Hardware for Fast Global Operations on Distributed Memory Multicomputers and Multiprocessors

Douglas V. Hall
Portland State University

HARDWARE FOR FAST GLOBAL OPERATIONS ON DISTRIBUTED MEMORY MULTICOMPUTERS AND MULTIPROCESSORS

by

DOUGLAS VINCENT HALL

A submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

In

ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1995

DISSERTATION APPROVAL

The abstract and dissertation of Douglas Vincent Hall for the Doctor of Philosophy in Electrical and Computer Engineering were presented December 9, 1994, and accepted by dissertation committee and the doctoral program.

COMMITTEE APPROVALS: [redacted]
Michael A. Driscoll, Chair

[redacted]
Marek A. Perkowski

[redacted]
W. Robert Daasch

[redacted]
Richard D. Morris

[redacted]
Jingke Li
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL: [redacted]
Rolf Schaumann, Chair
Department of Electrical Engineering

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 30 March 1995

ABSTRACT

An abstract of the dissertation of Douglas Vincent Hall for the Doctor of Philosophy in Electrical and Computer Engineering presented December 9, 1994.

Title: Hardware for Fast Global Operations on Distributed Memory Multicomputers and Multiprocessors

"Grand Challenge" problems such as climate modeling to predict droughts and human genome mapping to predict and possibly cure diseases such as cancer require massive computing power. Three kinds of computer systems currently used in attempts to solve these problems are "Big Iron" multicomputers such as the Intel Paragon, workstation cluster multicomputers, and distributed shared memory multiprocessors such as the Cray T3D. Machines such as these are inefficient in executing some or all of a set of global program operations which are important in many of the "Grand Challenge" programs. These operations include synchronization, reduction, MAX, MIN, one-to-all broadcasting, all-to-all broadcasting, and orderly access to global shared variables.

My hypothesis was that a secondary network with a wide tree topology and one or more centralized processors optimized for these operations could substantially decrease their execution time on all three types of systems. To test my hypothesis, I developed the secondary network and Coordination Processor(COP) system described in this dissertation, modeled the major blocks of the design in VHDL, and simulated these blocks to verify their logic and get realistic timing values.

The analyses developed for the COP system clearly demonstrate that it can speed up a variety of common global operations by as much as 2-3 orders of magnitude when added to any of several current multicomputers and multiprocessors. Examples show that this speedup reduces overall execution time for important scientific programs and computational kernels by an average of 25% at an increase in system cost of only about 2%. Further analyses show that for these global operations the COP system has a greater combination of speed and versatility than any other system.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor, Dr. Michael A. Driscoll who taught me much about computer architecture and networks, and whose many excellent questions helped me focus this project. Thanks to Dr. Richard D. Morris for convincing me to come to Portland State to work on a Ph.D. and to Dr. Marek A. Perkowski who introduced me to the world of university research. Thank you to Dr. W. Robert Daasch for his very helpful suggestions in the area of scientific computation and to Dr. Jingke Li for the Computer Science perspective he provided. I thank all of these people collectively for serving on my dissertation committee and for the care they put into reviewing my work.

To Shirley Clark and Laura Riddell of the Electrical Engineering Department staff, my thanks for the many ways they have helped me serve my students and survive the Ph.D. process.

On a more personal note, I wish to thank my parents, June D. Hall and Edwin J. Hall who made this project possible by teaching me at an early age the value of hard work and perseverance. A heartfelt thank you to my wife, Rosemary Marshall for strongly encouraging me to pursue graduate study, supporting me along the way, and fighting her way though the hell of leukemia to stay with me. Finally, I thank our children, Linda, Brad, Mark, Lee, and Kathryn for their support and understanding during this long trek.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Grand Challenges - Why We Need Massively Parallel Computers

One of the main motivations for developing powerful "supercomputers" has been to more rapidly solve very large scientific problems. These development efforts have become more focused since 1991 when the U.S. Office of Science and Technology's Committee on Physical, Mathematical, and Engineering Sciences published a report "Grand Challenges: High Performance Computing and Communication." [Physi91a] The HPCC report is basically a "wish list" