symbol comprising all tokens that follow *i* in the symbol definition.
- if element *i* is an instance of some other symbol, then queue a *CSS* task between the called symbol and the hypothetical symbol described above.

Naturally, increasing the number of tasks in this fashion raises the overhead of queueing and recursion. For circuits that have no large leaf symbols, this can result in extra execution time (up to 20 percent in the case of FIFO). Where large leaf symbols are present, removing the lower bound more than makes up for this effect. Figure A-57 shows speedup of the design-rule checker modified in this way. Note that a speedup plateau is still reached, though at a much higher level than before. This is probably due to the asymmetry of the *CSR* and *CSS* tasks for the hypothetical symbols; the task for $i = 1$ still has a lot of work to do, while the last one must examine only two elements. Further research might examine whether it is feasible to break up these tasks further, or to avoid breaking up *CS* tasks that are smaller than a given threshold, in order to reduce the overhead.

A.20. Design-Rule Checking on StarOS

Algorithm name: Design-rule checking.
Cm configuration*: 50 Cm's, 5 clusters.
Operating system: StarOS.
Other software in environment: None.
Experimenter: Tom Lane, 1983–84.
Reference: [Lane 84].

Lane implemented design-rule checking under the StarOS operating system. He used the same general algorithm as Hon did [Hon 83], but his implementation differs in particulars from that of Gregoretti. Because he wished to use a straightforward algorithm to count the number of intersecting rectangles in the entire circuit, he was

Figure A-55 Execution Time of Gregoretti's Design-Rule Checker

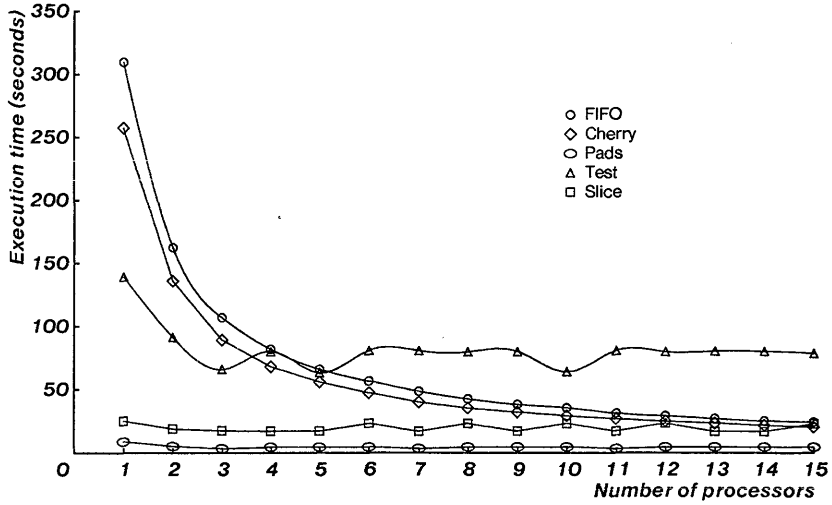


Figure A-56 Speedup of Gregoretti's Design-Rule Checker

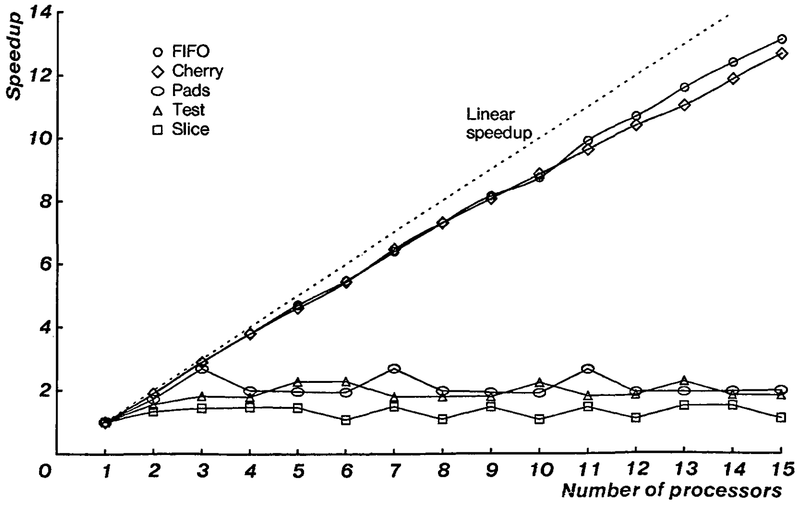
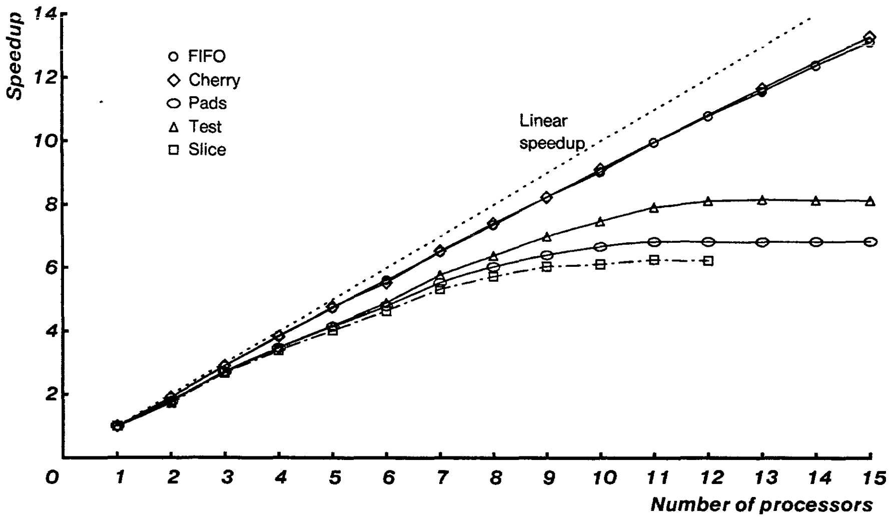


Figure A-57 Speedup of Gregoretti's Modified Design-Rule Checker



unable to avoid redoing *CS* for each instance of a symbol. He made multiple copies of the CIF description to decrease contention, and varied the number of copies. When there was a copy in the local memory of each Cm, the interprocessor communication bandwidth was quite low, since only the common queue of task descriptions needed to be shared. All his experiments broke *CS* invocations into calls of *CSR* and *CSS*, as Gregoretti did in his modified experiment.

Lane's program was used to analyze two VLSI circuit designs. One was a hardware FIFO queue, which used 1,717 CIF statements to describe 85,486 rectangles. The other was a random-access memory (RAM), using only 371 statements to describe 196,992 rectangles. Figure A-58 plots run time (mean of several runs) for each of the designs, with one copy of the CIF data per cluster; Figure A-59 shows speedup. While speedup is monotonically increasing for both the designs, the curves are slightly convex. The sublinear speedup is due in part to processor idle time, averaging about 5 percent for the FIFO design and 2 percent for the RAM.

If processor idle time were the only factor preventing linear speedup, the speedup would equal the average number of busy processors. Actually, the speedup is slightly less than this, a fact that can be attributed to two factors: contention for shared memory (the CIF data and the common task queue) and overhead of message operations to send and receive from the task queue. The magnitude of these effects can be estimated by dividing the speedup by the average number of busy processors. If this fraction is subtracted from 100 percent, the result is the percentage of run time lost due to contention and communication effects (Figure A-60).

This figure suggests that contention for CIF data is a more significant overhead

Figure A-58 Execution Time of Lane's Design-Rule Checker

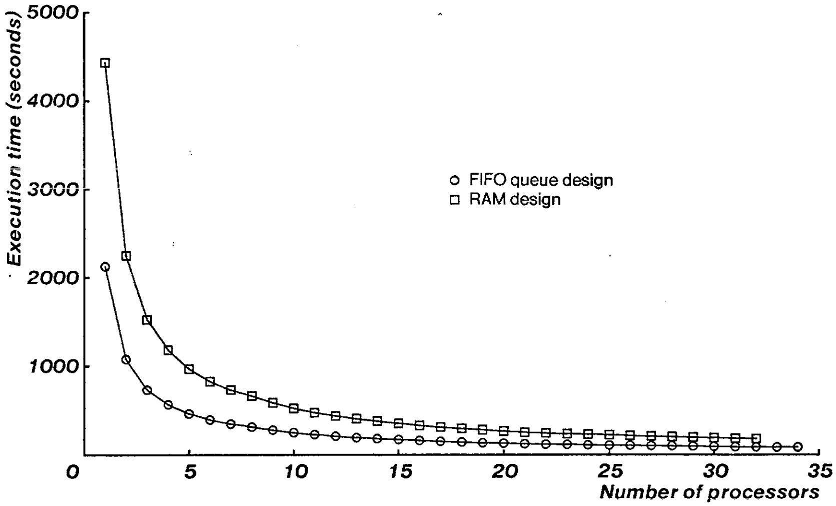
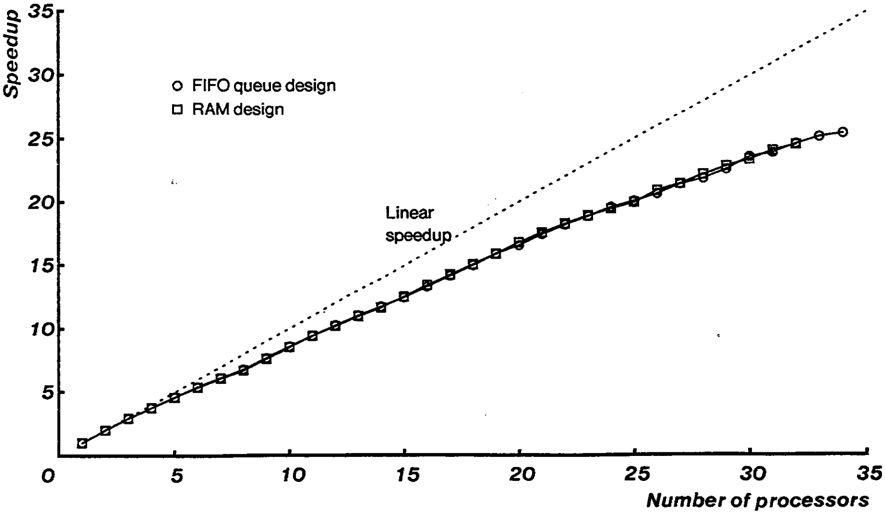


Figure A-59 Speedup of Lane's Design-Rule Checker



than contention for, and manipulation of, the common queue. Overhead increases significantly out to the first cluster boundary (at eight processors), then nearly levels off. If queue manipulation were the major overhead, we would expect the overhead to continue to rise—perhaps even faster, since costly intercluster accesses are now required. In fact, overhead drops slightly at nine and ten processors. This is consistent with the notion that a second, relatively contention-free copy of data in the new cluster lowers the overhead more than intercluster queue manipulation raises it.

More support for this explanation comes from overhead measurements when the experiment is run with a single copy of the CIF data (Figure A-61). The overhead continues to rise almost linearly, even after the first cluster boundary is crossed. In fact, the slope of the curve increases slightly, ostensibly as a result of the intercluster accesses. By contrast, replicating the data in each Cm differs little from using one copy per cluster. The reason is that the CIF data is held in objects that are smaller than 4K bytes, so memory accesses still require Kmap mediation (see Section 6.1.1). There is some improvement owing to reduced contention, but it is slight.

A.21. Parallel Production Systems: OPS3

Algorithm name: Parallel Production Systems: OPS3.
Cm configuration*: approximately 30 Cm's, 4 clusters.
Operating system: MEDUSA.
Other software in environment: NEST.
Experimenters: Mike Rychener, Joe Kownacki, Zary Segall, 1983–86.
References: [Rychener 80, Brownston *et al.* 85].

OPS3 is a language that can be used in rule-based programming for *production systems*, which have found wide application in artificial intelligence and the design of expert systems. Rule-based programming is founded upon the *production rule*, which consists of two parts: a left-hand side (LHS), which specifies conditions or patterns to be searched for, and a right-hand side (RHS), which represents changes to be made to working memory whenever the corresponding LHS is matched. The execution of an OPS3 program consists of a series of *recognize-act cycles*. In each cycle, a match is recognized in the LHS of some rule, and the action specified by its RHS is executed. It is possible for more than one LHS to match working memory during a cycle; in this case, a *conflict* is said to occur. One of the matches must be chosen by a process known as *conflict resolution* (*CR*).

CR proceeds as follows. Each time a match is found, information representing the match (specifying which rule matched and what data caused it to match) is recorded in a small data structure called an *instantiation*. During the CR stage, all the instantiations are evaluated based on criteria built into the program, and the one that best "promotes program execution" is chosen. This instantiation determines which RHS is to be executed. Working memory is then updated accordingly.

OPS3 programs tend to be characterized by large numbers of rules and long execution times. Matching consumes most of this time—90 to 95 percent in a typical program. While matching involves vast amounts of reading, however, it requires very little writing. Matching produces relatively few instantiations, and a single

Figure A-60 Overhead of Contention and Communication in Lane's Design-Rule Checker

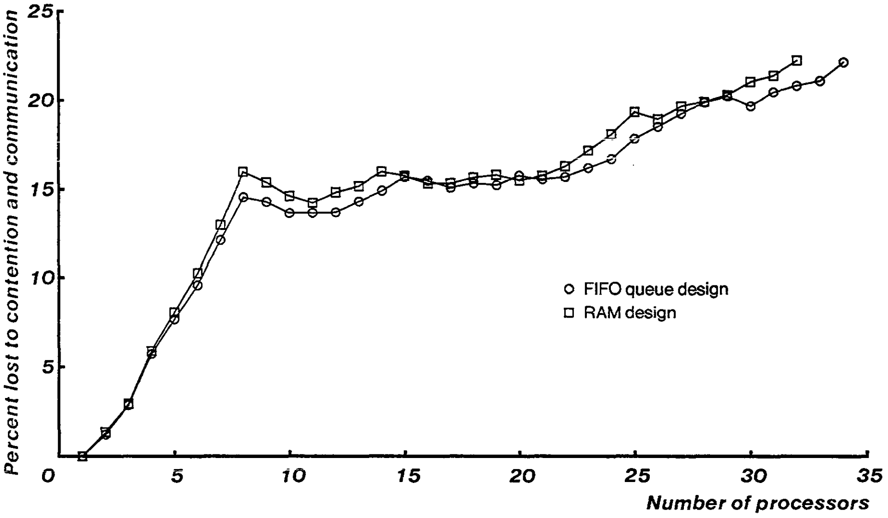
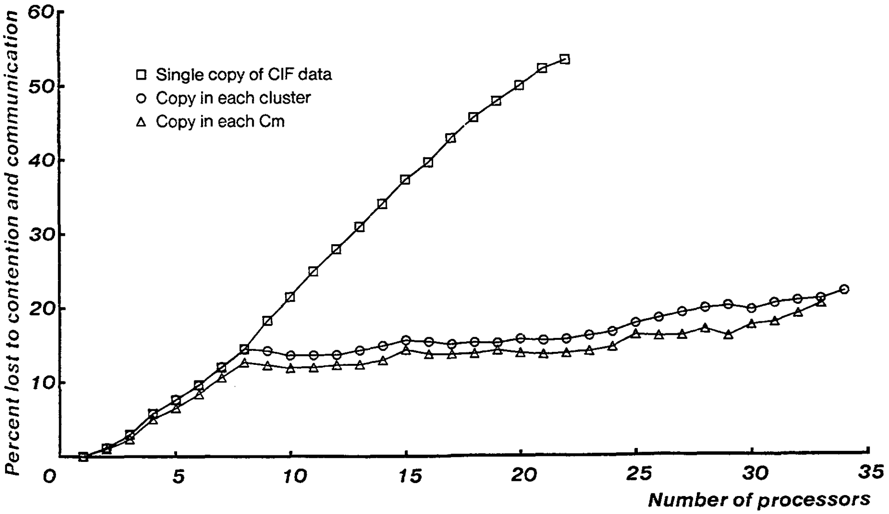


Figure A-61 Effects of Duplicating CIF Data on Overhead (FIFO Queue Design)



instantiation is unlikely to be larger than ten words. The other phases of the program are unlikely to be a bottleneck. CR is unlikely to limit performance because of the slow and staggered rate at which matches are produced. The typical RHS makes only a few modifications to working memory, so the action phase usually finishes quickly.

Fortunately, the matching phase is quite susceptible to parallelism. Each rule is examined independently so the rules can be parceled out among several processors. Because the rules are static in nature, it is not necessary to modify them once the program has begun execution. Relatively little information must be written, so contention for working memory is not a problem. Indeed, the matching phase of an OPS3 program almost perfectly fits the "small independent computations" paradigm of algorithms that approach linear speedup. The algorithm exhibits the multiphase structure introduced in Section 11.4.1. Both the length of the reconstitution phase and the other overheads are small (Figure 11-5), rendering the algorithm as a whole susceptible to near-linear speedup.

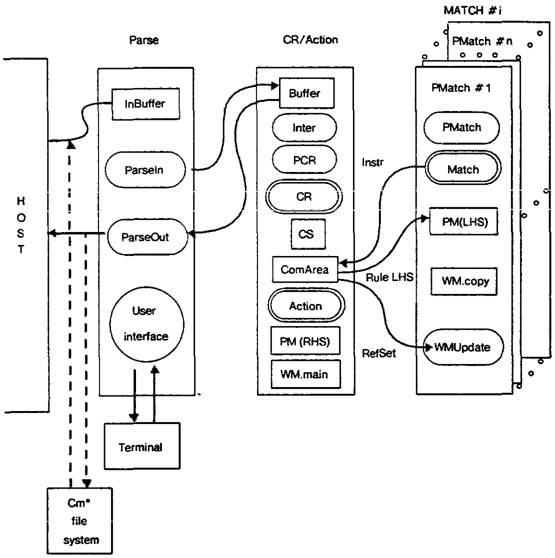
CM* IMPLEMENTATION. Parallel OPS3 is implemented on Cm* with three different kinds of processes (Figure A-62). First, there are the matching processes, each of which executes on a dedicated processor. The rules are partitioned between the local memories of these processors at load time. The memory that contains the rules is known as the *program memory*, or PM. In contrast to PM is *working memory*, or WM. This memory comprises a set of *working memory units*, or WMUs, which store data. Working memory is replicated in each processor and updated in parallel.

A single process is responsible for CR and action execution. It is active during the matching phase, comparing each new instantiation against the "best" instantiation so far. During the action phase it executes alone. In addition to these functions, it also handles communication between matching processes, as described below. A user-interface process rounds out the OPS3 task force. It handles communication with the terminal and sets the agenda for the other processors.

The only shared memory in the task force is a small communication area, which resides in the CR/action processor and is implemented as a pool of buffer slots. When matching processors produce an instantiation, they transfer it back to the communication area. As each matching process completes the recognition cycle, it signals CR/action using the same communication area. Communication occurs infrequently and does not significantly affect performance.

When the action phase modifies working memory, it places in the communication area a condensed representation of changes, known as the *refraction set*. Upon completion, the action phase signals the matching processes, which then update their copy of WM from the refraction set. This organization serves to remove all memory contention from PM and WM accesses. The only remaining sources of contention overhead are references to the communication area and updating with the refraction set. References to the communication area are so infrequent that contention is negligible; refraction-set contention is brief because the amount of data transferred is small. Most of the deviation from linear speedup is caused instead by unbalanced matching-processor workloads, a point that we shall return to later.

Figure A-62 Structure of Parallel OPS3

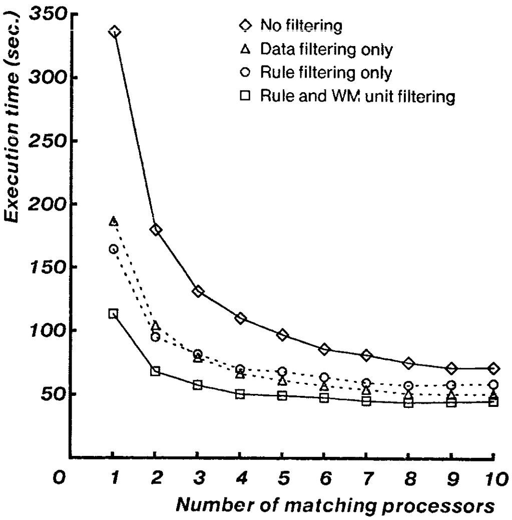


A naive, or *unfiltered*, implementation of matching would examine all the rules during each cycle. Furthermore, each variable unit (component of an LHS) would be compared to all WM units in each matching cycle. This strategy is inordinately expensive, and it is better to "filter" the rules or the data. Using the technique of *rule filtering*, the only rules examined during a matching cycle are those relevant to the changes made to WM during the previous action cycle.

Data filtering is a little more complicated to explain. Each variable unit and WM unit is endowed with three "anchors," which represent characteristics that best typify its attributes. For example, if an OPS3 program manipulated geometric shapes, an anchor might identify a certain rule as looking for a "ball" or a "block." Matching for this rule would ignore WM units that contained shapes other than balls or blocks, not even considering their other characteristics such as size and color. The choice of anchors is crucial in performance because,they control the order of comparison. The three most salient characteristics should be programmed as anchors. OPS3 limits the number of anchors per variable unit to three, as experience has demonstrated that additional anchors have little impact on performance.

THE TIC-TAC-TOE PROGRAM. The parallel production-system experiments on Cm* were run using a tic-tac-toe program written in OPS3. This program was tailored to provide a parallel-processing workload of interesting proportions but still of manageable size. The program makes moves for both the X and the O players. It also performs the functions of a "referee," examining the board for a win after each move. It consists of 23 rules, of which 9 are considered to determine the computational characteristics of the program. These examine the board for specific X and O pat-

Figure A-63 Performance of OPS3 Tic-Tac-Toe Program, with and without Filtering



terns to determine the next move, or if there is a next move. They consist of player rules, which examine the board for a line empty except for one player mark, a line that has two opponent marks, or a line with two player marks, along with other similar patterns. The referee rules examine the board for a line full of identical marks or no empty positions. The other rules are simpler and primarily perform control functions for the program. The working memory for the program represents each of the nine board positions by an individual WM unit. The value of a position's mark attribute is empty, X, or O.

Results. Figure A-63 shows how filtering affects execution time for different numbers of processors. The unfiltered case represents a situation in which all 23 rules of the tic-tac-toe program are examined each cycle. In addition, each variable unit of each LHS is compared to all WM units during the examination process. The combinatorial number of comparisons is drastically affected by filtering. For example, there are 27,705 such comparisons without data filtering and 3,762 with filtering. When rule filtering is added, this number drops to 2,877.

Without rule filtering, 989 rules are examined during the execution of the program; with rule filtering, this number decreases to 155. As a result, no more than 8 rules are examined each cycle. These 8 rules are the ones that produce the player moves during the game. Note that the minimum execution time for the filtered case is reached at the point where 8 matching processes are executing in parallel. In this situation, each rule resides in a separate process, so parallelism is maximal at this point.

Figure A-64 Observed vs. Optimal Execution Time for OPS3 Tic-Tac-Toe

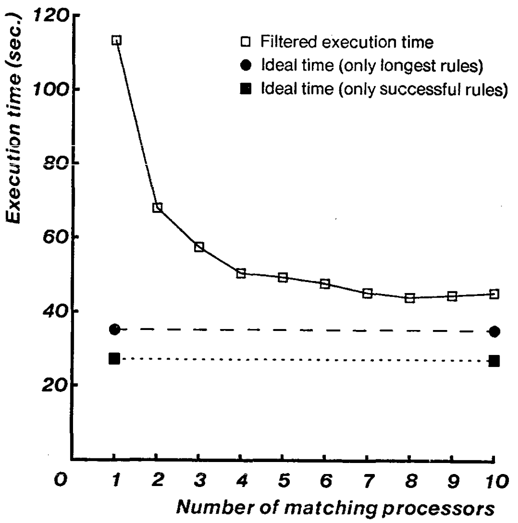


Figure A-65 Time to Make Local Copies of WM Data in Tic-Tac-Toe Program

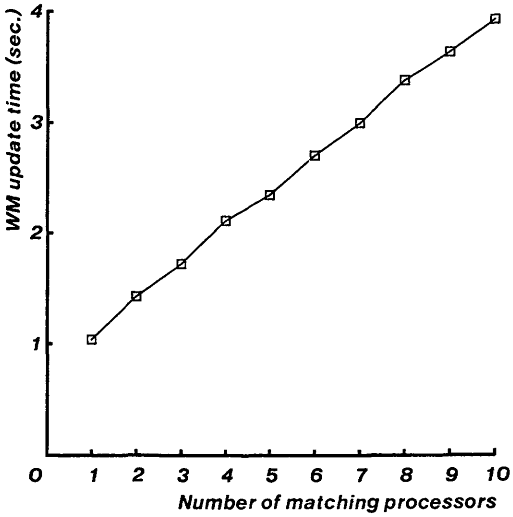


Figure A-64 compares the observed execution time for both kinds of filtering to the theoretical minimum execution time. The lower of the two "ideal" lines represents the calculated execution time (for one matching process) in the impossible case where only the correct rule is examined and executed each cycle. This situation is unrealizable because it assumes a 100 percent *a priori* knowledge of the OPS3 program execution sequence. In one sense, it does not even provide a fair comparison. The successful rules in a production-systems program are not necessarily the longest-executing rules. If each processor executed exactly one rule, the execution time would be determined by the longest rule, since CR cannot complete before all possible instantiations have been created. Using the time to match and execute the longest rule in place of the successful rule, the higher of the two "ideal" lines is obtained. At eight processors, the observed execution time is only 25 percent higher than the theoretical minimum.

As we have seen so often before, the poorest algorithm again shows the best speedup. At eight processors, the speedup of the unfiltered algorithm is 4.5, while it is only 2.6 in the case where both kinds of filtering are used. Part (generally 5 to 10 percent) of the deviation from linear speedup can be explained by the cost of maintaining local copies of WM in each matching processor. The cumulative cost over 40 cycles is shown in Figure A-65. The curve represents both filtered and nonfiltered cases because the same changes are made to WM in both cases. The fact that the deviation from linearity is so large testifies to the effect of unbalanced workloads.

The configuration of matching and CR / action processes was varied in an attempt to discover how distribution across clusters would affect execution time. Surprisingly, placing the CR / action process in a separate cluster had no perceptible effect on performance. The OPS3 / Cm* design strove to minimize interprocess communication and succeeded to the extent that communication overhead was insignificant.

Acknowledgment. The discussion of the LMSK algorithm in Section A.18 was taken directly from [Mohan 83].

Appendix B
Coscheduling Performance

Section 5.5 explained the need to ensure that all activities (or processes) of a task force are simultaneously assigned to processors in order to prevent activities from blocking for unnecessarily long periods of time when they communicate via messages. Suppose, for example, several processes are executing on a multiprocessor and sending and receiving messages among themselves, but that the system's scheduling policy allows half the processes to execute only in odd time slices and the other half only in even time slices. If the processes are interacting frequently, it is likely that most or all of the processes in the executing half will block awaiting messages from processes in the descheduled half. Regardless of the raw speed of the processes or of the interprocess communication mechanism in this example, the processes will be able to interact only as frequently as the system reschedules processors. Even more intelligent schedulers than the one in this example can produce similar behavior unless they take into account the communication patterns of the processes.

A task force is said to be coscheduled when all its runnable activities are executing simultaneously on different processors. Each of the activities in that task force also is said to be coscheduled. If at least one activity of a task force is executing but the task force is not coscheduled, then it is said to be *fragmented*; the collection of executing activities is referred to as the *executing fragment* of the task force. The coscheduling algorithms described in this section include both allocation and scheduling. It is important that scheduling be efficient because it occurs frequently. Allocation occurs less often, so it can involve more complex decisions. All the algorithms below assume that activities are never moved once they have been created and that a given activity executes on only a single processor.

The system is considered to contain p processors. At any given time, the scheduler on each processor can choose between at most a activities.[1] Thus activity space consists of *pa activity slots* to which activities may be assigned. Task-force allocation consists of assigning each activity of the task force to an empty activity slot. The value a is assumed to be large enough so that allocation always succeeds. Scheduling consists of selecting one activity slot in each processor, the activity in which it is to be executed by that processor. The slot assignment of an activity is not changed during its lifetime, and each activity is allowed to execute only on the processor to which it was assigned. The algorithms assume that no task force contains more than p activities.

[1] In MEDUSA p ranges from about 10 to 50, depending on the configuration, and a is fixed arbitrarily at 16.

B.1. Three Algorithms for Coscheduling

The three coscheduling algorithms discussed in the following sections derive their "flavor" largely from their views of activity space. The first algorithm views activity space as a two-dimensional matrix of activities, while the second and third algorithms view activity space as a linear sequence.

B.1.1. The Matrix Method

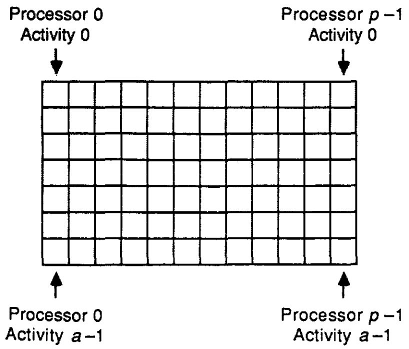
The matrix coscheduling algorithm is a very simple one, and it performs surprisingly well. The space of all activity slots is organized as a *matrix* with *a* rows and *p* columns (see Figure B-1). The activity slots in one column comprise all those belonging to a single processor. A row of the matrix selects one activity from each processor.

Allocation: See if there are enough unused slots in row 0 of the matrix to accommodate all the activities of the task force. If not, then attempt to assign all the activities to slots in row 1, and so on, until a single row is found that can hold the entire task force.

Scheduling: This algorithm uses a round-robin mechanism to multiplex the system between the different rows of the matrix. In time slice 0, each activity in row 0 is given highest execution priority on its processor, thereby coscheduling all task forces in that row. In time slice 1, row 1 is coscheduled, and so on, until all task forces have been scheduled. Then return to row 0 and repeat.

When a row is scheduled for execution, it is likely that one or more of the activity slots in that row will either be empty or contain activities that are blocked (e.g., while awaiting terminal input). Each processor so affected scans its activities for an activity in another slot that is runnable. If row *n* has highest priority, then the other slots for that processor are searched for a runnable activity in row order $n-1$, $n-2$, ...,

Figure B-1 Two-Dimensional Process Space in the Matrix Algorithm



0, $a - 1, a - 2, \ldots, n + 1$.[2] If an activity in one of these slots is runnable, it is allowed to execute as an *alternate*. Unless empty activity slots cause all the other activities in the alternate's task force to run, the alternate will execute as a fragment.

The alternate selection method presented above is applied independently by each processor; no attempt is made to select alternates in a way that maximizes coscheduling, except that the same selection algorithm is run by each processor. Thus it is almost certain that alternates will execute as fragments; an alternate that really needs coscheduling will block as soon as it attempts interactions with descheduled activities. The method does have the nice property that there is a clean separation between global and local scheduling decisions. At a global level, all that must be done is to select the high-priority row of the matrix. Given the number of that row, each processor can schedule itself and perform alternate selection independently. The simulation results presented below suggest that the gains to be had by selecting alternates centrally would be small; furthermore, a central scheduler would have to decide what to do with a processor anytime there is a change in the execution status of any activity belonging to that processor. With the simple-minded alternate selection, changes in execution status are handled locally by the kernel of the activity's processor. Global intervention occurs only on time-slice boundaries.

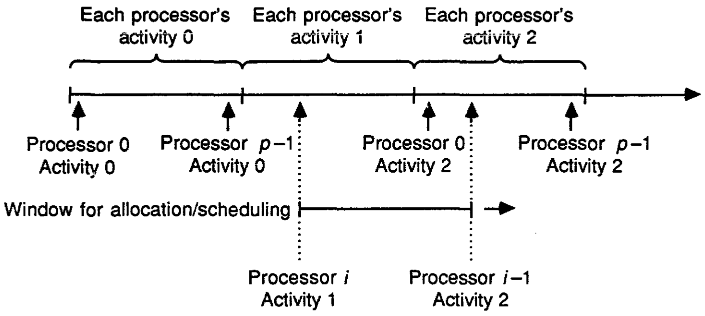
The attractive features of the matrix algorithm are the simplicity of its allocation and scheduling algorithms and the clean separation between local and global scheduling decisions. The algorithm has several drawbacks. First, the rigid partitioning of activity space into nonoverlapping blocks means that there are likely to be many unused activity slots in each row, especially when there are many task forces with just over $p/2$ activities. This inefficiency is reminiscent of *internal fragmentation* in paged virtual-memory systems. The other two algorithms are attempts to solve this problem; we shall see that they, in turn, suffer from an effect similar to *external fragmentation* in segmented-memory systems. A second disadvantage of the matrix method is the simple-mindedness of the alternate selection algorithm, which could cause opportunities for coscheduling to be missed. Finally, the matrix algorithm causes the execution priority of the whole system to change at the same time. Thus shared facilities used for context swapping (e.g., paging devices) will be loaded unevenly.

B.1.2. The Continuous Algorithm

Most of the drawbacks of the matrix algorithm arise because of the rigid partitioning of process space into the rows of the matrix. The continuous algorithm uses a different view of activity space to achieve denser packing and smoother scheduling (Figure B-2). For this algorithm, activity space is viewed as a *sequence* of activity slots. The activity slots in any p consecutive positions of the sequence belong to different processors. The allocation and scheduling algorithms consider at a particular moment a *window* of p consecutive positions in the sequence and slide the

[2] The direction of this search is significant. The activity in row $n - 1$ was scheduled in the previous time slot and hence is most likely to have all of its memory pages and other execution state still loaded.

Figure B-2 Activity Space for the Continuous Algorithm


Each processor's activity 0
Each processor's activity 1
Each processor's activity 2
Processor 0 Activity 0
Processor p–1 Activity 0
Processor 0 Activity 2
Processor p–1 Activity 2
Window for allocation/scheduling
Processor i Activity 1
Processor i–1 Activity 2

window across the sequence over time. For purposes of the algorithms below, activity slot 0 in processor 0 is considered to be in the leftmost position of the sequence, and activity slot $a-1$ of processor $p-1$ is considered to be in the rightmost position.

Allocation

Place a window with p slots at the left end of the activity sequence. See whether there are enough empty slots in the window to accommodate the new task force. If not, then move the window one or more positions to the right until the leftmost activity slot in the window is empty but the slot just outside the window to the left is full. Repeat this until a window position is found that can contain the entire task force.

Scheduling

Place a scheduling window of width p processors at the left end of the activity sequence. At the beginning of each time slice, move the window one or more slots to the right until the leftmost activity in the window is the leftmost activity of a task force that has not yet been coscheduled in the current sweep. When the window has advanced far enough that all existing activities have received execution time, return the window to the left side and start a new sweep. When the window contains empty activity slots or activities that are not runnable, use the alternate selection mechanism from the matrix algorithm.

There are several reasons for moving the allocation and scheduling windows in the way described above. In the case of the allocation window, there is no particular advantage in testing a window position if its leftmost slot is occupied; moving the window another slot will eliminate the occupied slot and may add an unoccupied slot at the right end. In addition, there is no point in testing a window position if an unoccupied slot has just been lost off the window's left edge; the best that could have happened is to add another unoccupied slot on the right side, which makes the new window position equivalent to the old position. There is much in common between this method of task-force allocation and the "bit-map" method of memory allocation; see, for example, Chapter 8 of [Habermann 76].

Figure B-3 Unequal Treatment of Task Forces by Naive Continuous Algorithm

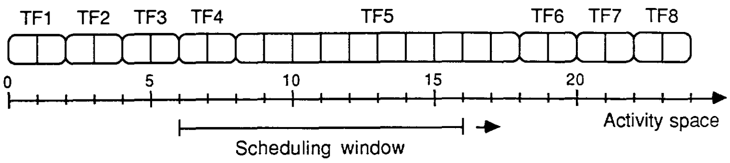


Table B-1 An Example of Unequal Treatment of Different Task Forces

	Coscheduled slices per scheduling sweep	
Task force	Move window to next task force	Move window to noncoscheduled TF
TF1	1	1
TF2	2	1
TF3	3	1
TF4	4	1
TF5	1	1
TF6	1	1
TF7	2	1
TF8	1	1

There are similar arguments for moving the scheduling window so that its leftmost slot is always the leftmost slot of a task force. If the leftmost window slot is empty, nothing is lost by moving the scheduling window another slot to the right. If the leftmost slot is occupied but its activity is not the leftmost slot of a task force, then that task force cannot possibly be coscheduled; we might as well advance the window to the next task force. Moving to the leftmost slot of a task force guarantees that every window position coschedules at least one task force and every task force is coscheduled at least once in every sweep.

If the scheduling window were simply advanced one task force each time slice, however, task forces of different size and location would receive unequal treatment. For example, consider the situation of Figure B-3 and Table B-1. where *p* is 10 and a task force with ten activities is surrounded by several task forces with two activities. The scheduling window tends to move more slowly across small task forces than across large ones, thereby giving the small ones more coscheduled time slices; in the left column of the table, notice that TF4 receives four coscheduled slices to every one for TF5. In addition, large task forces cast a "shadow" over the task forces just to their right; although TF4 and TF6 each have two activities, TF4 receives much more coscheduled time than TF6. Finally, task forces at the left end of the activity sequence receive harsher treatment than those at the right end (the left end of the sequence casts a shadow equivalent to that of a task force with *p* activities). To reduce this discrimination, the scheduling window should be moved each time slice, until (a) its leftmost activity is the leftmost activity of a task force and

(b) that task force has not yet been coscheduled in the current sweep across activity space.

The continuous algorithm reduces the problems caused by the rigidity of the row structure in the matrix algorithm. Task forces can be packed more tightly in activity space because empty slots that would have been in separate rows under the matrix algorithm may still lie within the allocation window used by the continuous algorithm. The context-swapping load is distributed more evenly by the continuous algorithm than the matrix algorithm because not all processors change scheduling priority each time slice. In spite of its dense packing of task forces, the continuous algorithm generally requires fewer position checks during allocation than the matrix algorithm; for details, see the results of the simulation described below.

Unfortunately, the continuous algorithm still suffers from several drawbacks. The most serious problem concerns divisions within task forces. When the activity sequence becomes populated with many small "holes" (contiguous empty slots), new task forces are likely to be divided among several holes; the distance in the activity sequence between the leftmost and rightmost activities of a new task force, referred to as its *width*, may be substantially greater than the size of the task force. A small task force with a large width will have similar scheduling properties to a large task force with the same width, as there are only a few scheduling window positions for which the task force will be coscheduled. As task forces become split between several holes, even the improved scheduling algorithm discussed above becomes unfair: Small contiguous task forces will receive many more coscheduled time slices than those that are large or badly split. This situation is analogous to *external fragmentation* in segmentation systems, where utilization of primary memory degrades substantially if the memory becomes too fragmented. The simulation results will verify that the continuous algorithm's behavior deteriorates badly as the system load increases and activity space becomes fragmented.

B.1.3. The Undivided Algorithm

This algorithm is identical to the continuous algorithm in every respect except that during allocation, all the activities of each new task force are required to be contiguous in the linear activity sequence (i.e., the task force may not be divided between two or more holes). The undivided algorithm does not pack activity space quite as efficiently as the continuous algorithm, but it reduces fragmentation and thus results in substantially better system behavior under heavy loads (see the simulation results). In fact, this algorithm performed smoothly under a variety of system conditions and showed the least sensitivity of the three algorithms to factors such as task-force size and load.

B.2. Analysis of the Algorithms by Simulation

A simulation model was written to provide a more quantitative understanding of the algorithms. The simulator analyzes the scheduling behavior of a 50-processor system using each of the above three scheduling algorithms under a variety of synthetic loads.

B.2.1. Parameters of the Simulations

Because of the paucity of experience with production workloads on general-purpose multiprocessors, there exists almost no information on how such systems are exercised by concurrent programs. In addition, the simulation was carried out relatively early in the history of Cm*, when there were relatively few experimental programs to analyze. The simulation experiments were therefore based on a simple-minded model parameterized in the following way:

Size — The expected task-force size (individual task forces were chosen from a pseudo-random exponential size distribution).

Load — The expected ratio of the number of runnable activities to the number of processors. (This value is used in generating task-force arrivals. Arrivals are most likely to be generated when the load is zero; the arrival rate declines linearly until it reaches zero when the load is twice as high as the expected load. This models the reluctance of users to submit jobs to a heavily loaded system.)

Lifetime — The expected lifetime of task forces (actual lifetimes are chosen from a pseudorandom exponential distribution).

Idle fraction — The simulator permits task forces to be in the system without being runnable. The idle fraction is defined as the expected fraction of the task forces' lifetime that they are not runnable; task forces are made runnable and not runnable for pseudorandom exponential periods of time based on the idle fraction. Most of the results presented below assume a zero idle fraction; Section B.2.4 discusses the effects of idle time.

For lack of a suitable model of communication between activities of a task force, the simulator does not allow a task force to be partially runnable. All the activities of a task force are made runnable or not runnable together. Fortunately, this assumption leads to a pessimistic estimate of coscheduling.[3]

Since the purpose of the simulation is to measure how effective the three algorithms are, most of the results are presented in terms of *coscheduling effectiveness*. Coscheduling effectiveness is the mean (across all the time slices of a simulation run) of the ratio of the total number of processors executing coscheduled activities to the total number of processors with runnable activities. A coscheduling effectiveness of 1.0 is ideal. Coscheduling effectiveness measures the system's ability to coschedule task forces, *not* the response time that will be seen by individual users or the overall performance of the system. In situations where coscheduling is not needed (for example, when there is no interprocess com-

[3] In a real system, if a task force is only partially coscheduled, the executing fragment might block on communication with the descheduled part of the task force; another task force might become coscheduled by the alternate selection mechanism. In the simulator, the fragment continues to execute even though it is not coscheduled, and hence it leads to a lower overall estimate of coscheduling.

munication), overall system performance may not depend on coscheduling effectiveness.

B.2.2. Effectiveness as a Function of Load

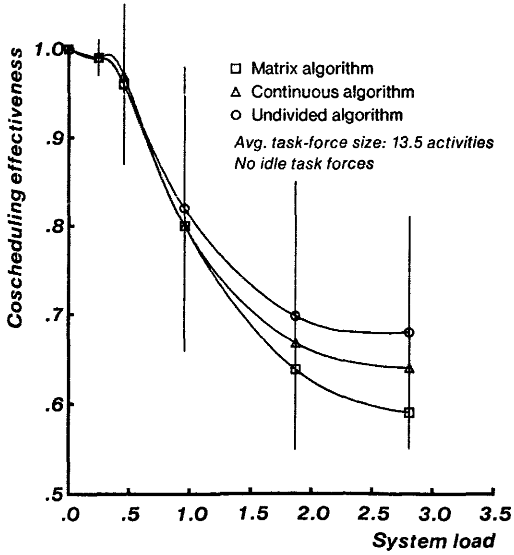
Figure B-4 plots the *coscheduling effectiveness* of the three coscheduling algorithms as a function of system load for a mean task force size of 13.5 activities and no idle time. As expected, all the algorithms performed quite well for underloaded systems and degraded as the system load increased. Above a load of two or three, the coscheduling effectiveness leveled off at about two-thirds.

The same overall behavior as in Figure B-4 was observed for all system conditions that were simulated. This can be understood by considering the two factors that result in a coscheduling effectiveness less than 1.0:

Straddling. In the continuous and undivided algorithms, it is possible for a task force to straddle the right end of the scheduling window. If this occurs, then those activities inside the window will execute as a fragment and lower the system's coscheduling effectiveness.

Alternate Selection. In all three algorithms, alternate selection occurs when there are vacant activity slots in the current high-priority portion of activity space. Because of its simple-mindedness, alternate selection will almost never result in coscheduling. If there are other runnable activities in the processors for

Figure B-4 Coscheduling Effectiveness of Three Algorithms



which alternate selection occurs, they will likely execute as fragments and degrade the system's coscheduling effectiveness.

For an average system load less than 1.0, almost all activities are allocated in the first row (for the matrix method) or in the first p activities in the sequence (for the other two methods). Thus straddling almost never occurs. When vacant slots exist in the high-priority portion of activity space, it is likely that the processors involved contain no activities at all; since the coscheduling effectiveness is determined only by processors with runnable activities, these holes will not degrade effectiveness. When the average system load becomes greater than 1.0, then both straddling and alternate selection begin to occur, and scheduling effectiveness degrades. Straddling and alternate selection are functions of how task forces are packed into activity space; for large loads the packing arrangement becomes independent of load (it depends only on the task-force size distribution and the allocation algorithm), so coscheduling effectiveness levels off.

Figure B-4 also plots 80 percent confidence intervals for the undivided algorithm, shown as vertical lines in the figure. The confidence intervals are for individual time slices: in any given time slice, one can expect the coscheduling effectiveness to fall within the range of the bars with 80 percent probability. Note that the short-term fluctuations for any single algorithm are larger than the differences between the algorithms. In spite of the short-term fluctuations, however, the average over several time slices converges very quickly. Different runs with different random seeds produced identical average effectiveness values to within several significant digits.

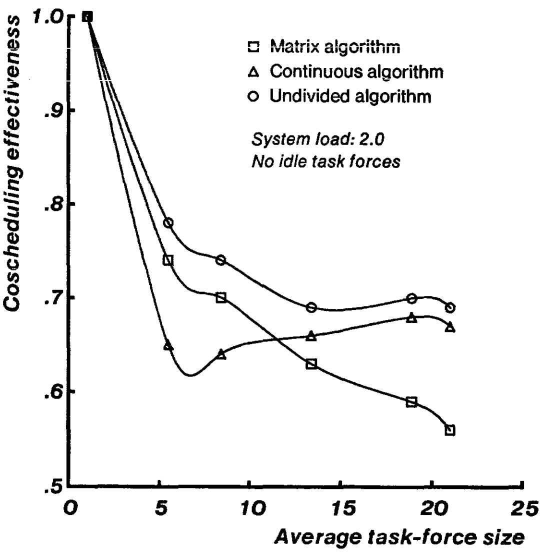
B.2.3. Effectiveness as a Function of Task-Force Size

The differences between the algorithms are most apparent in comparisons of system performance under varying task-force sizes. Figure B-5 shows how scheduling effectiveness varies as a function of task force size for an average load of two. For each size, the algorithm showed similar behavior (as a function of load) to that of Figure B-4, with variations only in the level at which effectiveness stabilized for high loads.

The data in Figure B-5 supports the predictions made earlier. For very small and very large task forces, all three algorithms should perform quite well. In the limiting cases of average size 1 or 50, the scheduling effectiveness of each algorithm is 1. The continuous algorithm performs worst when there are many small task forces. Under these conditions, the average hole size will be small; task forces are likely to be fragmented between several small holes and hence have widths much larger than their sizes. As the average task-force size increases, so does the average size of the holes; task-force fragmentation occurs less drastically, so coscheduling effectiveness improves.

The matrix algorithm is not as prone to fragmentation as the continuous algorithm, so its performance does not fall as rapidly when task-force size increases. As the average task-force size approaches 25 (one-half the number of processors), however, the matrix algorithm is unable to pack them very densely in the matrix rows. At an average size of 25, activity space will only be about 50 percent packed;

Figure B-5 Coscheduling Effectiveness as a Function of Mean Task-Force Size



alternate selection does not produce much coscheduling, so the coscheduling effectiveness is only about 50 percent.

The undivided algorithm offers a compromise where large task forces can be packed relatively densely, but small task forces do not cause fragmentation. It shows less sensitivity to task-force size than either of the other two algorithms. Since no data is available on what kind of task-force size distribution to expect in actual systems, the undivided algorithm appears to be the best choice. Because the undivided algorithm performed worst with a mean task force size of 13.5 activities, that size was used for most of the remaining measurements discussed in this section.

B.2.4. Idle Task Forces

In actual systems, all activities cannot be expected to be runnable all the time. If a task force becomes idle while waiting for some external event such as terminal input, then its activities occupy slots in activity space without being runnable. This will likely reduce the degree of coscheduling in the system by causing more alternate selection to occur. Experience with time-sharing systems indicates that most programs spend most of their time in an idle state, so the simulator was modified to provide data on the effects of idle task forces.

Figure B-6 plots coscheduling effectiveness as a function of load for an average task-force size of 13.5 and an idle fraction of one-half. Although the general behavior of the algorithms is not altered greatly by idle time, a comparison of Figures B-4 and B-6 shows that the continuous algorithm suffers somewhat more from idle time than

Figure B-6 Coscheduling Effectiveness with an Idle Fraction of 0.5

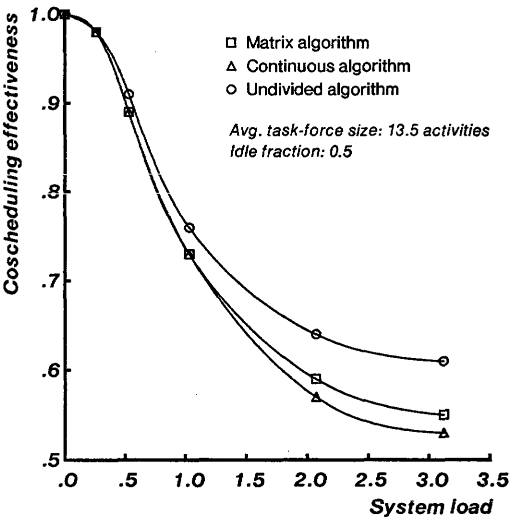
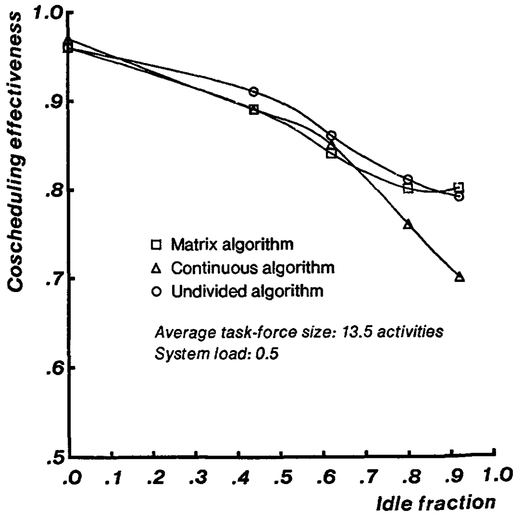


Figure B-7 Coscheduling Effectiveness as a Function of Idle Time



the other two algorithms. Figure B-7 charts coscheduling effectiveness as a function of idle fraction for an average system load of one-half. For very large idle fractions the performance degradation is one-third or greater.

The curves of Figure B-7can be explained as follows. In the limiting case of very high idle fractions, only one task force in the high-priority portion of activity space (the current row or scheduling window) will be runnable. If any other task forces are to be coscheduled, then that coscheduling must occur as a result of alternate selection. Note also that with an average load of 0.5 and an average task-force size of 13.5, on the average two or more task forces will be runnable at any given time. The continuous algorithm tends to fragment task forces so that it is quite likely that the runnable task forces will overlap in their processor usage; hence only one will be coschedulable. Both the undivided algorithm and matrix algorithm tend to allocate task forces in contiguous slots (the undivided algorithm by design, the matrix algorithm by a quirk of its implementation), so there will be fewer overlaps between the runnable task forces. Thus under high idle fractions the matrix and undivided algorithms have somewhat better characteristics than the continuous algorithm.

B.2.5. Relative Treatment of Different-Size Task Forces

Figure B-8 graphs the *coscheduled fraction* as a function of task-force size for the undivided algorithm under three different system loads. The coscheduled fraction for a task force is defined as the ratio of the number of coscheduled time slices received by the task force to the number of time slices when at least one of the task force's activities is executing. Under light loads all task forces are nearly always coscheduled, but under moderate or heavy loads the largest task forces are coscheduled less than one-third of the time. Although Figure B-8 contains data for just the undivided algorithm, the corresponding data for the other two algorithms is similar.

In Figure B-9 the average fraction of the activities of a task force executing is plotted as a function of task-force size. Once again, all task forces receive good, nearly identical treatment when the system is lightly loaded; large task forces suffer somewhat more than small task forces as the load increases. It is encouraging to note, however, that even under heavy loads, the average executing fragment for large task forces contains more than half the activities of the task force. The vertical bars in Figure B-8 are 80 percent confidence intervals for the behavior of any single task force in the load = 1.0 curve.

Figure B-10 shows the relative number of time slices given to each size executing fragment for large task forces (30 to 50 activities), using the undivided algorithm under three different system loads. The rightmost point in each curve is the overall coscheduled fraction for task forces with sizes between 30 and 50.

Figure B-8 Fraction of Time that Task Forces Are Scheduled

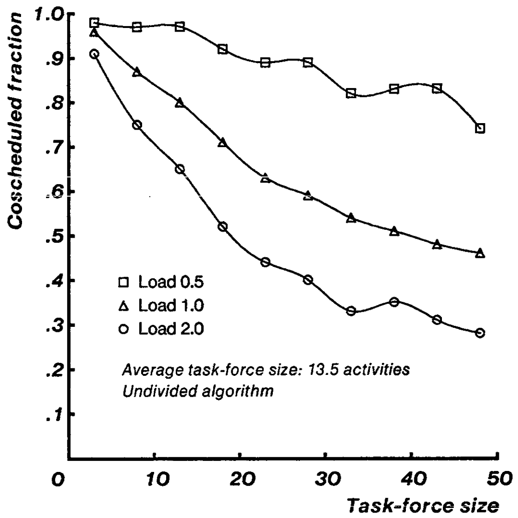


Figure B-9 Average Size of Executing Fragments, as a Fraction of Total Task Force

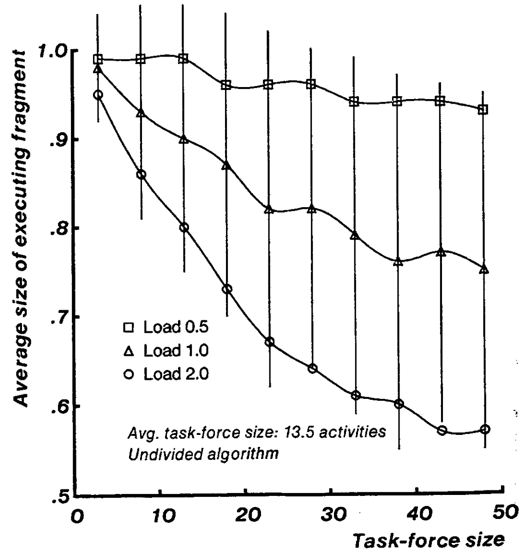
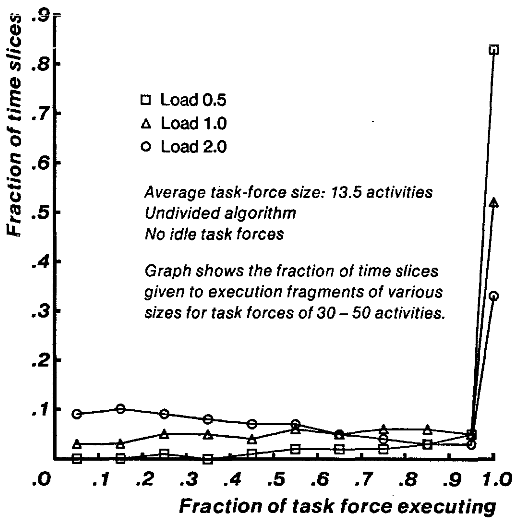


Figure B-10 Fraction of Time Slices Used by Various-Sized Task-Force Fragments



B.3. Efficiency of Allocation and Scheduling

The simulator gathered data about the amount of work involved in allocation and scheduling (see Table B-2). During allocation the undivided algorithm required many more potential task-force locations to be checked than either of the other two algorithms. The continuous algorithm consistently required the least number of checks but produced somewhat less coscheduling as a result. The continuous and undivided algorithms provided smoother processor scheduling, with only about half as many context swaps occurring in each time slice as occurred with the matrix algorithm.

Table B-2 Allocation and Scheduling Statistics for the Three Algorithms

	Matrix algorithm	Undivided algorithm	Continuous algorithm
Average number of positions examined during each allocation	1.6	1.3	8.7
Average fraction of processors context swapping each time slice	.67	.37	.37

B.4. Summary

The coscheduling of task forces is in many ways very similar to the dynamic allocation of memory. Naive scheduling algorithms can easily lead to thrashing, but with a little care, quite acceptable degrees of coscheduling appear to be obtainable. The algorithms presented here represent three sets of trade-offs involving the cost of allocation and scheduling, the density of packing (both within a task force and for the system as a whole), and forms of internal and external fragmentation. The continuous algorithm provides fast allocation and scheduling and dense packing of the system as a whole, but it is prone to severe external fragmentation. It consistently performs worse than the other two algorithms. The matrix algorithm provides fast allocation and scheduling but suffers from internal fragmentation when many large task forces are present. Its overall performance, however, was only occasionally worse than the undivided algorithm, and its implementation is by far the simplest. For these reasons the matrix algorithm is implemented in the current version of MEDUSA. The undivided algorithm requires rather more effort in allocation than either of the other two algorithms but provides for dense packing both within task forces (they are always allocated contiguously) and for the system as a whole; this resulted in insensitivity of the algorithm to several conditions that had strong effects on the other algorithms.

For moderate system loads (around 1.0), a coscheduling effectiveness of between 0.7 and 0.9 can be expected. Application programmers can expect small task forces to be coscheduled almost all the time under almost any conditions; under moderate loads, large task forces will be coscheduled about half the time, with the average executing fragment containing about four-fifths of the activities of the task force.

Some caution must be exercised in interpreting the simulation results because the algorithms and the simulation program do not take into account several factors that could affect performance. The simulator does not measure the effect of location specifiers on the algorithms (see Chapter 8); extensive use of location specifiers is most likely to harm the continuous and undivided algorithms by increasing the width of task forces. The algorithms described here contain no provision for the dynamic addition of activities to task forces. Although the algorithms can be extended to include dynamic task-force expansion, it appears that coscheduling will suffer somewhat as a result. The random distributions used in the simulation were chosen almost arbitrarily because no data is available on what kind of load to expect in a running system. It is encouraging to note that the simulation results are only mildly sensitive to the task-force size distribution and idle fraction. If the real-world load differs substantially from the simulated load, the results presented here could be invalidated.

Acknowledgment. This appendix has been condensed from [Ousterhout 82].

Appendix C
Performance of Parallel Algorithms

Chapter 11 studied several factors that affect the performance of parallel algorithms. One of the major tools in this study was a performance model created by Vrsalovic *et al.* [Vrsalovic *et al.* 84a, Vrsalovic *et al.* 84b]. Models of algorithm / architecture combinations are essential, both to discover how to optimize algorithms for existing architectures and to develop new architectures whose characteristics are well suited for running important algorithms. This appendix develops the Vrsalovic model, which uses the characteristics of architectures and algorithms to predict the performance of a particular algorithm on a particular architecture.

Several earlier multiprocessor models [Bhandarkar 75, Baskett and Smith 76, Marsan and Gregoretti 81, Marsan and Gerla 82] were based on statistical methods, predicting statistical mean values for performance over some time interval. It was shown [Vrsalovic and Siewiorek 83] that the performance of a parallel system in the short term—during one loop iteration, for example—can also be used to model long-term performance. Like the model of Kinney and Arnold [Kinney and Arnold 78], the model presented here represents a parallel computation as a sequence of identical loop iterations. Unlike a standard queueing or simulation model, it does not account for randomness in loop iteration times. Hence it may be unsuitable for applications where unpredictable branching or data dependencies cause iteration times to vary widely. Although it models only the parallel phase of a multiprocessor computation, it can easily be extended to consider a serial phase as well. It is computationally simpler than many queueing or simulation models, however, and thus may offer a more tractable means of dealing with complex but regular computations.

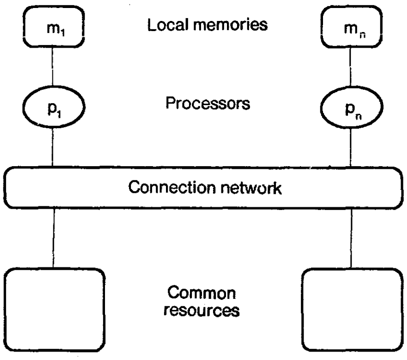
The model presented here explains performance differences between synchronous and asynchronous algorithms. It accounts for the effects of different processor and global-memory speeds. It also considers the different ways in which algorithms decompose for parallel processing—for example, dividing a computation into N processes does not necessarily mean that each process does $1/N$th as much work as the original. The model will be described in two stages. In the rest of this section, we outline a simple and intuitive model whose capabilities are limited. Later sections describe a more detailed and powerful version of it.

C.1. A Simple Analytical Model

The model assumes a multiprocessor composed of N processors, each of which has its own private memory for code and local data. In addition, each processor has

This appendix has been adapted from [Vrsalovic *et al.* 84b] by Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer.

Figure C-1 General Architecture of a Multiprocessor System



access, via the interconnection network, to the common resources of the system, notably a global memory (Figure C-1). Access time to the global memory is independent of the location of the processor accessing it. Requests for common resources are granted on a FIFO basis rather than according to a priority scheme. This allows the waiting time to be modeled as a linear function of *N*.

The model assumes that the parallel workload is infinitely decomposable. Each process performs a series of identical iterations. Each iteration is made up of a single period of processing and a single period of access to global memory (access to local memory is "charged" to processing time rather than being modeled separately). Due to the large number of iterations, startup transients are neglected. We defined the following:

$t_p \equiv$ the processing time within an iteration

$t_a \equiv$ the global access time within an iteration

$t_w \equiv$ the waiting time due to contention for global resources

The model utilizes the concept of *effective processing power*, which is defined as the number of processors in the system times the average utilization of each processor. The effective processing power *E* can be expressed as

$$E \equiv \sum_{N} \frac{t_a + t_p}{t_a + t_p + t_w} \tag{1}$$

Let us assume that the load is balanced (i.e., that t_p and t_a are the same for all processors). In the case of a synchronous workload, all processors must finish the current iteration before any processor may begin a new iteration. In the worst (that is, most synchronous) case, all processors attempt to access memory at precisely the same time. The completion time for the iteration will be determined by the processor whose access finishes last. This processor waits for $t_w = (N-1)t_a$ time units. Since the rest of the processors must wait for the last processor to finish, all other processors also have $t_w = (N-1)t_a$. If the load is balanced, the sum in equation 1 degenerates to multiplication by N, from which we obtain

$$E = N\frac{t_a + t_p}{t_a + t_p + t_w} = \frac{N}{1 + ((N-1)t_a)/(t_a + t_p)} \tag{2}$$

and thus,

$$E = \frac{N}{1 + (N-1)\rho}\ , \tag{3}$$

where

$$\rho \equiv \frac{t_a}{t_a + t_p}$$

If the workload is asynchronous, and the processors are executing the same instructions, they will soon become "skewed" so that they attempt their global accesses at different times. Thus there will be no contention if $N-1$ processors have time to complete their accesses while the Nth is processing—that is, if $t_p \geq (N-1)t_a$, or, equivalently, when $\rho \leq 1/N$.

Hence when $N \leq 1/\rho$, iteration time is dominated by processing time and $E = N$. When $N > 1/\rho$, however, contention does occur, and the waiting time for each task is $(N-1)t_a - t_p$. In this case, iteration time is dominated by access time, and

$$E = \sum_N \frac{t_a + t_p}{t_a + t_p + (N-1)t_a - t_p} = \frac{1}{\rho}$$

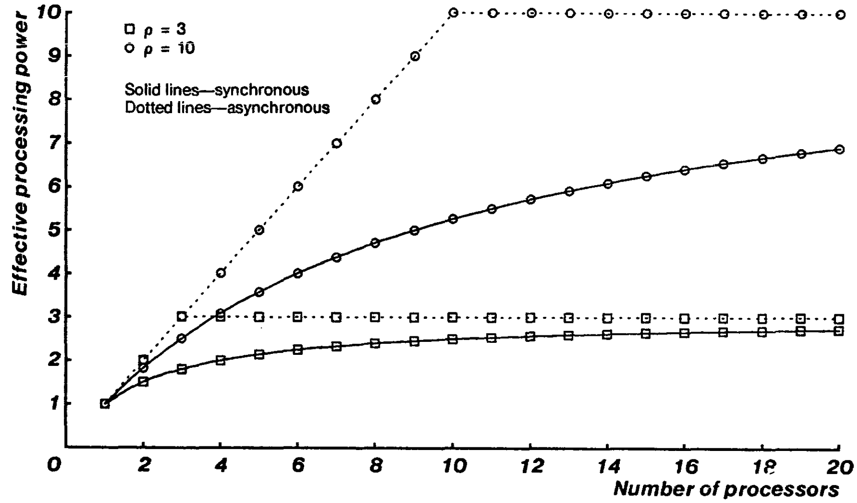
Thus in the asynchronous case,

$$E = \min[N,\ 1/\rho] \tag{4}$$

Figure C-2 gives a general idea of the shape of these curves for $\rho = 1/3$ and $\rho = 1/10$. Note that E is convex for the synchronous load and linear for the asynchronous load until the maximum is reached at $E = 1/\rho$.

The expressions given above for E have been derived under the assumption that global access and local processing are atomic operations and that each iteration has only one period of each kind. In practice, this is seldom true—an iteration frequently consists of several interleaved processing and access periods. However, the expressions we have given for E in the synchronous case and asynchronous cases, respectively, turn out to be lower and upper performance bounds for all arrangements of processing and access periods within an iteration.

Figure C-2 Effective Processing Power in the Synchronous and Asynchronous Cases



Intuitively, the worst case is encountered when synchronization is required after each consecutive processing-access pair. There is obviously no case worse than for all processes to request access to global memory at the same time and then wait for each other to finish in order to perform simultaneous accesses again. In such a system,

$$E = \sum_{N} \frac{\sum_{k} (t_{pk} + t_{ak})}{\sum_{k} (t_{pk} + t_{ak} + t_{wk})} \tag{5}$$

where k ranges over the processing-access pairs in the iteration. Because t_w depends linearly on t_a and N, equation (5) yields the same result as would only one atomic pair (t_p, t_a) having a duration equal to the length of an iteration. Hence this case is equivalent to the synchronous case described above.

We shall demonstrate that, for a given total processing time t_p and access time t_a in an iteration, upper-bound performance is achieved by an asynchronous algorithm with a single (t_p, t_a) pair.

CLAIM. The asynchronous case of N parallel processes, each with only one atomic pair (t_p, t_a) per iteration, yields the best performance of all cases with access and processing periods interleaved within the iteration.

PROOF. Given an iteration with a single (t_p, t_a) pair, suppose we rearrange it into M pairs (t_{pk}, t_{ak}) such that $\sum t_{pk} = t_p$ and $\sum t_{ak} = t_a$. For this case to be "better" than the original iteration, the total waiting time must be shorter than in the original case. If the first case had no waiting time, then it was optimal, and no further improvements are possible.

Now consider the case in which the original iteration has a positive waiting time. For this to happen, each process must finish its processing in less time than it takes for all the processes to complete their accesses. Hence $t_w = (N-1)t_a - t_p$ and the time to complete one iteration is $t_p + t_a + t_w = Nt_a$. Since this is the time required for all processes to access the shared data, improvement on this time is not possible, so any optimal rearrangement must complete one iteration for all processes in Nt_a.

$$\sum t_{ak} + \sum t_{pk} + \sum t_{wk} = Nt_a$$

$$t_a + t_p + \sum t_{wk} = Nt_a$$

$$\sum t_{wk} = (N-1)t_a - t_p = t_w$$

Therefore, no rearrangement of the original one-pair iterations can reduce total waiting time, so the one-pair asynchronous iterations are optimal as originally claimed.

Though simple, the model just presented is quite limited in its scope of application. It cannot predict the effects of changing processor speed or global access time. Its implicit assumption that ρ does not depend on N is at odds with the structure of many parallel algorithms. The rest of this appendix will set forth a more refined model capable of describing a wider range of architectures and applications. Section C.2 introduces the parameters of the model and shows how it can be used to derive upper and lower bounds for the performance of a parallel algorithm. Section C.3 shows how the model can be used to predict the effectiveness of increasing the speed of processors or global memory; Section C.4 uses the model to predict speedup.

C.2. The Refined Analytical Model

In this section, we augment the basic model by adding parameters for architectural factors such as processor and memory speed (Section C.2.1) and introducing decomposition functions (Section C.2.2) to describe how processing and access times change as the number of processors varies.

C.2.1. Architectural Refinements

Let us introduce three parameters that we can use to quantify architectural differences:

- p, processor speed relative to the speed of a *reference* processor, which we assign a speed of 1.
- q, global access speed. The speed of accessing global memory relative to the speed of a *reference* memory with a speed of 1.
- r, global access throughput. The value of this parameter is the number of processors that can simultaneously access the global memory without contention. This value may depend not only on hardware features (e.g., architecture of the interconnection network and existence of multiple global resources) but also on the algorithm decomposition and the partitioning of global data among multiple global resources.

Many performance metrics do not depend on the processing and access times themselves but merely on the ratio between them. Define the *processing-to-access ratio* of a workload as

$$x \equiv \frac{t_p}{t_a} \tag{6}$$

A list of all model parameters is presented in Table C-1. For the moment, assume that x is constant and independent of the number of processes in the parallel decomposition, although it will be seen later that this is only a special case. Since $\rho = 1/(1+x)$, substitution of (6) into (3) and (4) yields

$$E = \frac{N(1+x)}{N+x} \tag{7}$$

and

$$E = \min[N,\ 1+x] \tag{8}$$

Note that both expressions are 1 when $N=1$. As x grows large, both approach N, reflecting the declining influence of access time. Between the two extremes, the asynchronous E is always larger.

Influence of the Processor and Access Speed. When the processing speed is increased (or decreased), t_p is changed proportionately. Similarly, improving the global access speed will shorten t_a. The modified processing and access times will be

$$t'_p \equiv \frac{t_p}{p} \qquad\qquad t'_a \equiv \frac{t_a}{q}$$

and, consequently, the modified processing-to-access ratio is

$$x' \equiv \frac{t'_p}{t'_a} = x\,\frac{q}{p}$$

Table C-1 Parameters of the Model

E	*Effective processing power*, the "effective" number of processors working cooperatively on every iteration (page 412).
f_a	The *access decomposition function*, the ratio of global access time in a uniprocessor implementation to global access time by an individual process in a multiprocessor implementation (page 419).
f_p	The *processing decomposition function*, the ratio of processing time in a uniprocessor implementation to processing time for an individual process in a multiprocessor implementation (page 419).
N	The number of processors in the system (page 411).
p	Processor speed relative to the *reference* processor (page 416).
q	*Global access speed*, the speed at which global memory can be accessed, relative to the speed of a *reference* memory (page 416).
r	*Global access throughput*, the number of processors that can simultaneously access a common resource without contention (page 416).
S	*Speedup*, the ratio of the reference iteration time in a uniprocessor implementation to the iteration time of an individual process in a parallel implementation (page 423).
t_a	*Access time* for a subprocess within a cycle (page 412).
t'_a	*Modified access time*, the time it takes to make t_a accesses to a memory of speed q (page 416).
T_a	Access time for a uniprocessor implementation within a cycle (page 419).
t_c	*Cycle time*, the sum of the iteration time (t_p+t_a) and the waiting time t_w (page 420).
t_i	*Iteration time*, the sum of processing and access time for a single process in an iteration (page 423).
T_i	*Iteration time* for the uniprocessor implementation (page 423).
t_p	*Processing time* for a subprocess within a cycle (page 412).
t'_p	*Modified processing time*, the time it takes a processor of speed *ps* to perform t_p units of work (page 416).
T_p	Processing time for a uniprocessor implementation within a cycle (page 419).
t_w	*Waiting time* due to contention for a subprocess within a cycle (page 412).
x	*Processing-to-access ratio* for a subprocess, equal to t_p / t_a (page 416).
x'	*Modified processing-to-access ratio*, defined as t'_p / t'_a (page 416).
X	*Processing-to-access ratio* for a uniprocessor implementation, equal to T_p / T_a (page 419).
δ	Average utilization of an individual processor with a p of 1 (page 422).
μ_{qr}	*q / r-efficiency*, the (theoretical) percentage decrease in cycle time that would occur if q and r were infinite (page 421).
μ_p	*p-efficiency*, the (theoretical) percentage decrease in cycle time that would occur if p were infinite (page 421).
ρ	The fraction of time a processor spends performing global accesses, $t_a / (t_a + t_p)$ (page 413).

Substitution of x' for x in (7) and (8) gives

$$E = \frac{N(p+qx)}{Np+qx} \tag{9}$$

and

$$E = \min\,[N,\ 1+qx/p]$$

In the synchronous case, when $N > 1$, increasing q increases E; increasing p leads to a decrease in E, however, as access time grows more significant, making it more difficult to use the increased processing power. The same effects are present in the asynchronous case after contention begins to occur.

Influence of the Connection Throughput. To investigate interconnection networks that allow more than one processor to access data simultaneously without degradation, recall from (2) that, assuming a balanced load,

$$E = N\frac{t_a+t_p}{t_a+t_p+t_w}$$

The waiting time of a process obviously depends on r, the number of processes that can simultaneously access the data without contention. If parallel access is possible, r measures this parallelism. Changing r affects the system only insofar as it affects waiting time; the waiting time for a process is proportional to the integer $\lfloor N/r \rfloor$. (If there are seven processes, for example, and r is 2, then the seventh process must wait for three sets of two processes to complete their accesses; hence the waiting time is proportional to 3.) To keep the performance-prediction functions continuous, let us approximate the waiting time of a parallel process by defining t_w to be proportional to N/r:

$$t_w = \left[\frac{N}{r} - 1\right] t_a \tag{10}$$

for the synchronous case, and

$$t_w = \max\left[0,\ \left[\frac{N}{r} - 1\right] t_a - t_p\right] \tag{11}$$

for the asynchronous case [Vrsalovic and Siewiorek 83]. (We have just replaced N by N/r in the equations of Section C.1.) Substituting the definitions of (10) and (11) and the result of (9) into the definition of E, we derive

$$E = rN\frac{p+qx}{Np+qrx} \tag{12}$$

$$E = \min\,[N, r\,(1 + qx/p)] \tag{13}$$

See Figure 11-11 for a comparison of changes to q versus changes to r.

It is interesting to predict the results of a "brute-force" application of a huge number of processors. A few conclusions can be drawn by analyzing the $\lim_{N\to\infty} E(N)$:

- The maximal E for synchronous systems is $E(\infty) = r(1 + qx/p)$. In the case of a large qx/p (i.e., a large amount of processing per iteration), a reasonable E could be obtained using a conservative architecture with r approximately equal to 1. Conversely, in the case of small qx/p (i.e., a small amount of processing per iteration), the only means of improving E is to use an architecture/algorithm combination with a larger r.
- For an asynchronous system, there is no gain at all by increasing N beyond $r(1 + qx/p)$ because E will not increase for an N greater than this.

C.2.2. Decomposition of an Algorithm into Processes

When an algorithm is partitioned for parallel processing, each of the processes has less work (access and/or processing) to do than the original uniprocessor algorithm. The amount of work may or may not fall proportional to the number of processors that are added. A process that participates in solving a Poisson partial differential equation (PDE) (Section 11.3.2), for example, makes a number of global accesses that are proportional to the *perimeter* of the grid on which it operates, and the size of the perimeter does not fall proportionate to the decrease in the area of the grid. This suggests that we incorporate two functions f_p and f_a into the model to describe how the processing and access time *per processor* changes as processors are added to the system. Let the uppercase letters T_p and T_a denote the processing and access times in a one-processor system and the lowercase letters t_p and t_a denote the times in the multiprocessor version. Now, define

$$f_p(N) \equiv \frac{T_p}{t_p(N)} \qquad\qquad f_a(N) \equiv \frac{T_a}{t_a(N)} \tag{14}$$

Unless $f_p = f_a$ (the "special case" referred to in Section C.2.1), the processing-to-access ratio x will change as N varies. Let X be defined as T_p/T_a; then

$$x(N) = \frac{t_p(N)}{t_a(N)} = \frac{T_p}{T_a}\,\frac{f_a}{f_p} = X\frac{f_a}{f_p} \tag{15}$$

The substitution of (15) into (12) and (13) yields the final expressions for E:

$$E = \frac{rN(pf_p + qxf_a)}{Npf_p + qrxf_a} \qquad \text{synchronous case} \tag{16}$$

$$E = \min\left[N, r\left[1 + \frac{qXf_a}{pf_p}\right]\right] \quad \text{asynchronous case} \tag{17}$$

An ordered pair ($f_p\,;\, f_a$) is said to define a *decomposition group*. Two of the more common decomposition groups are ($N\,;\, N$)—linear decrease in processing and access times—and ($N\,;\, \sqrt{N}$) (like the PDE referred to above). Section 11.3.2 explores decomposition groups in more detail.

C.3. The Effect of Architectural Changes

Now that the performance model has been extended to encompass processor and memory speed, we can characterize the situations when it is more cost-effective to increase processing power rather than improve memory performance or vice versa. Let us begin by considering how changes in p, q, and r affect t_p and t_a. From (14) we have $t_p = T_p/f_p$ and a similar expression for t_a, but increasing the processor speed to p will decrease t_p by $1/p$, so

$$t_p = \frac{T_p}{pf_p} \qquad\qquad t_a = \frac{T_a}{qf_a} \tag{18}$$

C.3.1. The Synchronous Case

Let us define the *cycle time* as the sum of the iteration time and the waiting time:

$$t_c \equiv t_p + t_a + t_w$$

and hence

$$t_c = t_p + t_a + \left[\frac{N}{r} - 1\right] t_a = \frac{T_p}{pf_p} + \frac{N\,T_a}{qrf_a}$$

Thus the cycle time is made up of two distinct components:

- a part proportional to the processing time

$$t_{cp} \equiv \frac{T_p}{pf_p}$$

- a part proportional to the access time

$$t_{ca} \equiv \frac{NT_a}{qrf_a}$$

Although cycle time will always decrease as p, q, or r is increased, there will be some situations in which the t_{cp} or t_{ca} component will be so small that even bringing it down to near zero via huge investments in hardware would improve performance very little. If p were infinite, then the t_{cp} component would be zero; and an infinite q or r would mean a t_{ca} of zero. Let

$$f_x \equiv \frac{f_p}{f_a}$$

and define *p-efficiency* μ_p and *q/r-efficiency* μ_{qr} as the theoretical percentage decrease in cycle time by making p, q, or r infinite:

$$\mu_p = \frac{t_{cp}}{t_{cp} + t_{ca}} = 1 / \left[1 + \frac{Nf_x p}{Xqr} \right] \times 100$$

$$\mu_{qr} = \frac{t_{ca}}{t_{cp} + t_{ca}} = 1 / \left[1 + \frac{Xqr}{Nf_x p} \right] \times 100$$

It is probably more cost-effective to concentrate on reducing the larger of μ_p or μ_{qr} by increasing p or by increasing both q and r.

C.3.2. The Asynchronous Case

In the asynchronous case, there are two different possibilities:

1. No waiting time:

$$t_c = t_p + t_a = \frac{T_p}{pf_p} + \frac{T_a}{qf_a}$$

The cycle time does not depend on r and the efficiency factors are

$$\mu_p = \frac{1}{\left[1 + \frac{pf_x}{qX} \right]}$$

and

$$\mu_{qr} = \frac{1}{\left[1 + \frac{qX}{pf_x} \right]}$$

2. Waiting time equal to $[(N/r) - 1]\ t_a - t_p$. This is the case where

$$\left[\frac{N}{r} - 1 \right] t_a > t_p$$

or

$$p > \frac{Xf_a q}{f_p[(N/r) - 1]} \tag{19}$$

In this case, the cycle time does not depend on p because global access by all N processors takes longer than processing anyway:

$$t_c = \frac{N}{r}\, t_a$$

or, using the notation defined in (18),

$$t_c = \frac{NT_a}{qrf_a}$$

There is obviously no benefit to increasing p beyond the value given by (19). The efficiency factors are $\mu_p = 0$ and $\mu_{qr} = 100$ for such an asynchronous case.

The design process inevitably involves trade-offs between cost and performance. It is impossible to say which cost/performance trade-offs should be made in a particular design, but the approach of balancing efficiency factors ($\mu_p = \mu_{qr}$) may be worthwhile to consider, at least in the initial design phases.

C.4. Speedup

Effective processing power, as defined in equation 1, can be viewed as the amount of computation a system can perform in $t_p + t_a$ time units. Let us define the *iteration time*

$$t_i \equiv t_p + t_a$$

In the *reference* uniprocessor system, $p = q = r = 1$. For this system, $E = 1$ since $t_i = t_c$ because of the absence of degradation in the uniprocessor case. In a multiprocessor system δ, the average utilization of an individual processor, is defined as

$$\delta \equiv \frac{E}{N} \tag{20}$$

Intuitively, we expect an N-processor system to have a higher E than a uniprocessor system but not necessarily N times as high, since the waiting time t_w may be nonzero. Usually, the amount of waiting increases with the number of processors; this is reflected in a decreasing δ.

While it is obviously desirable for a system designer to keep resource utilization high, the designer's main concern is to minimize the amount of time needed to complete an application—in other words, to speed it up as much as possible by using multiple processors. We will use the *speedup factor* S to measure the performance of a multiprocessor system compared with the reference uniprocessor system. Speedup has been defined in several different ways [Kurskal 83]. We will define it as a ratio of cycle times:

$$S(N) \equiv \frac{T_c}{t_c(N)}$$

It is useful to express S in a slightly different form to illustrate how it is influenced by the utilization δ and the decomposed iteration time t_i (note that $T_i \equiv T_p + T_a = T_c$).

$$S(N) = \delta \frac{T_i}{t_i} \tag{21}$$

Speedup is thus the ratio between the reference iteration time and the decomposed iteration time, slowed by the utilization δ. If the processor speed p or the memory speed q is changed from unity, T_i changes accordingly, so

$$T_i = \frac{T_p}{p} + \frac{T_a}{q} \tag{22}$$

$$t_i = t_a + t_p = \frac{T_p}{pf_p} + \frac{T_a}{qf_a} \tag{23}$$

Substituting the results of (22) and (23) into (21) yields

$$S(N) = \delta \frac{\dfrac{T_p}{p} + \dfrac{T_a}{q}}{\dfrac{T_p}{pf_p} + \dfrac{T_a}{qf_a}}$$

Using the definition of δ from equation (20) and substituting for E using (16), we derive

$$S = \frac{rf_af_p(p + qX)}{Npf_p + qrXf_a} \tag{24}$$

for the synchronous case, and substituting for E using (17),

$$S = \min\left[f_af_p \frac{p + qX}{pf_p + qXf_a}, rf_a \frac{p + qX}{Np}\right] \tag{25}$$

Notice that if $p = q = r = 1$ $f_a = f_p = N$ these equations reduce to (7) and (8). From these equations, we can draw this general conclusion: While effective processing power depends only on the ratio of the decomposition functions f_p and f_a, speedup also depends on their values. In practical terms, this means that it is possible to decompose an algorithm into a set of processes that exhibit high processor utilization but low speedup.

C.5. Calibrating the Model with Experimental Data

This section compares the predictions of the model with the results of three previously published experiments [Whiteside *et al.* 83, Whiteside *et al.* 82]. Because the original source code was not readily available, it was necessary to estimate the relationship between t_p and t_a. These values were estimated by attempting to fit the predicted curves to the measured curves, but it should be noted that the same choice of values produces a good fit for all the curves on each graph.

C.5.1. Processor-Speed Variations

This experiment illustrates the influence of processor speed on performance. The parallel workload was an implementation on the Cm* multiprocessor of a molecular-dynamics algorithm. It consists of a number of parallel processes, each calculating the binding energy between particles. After all calculations are done, the final result is saved as global data to be used in the next iteration. This description implies that local processing time for a decomposed process is variable and equal to

$$t_p = \frac{T_p}{N}$$

The global access time is fixed and equal to the time required for returning a final result to global memory. Therefore,

$$t_a = T_a$$

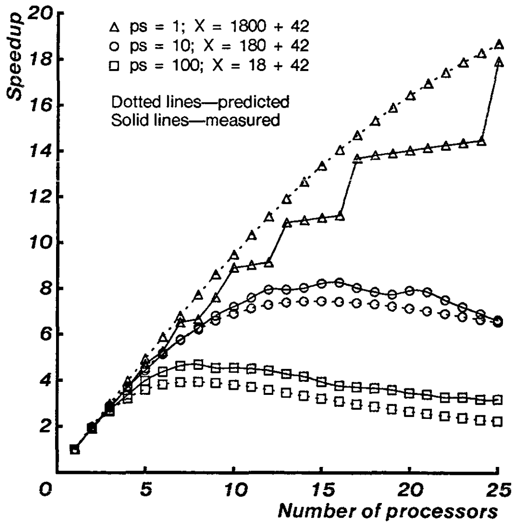
Consequently, the decomposition functions are

$$f_p = N \qquad f_a = 1$$

To ensure an atomic access for every process, a locking mechanism is used. Speedup was measured as a function of the processor speed. While the speed of the processor hardware could not be varied, faster processors could be simulated by replacing the slow LSI-11 floating-point calculations by "synthetic procedures," which took less time than the calculations and returned arbitrary results. (The goal was to study the effect of processor speed, so it did not matter that the results were incorrect.)

Figure C-3 shows both measured and theoretical results, assuming an estimated

Figure C-3 Molecular Dynamics Speedup, Dependency on Processor-Speed Variations



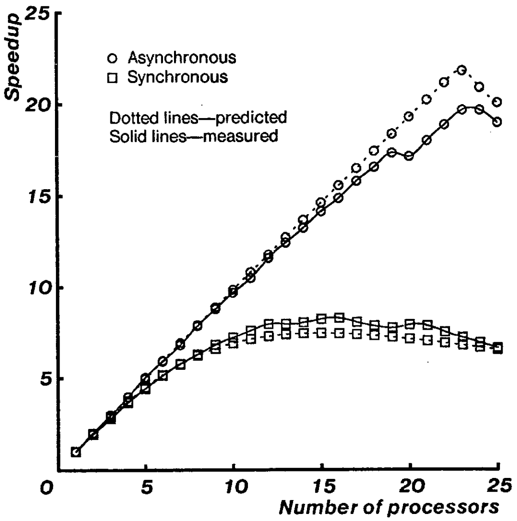
X. Although the synchronous case is supposed to be the worst case, the measured performance for slow processors is worse than the theoretical lower bound due to the lack of matching parallelism (see page 247). Some processors have much more work to do than others. Therefore, when processors are slow and t_p is dominant in the iteration, the idle processors induce a nonmonotonic performance curve and exhibit worse performance than theoretically predicted for a balanced load.

C.5.2. Speedup versus Synchronization

In the second experiment, speedup was measured for various degrees of synchronization between processes. Experimental and theoretical results for the two extreme cases (full synchronization and no synchronization) are given in Figure C-4. The *X* of the synchronous implementation is only about half as large as the *X* of its asynchronous counterpart because the omitted synchronization code consists mainly of access to common data.

The implementation for molecular-dynamic simulation analyzed here is a typical one-pair implementation (the kind considered in the proof of Section C.1), and for that reason the measured results are very close to the upper and lower bounds, respectively.

Figure C-4 Influence of Synchronization on Molecular Dynamics Calculations



C.5.3. Matrix Multiplication

The decomposition functions for matrix multiplication with the creation of local copies are as follows [Vrsalovic *et al.* 85]:

$$f_p = \frac{(MT_{iter} + T_{copy})N}{MT_{iter} + T_{copy}\sqrt{N}}$$

and

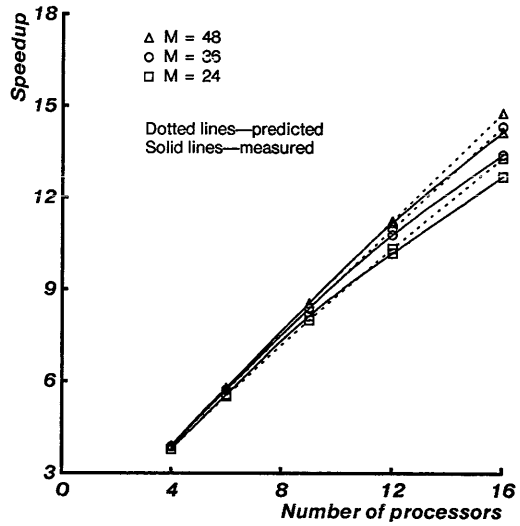
$$f_a = \sqrt{N}$$

where T_{iter} is the time needed to perform a single iteration (which calculates a single element of the result matrix) and T_{copy} is the additional *processing* time required to copy a single matrix element to local memory (including its share of the loop overhead). Let T_{acc} be the time it takes to perform one global access, exclusive of waiting time, and let

$$k_1 \equiv \frac{T_{iter}}{T_{acc}} \qquad k_2 \equiv \frac{T_{iter}}{T_{copy}} \tag{26}$$

express the relationship between access and processing speeds without reference to particular hardware technologies.

Figure C-5 Speedup of Matrix Multiplication



We ran the local-copies algorithm on Cm*. The constants k_1 and k_2 were measured at 1.694 and 3.678, respectively. Correlation of the experimental and predicted curves is quite good but falls off somewhat as more processors are added, largely because of the undecomposable constant overhead added to each process by the need to perform loop initialization and to read the clock. This overhead grows more significant as processors are added and the work per process declines. To try and factor out the effect of this overhead, which we called T_{init}, we solved the following system of equations

$T_{init} + 48^3 T_{iter} = 82.3$ ms. (measured processing time for $M = 48$ without local copies)

$T_{init} + 24^3 T_{iter} = 10.3$ ms. (measured processing time for $M = 24$ without local copies)

The results of $T_{iter} = 7.4 \times 10^{-4}$ and $T_{init} = 1.14 \times 10^{-2}$ "predicted" the execution time for $M = 36$ within 1 percent. From this we derived the revised decomposition functions:

$$t_p = T_{init} + \frac{M^3 T_{iter}}{N} + \frac{M^2 T_{copy}}{\sqrt{N}} \qquad t_a = \frac{M^2 T_{acc}}{N}$$

We can proceed as before to derive equations to predict speedup. These equations are graphed against the observed values in Figure C-5. The measured values are everywhere within 5 percent of the predicted values. The close correspondence deteriorates slightly for increasing values of N. One probable reason is that the initialization of the inner loop grows more significant as it gets shorter; for $M=24$ and $N=16$, the inner loop is executed only six times before it terminates.

C.6. Summary

We have defined a simple multiprocessor model, which was then enhanced to accommodate architectural parameters such as processors and memories of different speeds. The model has shown that improving processor or memory speed is effective only up to a point. The model bounds the worst performance that can be expected from a synchronous algorithm and predicts the best performance that can be achieved by an asynchronous one.

The performance of the multiprocessor implementation of an algorithm is frequently expressed in terms of speedup. Speedup is a function of the way an algorithm is decomposed. We have defined decomposition functions for the processing and access times, which tell how the per-process times change as the number of processors is increased. Algorithms can be divided into decomposition groups based on these functions—groups that display characteristic speedup curves for varying numbers of processors.

Appendix D
Smap: the Simple Microcode

Written during the spring of 1977, Smap was one of the first microcode systems for the Cm* Kmaps. It closely reflects the underlying Cm* hardware but can hardly be called an operating system, hence its inclusion in this appendix. No attempt was made in its design to provide protection or generality because its primary use was seen as a base for benchmark programs and diagnostics for the hardware.

The main function of the simple microcode is to allow each processor to associate any 4,096-byte physical page in the system with any of the pages in the immediate address space of the processor. A page in the system is identified by its cluster number, Cm within the cluster, and high-order 6 bits of its base address. A page in the immediate address space of the processor is identified by the address space of the processor (user / kernel) and the high-order 4 bits of the processor address; it is referred to as a *window*. Once a window has been associated with a physical page by writing the mapping tables in the Slocal and Kmap, all references to the window are mapped to the physical page, with the bottom 12 bits of the processor's address serving as the offset into the page. In binding a window to a physical page in its local memory, a processor may choose to have references to the page proceed directly through the Slocal or mapped via the Kmap.

The Slocal mapping table and the mapping table maintained by the Kmap on behalf of a processor are both addressable from the processor. A processor may therefore make any page in the system addressable to itself simply by writing into the appropriate entries in these tables. The microcode does not provide any protection.

The remainder of the microcode is made up of a variety of synchronization operations that are useful in making indivisible access to shared data and in coordinating the actions of processors working on a common task. There is one operation to indivisibly increment an arbitrary word in memory and four operations to indivisibly decrement an arbitrary word in memory. The decrement operations vary along two dimensions—conditionality and manner of notification. Unconditional decrements are performed regardless of the old contents of the target location; conditional decrements are not performed if the old value of the operation is zero. A processor can select from two methods of notification, synchronous and asynchronous. In the synchronous method, the processor examines a result location that is set after the operation completes; in the asynchronous method, it is interrupted when the result of the operation is zero.

Smap contains simple facilities to aid processors in handling errors. When an error occurs during a microcode operation, the Kmap does not attempt to recover from the error. Instead, it collects as much information about the error as it can and stores this information in a place that is accessible to the invoking processor; it then interrupts the processor to signal the error. Recovery from the error is left entirely up to the invoking processor.

There are certain quirks in the implementation of Smap that prevent it from functioning in a robust way when large amounts of contention are generated at a particular Slocal. The problem has to do with the way in which the microcode handles a busy Slocal condition when it tries to reference the memory of a Cm. The solution used simply waits a fixed amount of time and then retries the reference. Although this is simple to implement and turns out to be the most efficient solution as long as it works, it runs into problems when large numbers of processors are continually referencing a given Cm. The amount of time a given context waits for a busy Slocal to become free is nondeterministic and may be large enough to exceed any reasonable time-out period for components that are waiting for the reference to complete. Because the hardware of Cm* provides no way to recognize return requests from operations that have already been timed out, arbitrary damage may result when such a request does return.

John Ousterhout wrote the initial single-cluster version of Smap; Andy Bechtolsheim added intercluster references, and Pradeep Sindhu undertook a major revision to incorporate synchronization operations and better error reporting.

Acknowledgment. This appendix was originally written by Pradeep Sindhu for [Jones and Gehringer 80].

Bibliography

[Ackerman 78]
W. B. Ackerman. "A structure processing facility for data flow computers." In G. Jack Lipovski [editor], *Proceedings of the 1978 International Conference on Parallel Processing.* IEEE Computer Society, August 1978, 166–172.

[Adams and Siegel 82]
G. B. Adams and H. J. Siegel. "On the number of permutations performable by the augmented data manipulator." *IEEE Transactions on Computers* C-31(4): 270–277 (April 1982).

[Anderson and Jensen 75]
G. A. Anderson and E. D. Jensen. "Computer interconnection structures: taxonomy, characteristics and examples." *Computing Surveys* 7(4): (December 1975), 197–213.

[Baker 78]
Henry G. Baker, Jr. "List processing in real time on a serial computer." *Communications of the ACM* 21(4): 280–294 (April 1978).

[Baskett and Smith 76]
F. Baskett and A. J. Smith. "Interference in multiprocessor computer systems and interleaved memory." *Communications of the ACM* 19(6): 327–334 (June 1976).

[Baudet 78]
G. M. Baudet. "The design and analysis of algorithms for asynchronous multiprocessors." Ph.D. diss., Carnegie-Mellon University, April 1978.

[Bell and Newell 71]
C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* New York: McGraw-Hill, 1971.

[Bell *et al.* 72]
C. G. Bell, J. L. Eggert, J. Grason, and P. Williams. "The description and the use of register transfer modules (RTM's)." *IEEE Transactions on Computers* C-21(5): 495–500 (May 1972).

[Bhandarkar 75]
D. P. Bhandarkar. "Analysis of memory interference in multiprocessors." *IEEE Transactions on Computers* C-24(9): 897–908 (September 1975).

[Bhuyan and Agrawal 84]
Laxmi N. Bhuyan and Dharma P. Agrawal. "Generalized hypercube and hyperbus structures for a computer network." *IEEE Transactions on Computers* C-33(4): 323–333 (April 1984).

[Brownston *et al.* 85]
Lee Brownston, Robert Farrell, Elaine Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Reading, MA: Addison-Wesley, 1985.

[Carey 80]
Michael J. Carey. "Parallel processing for power system transient simulation: a case study." Master's thesis, Carnegie-Mellon University, December 1980.

[Cerf 72]
V. Cerf. *Multiprocessors, semaphores and a graph model of computation.* Technical Report 7223, Computer Science Department, UCLA, April 1972.

[Chansler 82]
Robert J. Chansler, Jr. "Coupling in systems with many processors." Ph.D. diss., Carnegie-Mellon University, August 1982. Also available as *Efficient Use of Systems with Many Processors.* Ann Arbor, Mich.: UMI Research Press, 1982.

[Chen 75]
T. Chen. "Overlap and pipeline processing." In H. Stone (editor), *Introduction to Computer Architecture*. Chicago: Science Research Associates, Inc., 1975, 375–431.

[CMU 79]
Robert J. Chansler, Jr., Ivor Durham, Wayne Gramlich, Anita Jones, Karsten Schwans, and Steven Vegdahl. StarOS design manual. Unpublished report from StarOS group, Carnegie-Mellon University, 1979.

[Dannenberg 81]
Roger B. Dannenberg. *Ampl: design, implementation, and evaluation of a multiprocessing language*. Technical Report, Department of Computer Science, Carnegie-Mellon University, March 1981.

[Davies 78]
D. Davies and J. Wakerly. "Synchronization and matching in redundant systems." *IEEE Transactions on Computers* C-27(6): 531–539 (June 1978).

[Deminet 82]
Jaroslaw Deminet. "Experience with multiprocessor algorithms." *IEEE Transactions on Computers* C-31(4): 278–287 (April 1982).

[Dijkstra 78]
Edsger W. Dijkstra, *et al.* "On-the-fly garbage collection: an exercise in cooperation." *Communications of the ACM* 21(11): 966–975 (November 1978).

[Dugan *et al.* 79]
R. C. Dugan, I. Durham, and S. N. Talukdar. "An algorithm for power system simulation by parallel processing." In *Text of abstracts, Summer Power Meeting.* IEEE Power Engineering Society, 1979.

[Durham *et al.* 79]
I. Durham, R. C. Dugan, A. K. Jones, and S. N. Talukdar. "Power system simulation on a multiprocessor." In *Text of abstracts, Summer Power Meeting.* IEEE Power Engineering Society, 1979.

[Feng 81]
T. Feng. "A survey of interconnection networks." *IEEE Computer* 14(12): 12–27 (December 1981).

[Forman 79]
P. Forman and K. Moses. "SIFT: multiprocessor architecture for software implemented fault tolerance flight control and avionics computers." In *Proceedings of the Third Digital Avionics Systems Conference*: 325–329 (November 1979).

[Fuller *et al.* 73]
S. H. Fuller, D. P. Siewiorek, and R. J. Swan. "Computer modules: an architecture for large digital modules." In *Proceedings of the First Annual Symposium on Computer Architecture* (reprinted as *Computer Architecture News* 2[4]). ACM / SIGARCH, University of Florida, Gainesville, December 1973, 231–236.

[Fuller *et al.* 75]
S. H. Fuller, D. P. Siewiorek, V. Lesser, W. Brantley, and R. J. Swan. Proposal for multiple processor systems based on DEC's Western Digital microcomputers. Unpublished report, Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, January 1975.

[Fuller *et al.* 77]
Samuel H. Fuller, Anita K. Jones, and Ivor Durham [eds.]. *Cm* review,* June 1977. Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Fuller *et al.* 78]
S. H. Fuller, J. K. Ousterhout, L. Raskin, P. Rubinfeld, P. S. Sindhu, and R. J. Swan. "Multi-microprocessors: an overview and working example." *Proceedings of the IEEE* 66(2): 216–228 (February 1978).

[Gehringer *et al.* 82]
Edward F. Gehringer, Anita K. Jones, and Zary Z. Segall. "The Cm* testbed." *IEEE Computer* 15(10): 40–53 (October 1982).

[Gehringer and Chansier 82]
Edward F. Gehringer and Robert J. Chansler, Jr. *StarOS user and system structure manual*. Technical Report, Department of Computer Science, Carnegie-Mellon University, July 1982.

[Goldberg 80]
J. Goldberg, C. Weinstock, M. Green, W. Kautz, L. Lamport, and P. Melliar-Smith. *Development and evaluation of a SIFT computer: SIFT operating system.* Interim Technical Report 2, SRI International, April 1980.

[Goldberg 81]
J. Goldberg. "The SIFT computer and its development." In *Proceedings of the Fourth Digital Avionics Systems Conference*, SRI International: November 1981.

[Gostelow 71]
K. P. Gostelow. "Flow of control, resource allocation, and the proper termination of programs." Ph.D. diss., University of California, Los Angeles, December 1971.

[Gregoretti and Segall 84]
Francesco Gregoretti and Zary Segall. "Analysis and evaluation of VLSI design rule checking implementation in a multiprocessor." In *Proceedings of the 13th International Conference on Parallel Processing.* IEEE Computer Society, August 21–23, 1984, 7–14.

[Habermann 76]
A. Nico Habermann. *An Introduction to Operating System Design.* Chicago: Science Research Associates, 1976.

[Habermann *et al.* 76]
A. Nico Habermann, Lawrence Flon, and Lee Cooprider. "Modularization and hierarchy in a family of operating systems." *Communications of the ACM* 19: 266–272 (May 1976).

[Haynes *et al.* 82]
Leonard S. Haynes, Richard L. Lau, Daniel P. Siewiorek, and David W. Mizell. "A survey of highly parallel computing." *IEEE Computer* 15(1): 9–24 (January 1982).

[Hibbard 80]
Peter G. Hibbard. Personal communication 1980.

[Hoare 78]
C.A.R. Hoare. "Communicating sequential processes." *Communications of the ACM* 21(8): 666–677 (August 1978).

[Hon 83]
Robert W. Hon. "The hierarchical analysis of VLSI designs." Ph.D. diss., Carnegie-Mellon University, December 1983. Published as Technical Report CMU-CS-83-170.

[Jones and Ardö 82]
Anita K. Jones and Anders Ardö. "Comparative efficiency of different implementations of the Ada rendezvous." In *Proceedings of the AdaTEC Conference on Ada.* October 1982, 212–223.

[Jones and Gehringer 80]
Anita K. Jones and Edward F. Gehringer [eds.]. *The Cm* multiprocessor project: a research review*. Technical Report CMU-CS-80-131, Department of Computer Science, Carnegie-Mellon University, July 1980.

[Jones and Schwans 79]
Anita K. Jones and Karsten Schwans. "Task forces: distributed software for solving problems of substantial size." In *Proceedings of the Fourth International Conference on Software Engineering.* ACM / SIGSOFT, Munich, September 14–16, 1979.

[Jones and Schwans 80]
A. K. Jones and K. Schwans. The Task language specification. Unpublished report, Department of Computer Science, Carnegie-Mellon University, July 1980.

[Keedy 79]
J. Leslie Keedy. *A comparison of two process structuring models*. MONADS Report No. 4, Department of Computer Science, Monash University, 1979.

[Kinney and Arnold 78]
L. L. Kinney and R. G. Arnold. "Analysis of a multiprocessor system with a shared bus." In *Proceedings of the 5th Annual Symposium on Computer Architecture.* Palo Alto, CA, April 3–5, 1978, 89–95.

[Kleinrock 75]
Leonard Kleinrock. *Queueing Systems.* Vol. 2: *Computer Applications.* New York: John Wiley & Sons, 1975.

[Kong 82]
Thomas H. Kong. "Measuring time for performance evaluation of multiprocessor systems." Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, November 1982.

[Kong *et al.* 83]
Thomas H. Kong, Alfred Z. Spector, and Daniel P. Siewiorek. Measuring time in multiprocessor systems. Unpublished report, Department of Computer Science, Carnegie-Mellon University, July 1983.

[Kruskal 83]
C. P. Kruskal. "Searching, merging, and sorting in parallel computation." *IEEE Transactions on Computers* C-32(10): 942–946 (October 1983).

[Kung and Song 77]
H. T. Kung and S. W. Song. "An efficient parallel garbage collection system and its correctness proof." In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, October 1977, 120–131. (Also available as a Carnegie-Mellon University Computer Science Department Technical Report, September 1977.).

[Lampson 80]
Butler W. Lampson and David D. Redell. "Experience with processes and monitors in Mesa." *Communications of the ACM* 23(2): 105–117 (February 1980).

[Lampson and Sproull 79]
B. W. Lampson and R. F. Sproull. "An open operating system for a single-user machine." In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM / SIGOPS, Pacific Grove, California, December 10–12, 1979, 98–105.

[Lane 84]
Tom Lane. A parallel algorithm for VLSI design rule checking. Unpublished report, September 3, 1984.

[Lauer and Needham 78]
H. C. Lauer and R. M. Needham. "On the duality of operating system structures." In *Proceedings of the Second International Symposium on Operating Systems.* IRIA, 1978. Reprinted in *Operating Systems Review* 13(2): 3–19 (April 1979).

[Lawrie 75]
D. H. Lawrie. "Access and alignment of data in an array processor." *IEEE Transactions on Computers* C-24(12): 1145–1155 (December 1975).

[Lee 80]
R. B. Lee. "Performance characteristics of parallel processor organizations." Ph.D. diss., Stanford University, May 1980.

[Levin 77]
Roy Levin. "Program structures for exceptional condition handling." Ph.D. diss., Carnegie-Mellon University, 1977.

[Levin *et al.* 75]
R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. "Policy / mechanism separation in Hydra." In *Proceedings of the Fifth Symposium on Operating Systems Principles.*

(Reprinted as *ACM Operating Systems Review*, 9[5]). ACM / SIGOPS, University of Texas at Austin, November 19–21, 1975, 132–140.

[Levy 74]

H. Levy. *Simulation of two algorithms on computer modules*. Technical report, Department of Computer Science, Carnegie-Mellon University, May 1974.

[Little *et al.* 63]

J.D.C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. "An algorithm for the traveling salesman problem." *Operations Research* 11 (November–December 1963), 972–989.

[Lowerre 76]

Bruce Lowerre. "The Harpy speech recognition system." Ph.D. diss., Carnegie-Mellon University, April 1976.

[Marsan and Gerla 82]

M. A. Marsan and M. Gerla. "Markov models for multiple-bus multiprocessor systems." *IEEE Transactions on Computers* C-31(3): 239–248 (March 1982).

[Marsan and Gregoretti 81]

M. A. Marsan and F. Gregoretti. "Memory interference models for a multimicroprocessor model with shared bus and single external common memory." *Euromicro J.* (February 1981) 124–133.

[McConnel 81]

S. McConnel and D. Siewiorek. "Synchronization and voting." *IEEE Transactions on Computers* C-30(2): 161–164 (February 1981).

[McConnel *et al.* 79a]

S. R. McConnel, D. P. Siewiorek, and M. M. Tsao. "The measurement and analysis of transient errors in digital computer systems." In *FTCS9*, IEEE Computer Society, 1979, 67–70.

[McConnel *et al.* 79b]

S. R. McConnel, D. P. Siewiorek, and M. M. Tsao. *Transient error data analysis*. Technical Report CMU-CS-79-121, Department of Computer Science, Carnegie-Mellon University, May 1979.

[Mehrotra and Gehringer 85]

Ravi Mehrotra and Edward F. Gehringer. "Superlinear speedup through randomized algorithms." In *Proceedings of the 14th International Conference on Parallel Processing.* St. Charles, IL, August 20–23, 1985, 291–300.

[Metropolis *et al.* 53]

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. "Equation of state calculations by fast computing machines." *J. Chem. Phys.* 21: 1087–1092 (June 1953).

[Michalopoulos 82]

D. Michalopoulos. "Uniquely maneuverable fighter plane to use digital processors." *IEEE Computer* (October 1982) 120–121.

[Mohan 82]

Joseph Mohan. *A study in parallel computation—the traveling salesman problem*. Technical Report CMU-CS-82-136, Department of Computer Science, Carnegie-Mellon University, August 1982.

[Mohan 83]

Joseph Mohan. "Experience with two parallel programs solving the traveling salesman problem." In *Proceedings of the 12th International Conference on Parallel Processing.* IEEE Computer Society, August 23–26, 1983, 191–193.

[Mohan 84]

Joseph Mohan. "Performance of parallel programs: model and analyses." Ph.D. diss., Department of Computer Science, Carnegie-Mellon University, July 1984.

[Mohan *et al.* 85]
Joseph Mohan, Anita K. Jones, Edward F. Gehringer, and Zary Z. Segall. "Granularity of parallel computation." In Edmond L. Gallizzi *et al.* [editors], *Proceedings of the Eighteenth Hawaii International Conference on System Sciences,* Vol. 1, January 1985, 249–256.

[Nelson 81]
Bruce J. Nelson. "Remote procedure call." Ph.D. diss., Department of Computer Science, Carnegie-Mellon University, May 1981.

[Organick 72]
Elliott I. Organick. *The Multics System: An Examination of Its Structure.* Cambridge, MA: The MIT Press, 1972.

[Organick 83]
Elliot I. Organick. *A Programmer's View of the Intel 432 System.* New York: McGraw-Hill, 1983.

[Ostlund *et al.* 82a]
N. S. Ostlund, P. G. Hibbard, and R. A. Whiteside. "A case study in the application of a tightly-coupled multiprocessor to scientific computations." In B. Alder, S. Fernbach, and M. Rotenberg [editors], *Parallel Computations.* New York: Academic Press, 1982, 315–364.

[Ostlund *et al.* 82b]
Neil S. Ostlund, Robert A. Whiteside, and Peter G. Hibbard. "Computational chemistry and computer science." *Journal of Physical Chemistry* 86(12): 2190–2197 (June 10, 1982).

[Ousterhout 80]
John K. Ousterhout. "Partitioning and cooperation in a distributed multiprocessor operating sytem: MEDUSA." Ph.D. diss., Carnegie-Mellon University, April 1980.

[Ousterhout 82]
John K. Ousterhout. "Scheduling techniques for concurrent systems." In *Proceedings of the Third International Conference on Distributed Computer Systems.* October 1982, 22–31.

[Parnas 72]
D. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM* 15: 1053–1058 (December 1972).

[Patterson and Sequin 81]
David Patterson and Carlo Sequin. "RISC I: a reduced instruction set VLSI computer." In *Proceedings of the Eighth Annual Symposium on Computer Architecture.* May 12–14, 1981, 443–457.

[Preparata and Vuillemin 81]
F. P. Preparata and J. Vuillemin. "The cube-connected cycles: a versatile network for parallel computation." *Communications of the ACM* 24(5): 300–309 (May 1981).

[Raskin 78]
Levy Raskin. "Performance evaluation of multiple processor systems." Ph.D. diss., Carnegie-Mellon University, August 1978. Published as Technical Report CMU-CS-78-141.

[Reilly 83]
M. H. Reilly. "A resource management system for multiple processors with support for automatic experiment supervision." Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, March 1983.

[Robinson 82]
John T. Robinson. "Design of concurrency controls for transaction processing systems." Ph.D. diss., Carnegie-Mellon University, April 1982. Published as Technical Report CMU-CS-82-114.

[Rychener 80]
Michael D. Rychener. *OPS3 production system language tutorial and reference manual.* Unpublished Report, Department of Computer Science, Carnegie-Mellon University, March 1980.

[Scelza 79]
D. A. Scelza. An auto-diagnostic program for Cm*. Department of Computer Science, Carnegie-Mellon University, internal report, April 1979.

[Scelza *et al.* 81]
Donald A. Scelza, John K. Ousterhout, and Rudy Nedved. The MEDUSA task force linker user documentation.Department of Computer Science, Carnegie-Mellon University, internal report, September 1981.

[Schwan and Jones 86]
Karsten Schwan and Anita K. Jones. *IEEE Software* 3(3): 60–70 (May 1986).

[Schwans 82]
Karsten Schwans. "Tailoring software for multiple processor systems." Ph.D. diss., Carnegie-Mellon University, October 1982.

[Sedgewick 78]
R. Sedgewick. "Implementing quicksort programs." *Communications of the ACM* 21(10): 847–857 (October 1978).

[Segall 83]
Z. Segall, A. Singh, R. Snodgrass, A. Jones, and D. Siewiorek. "An integrated instrumentation environment for multiprocessors." *IEEE Transactions on Computers* C-32(1): 4–14 (January 1983).

[Siegel 85]
Howard Jay Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington, MA: Lexington Books, 1985.

[Siewiorek *et al.* 78a]
D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel, and M. Tsao. "A case study of C.mmp, Cm*, and C.vmp: part I—experiences with fault tolerance in multiprocessor systems." *Proceedings of the IEEE* 66(10): 1178-1199 (October 1978).

[Siewiorek *et al.* 78b]
D. P. Siewiorek, V. Kini, R. Joobbani, and H. Bellis. "A case study of C.mmp, Cm*, and C.vmp: part II—predicting and calibrating reliability of multiprocessor systems." *Proceedings of the IEEE* 66(10): 1200–1220 (October 1978).

[Siewiorek *et al.* 82]
Daniel P. Siewiorek, C. Gordon Bell, and Allen P. Newell. *Computer Structures: Principles and Examples.* New York: McGraw-Hill, 1982.

[Sindhu 84]
Pradeep S. Sindhu. "Distribution and reliability in a multiprocessor operating system." Ph.D. diss., Carnegie-Mellon University, April 1984. Published as Technical Report CMU-CS-84-125.

[Sindhu and Singh 83]
Pradeep Sindhu and Ajay Singh. Performance evaluation of message mechanisms. Unpublished report, Carnegie-Mellon University, April 1983.

[Singh 81]
A. Singh. "Pegasus: A workload generator for multiprocessors." Master's thesis, Carnegie-Mellon University, Department of Electrical Engineering, 1981.

[Singh *et al.* 82]
Ajay Singh and Zary Segall. "Synthetic workload generation for experimentation with multiprocessors." In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* October 1982, 778–785.

[Snodgrass 82]
Richard Snodgrass. "Monitoring Distributed Systems." Ph.D. diss., Carnegie-Mellon University, 1982.

[Stone 75]
Harold S. Stone. *Introduction to Computer Architecture.* Chicago: Science Research Associates, 1975, chapter 11.

[Stonebraker 76]

M. Stonebraker, E. Wong, P. Kreps, and G. Held. "The design and implementation of INGRES." *ACM TODS* 1(3): 189-222 (September 1976).

[Swan 78]

Richard J. Swan. "The switching structure and addressing architecture of an extensible multiprocessor, Cm*." Ph.D. diss., Carnegie-Mellon University, August 1978.

[Swan *et al.* 76]

R. J. Swan, S. H. Fuller, and D. P. Siewiorek. "The structure and architecture of Cm*: a modular multi-microprocessor." In *Computer Science Research Review*. Carnegie-Mellon University, Department of Computer Science, 1975–76, 25–47.

[Swan *et al.* 77]

Richard J. Swan, Samuel H. Fuller, and D. P. Siewiorek. "Cm*: A modular, multi-microprocessor." In *Proceedings of the National Computer Conference.* AFIPS, 1977, 637–644.

[Talukdar 79]

Sarosh N. Talukdar. "On using MIMD-type multiprocessors—some performance bounds, metrics, and algorithmic issues." In *Proceedings of the 10th Pittsburgh Modeling and Simulation Conference* (1979) 1167–1173.

[Talukdar *et al.* 81]

S. N. Talukdar, M. J. Carey, and S. S. Pyo. "Multiprocessors for power system problems." *Joho-Shori (Information Processing Society of Japan)* 22(12): (December 1981).

[Talukdar *et al.* 82]

S. N. Talukdar, M. J. Carey, and S. S. Pyo. "Multiprocessors for power systems—some programming and research issues." In *Proceedings of the IFAC Symposium on Digital Control.* IFAC, January 1982.

[Thurber 74]

K. J. Thurber. "Interconnection networks—a survey and assessment." In *AFIPS Conference Proceedings, National Computer Conference* (1974) 909–919.

[Tsao 78]

M. M. Tsao. "A study of transient errors on Cm*." Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, December 1978.

[Ullman 80]

J. D. Ullman. *Principles of Database Systems.* Potomac, MD: Computer Science Press, 1980.

[Verlet 67]

L. Verlet. "Computer experiments on classical fluids: Part 1—thermodynamic properties of Lennard-Jones molecules." *Physical Review* 159: 98–103 (1967).

[Vrsalovic 83]

D. Vrsalovic and D. P. Siewiorek. "Performance analysis of multiprocessor based control systems." In *Proceedings of the Real-Time Systems Symposium* (December 1983) 73–78.

[Vrsalovic *et al.* 84a]

Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer. "Performance prediction for multiprocessor systems." In *Proceedings of the 13th International Conference on Parallel Processing.* IEEE Computer Society, August 21–23, 1984, 139–146.

[Vrsalovic *et al.* 84b]

Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer. *Performance prediction and calibration for a class of multiprocessor systems.* Technical Report, Department of Computer Science, Carnegie-Mellon University, 1984.

[Vrsalovic *et al.* 85]
Dalibor Vrsalovic, Edward F. Gehringer, Zary Z. Segall, and Daniel P. Siewiorek. "The influence of parallel decomposition strategies on the performance of multiprocessor systems." In *Proceedings of the 12th Annual International Symposium on Computer Architecture.* Boston, June 17–19, 1985, 396–405.

[Weide 78]
Bruce W. Weide. "Statistical methods in algorithm design and analysis." Ph.D. diss., Carnegie-Mellon University, August 1978. Published as Technical Report CMU-CS-78-142.

[Whiteside *et al.* 82]
Robert A. Whiteside, Peter G. Hibbard, and Neil S. Ostlund. "Systolic algorithms for Monte Carlo simulations." In *Proceedings of the Third International Conference on Distributed Computing Systems.* Miami / Ft. Lauderdale, October 18–22, 1982, 800–804.

[Whiteside *et al.* 83]
Robert A. Whiteside, Peter G. Hibbard, and Neil S. Ostlund. "Conventional" and systolic parallel algorithms for Monte Carlo simulations of molecular motions. Unpublished report, Submitted to *ACM Transactions on Computer Systems*, 1983.

[Wilkes 77]
M. V. Wilkes. "Beyond today's computers." In *Information Processing '77 (IFIP 1977 Congress).* North Holland Publishing Company, 1977, 1–5.

[Wilkes and Needham 79]
M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System.* New York: Elsevier North Holland, 1979.

[Wilson *et al.* 83]
A. W. Wilson, D. P. Siewiorek, and Z. Z. Segall. "Evaluation of multiprocessor interconnect structures with the Cm* testbed." In *Proceedings of the 1983 International Conference on Parallel Processing* (1983).

[Wirth 77]
N. Wirth. "Design and implementation of Modula." *Software—Practice and Experience* 7(1): 67–84 (1977).

[Wulf *et al* 70]
W. Wulf, *et al. BLISS-11 Programmer's Manual.* Digital Equipment Corporation, 1970.

[Wulf 72]
William A. Wulf. "C.mmp: a multi-mini processor." In *Fall Joint Computer Conference, Proceedings 41* (II). AFIPS, 1972, 765–777.

[Wulf *et al.* 74]
W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. "HYDRA: The kernel of a multiprocessor operating system." *Communications of the ACM* 17(6): 337–345 (June 1974).

[Wulf *et al.* 81]
William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA / C.mmp: An Experimental Computer System.* New York: McGraw-Hill, 1981.

[York *et al.* 83]
Gary York, Daniel P. Siewiorek, and Zary Z. Segall. *Software voting in asynchronous NMR computer structures*. Technical Report CMU-CS-83-128, Department of Computer Science, Carnegie-Mellon University, May 6, 1983.

Index

Ordering Information

To order additional copies of this book and related titles, fill in and mail this form or call the toll-free telephone number below. Orders under $50 must be prepaid by check or credit card; postage and handling are free on prepaid orders.

Digital Press/Order Processing
Digital Equipment Corporation
12A Esquire Road
Billerica, MA 01862

Qty.	Author/Title	Order No.	Price*	Total
	Gehringer et al. *Parallel Processing: Cm**	EY-6709E-DP	$40.00	
	Siewiorek/Swarz: *Reliable System Design*	EY-AX016-DP	45.00	
	Levy: *Capability-Based Systems*	EY-00011-DP	28.00	

Total ______

Add state sales tax ______

Total remitted ______

Method of Payment

____ Check included (Make checks payable to Digital Equipment Corporation)

____ Purchase order (Please attach)

____ MasterCard/Visa

Charge Card Acc't No. ______________

Expiration Date ______________

Authorized Signature ______________

Name ______________ Phone ______________

Address ______________

City ______________ State ______________ Zip ______

Toll-free Order Number

To order books by MasterCard or VISA, call 1-800-343-8321. Phone lines are open from 8:00 A.M. to 4:00 P.M., Eastern time.

*Price and terms quoted are U.S. only and are subject to change without notice. For prices outside the U.S., contact the nearest office of Educational Services, Digital Equipment Corporation.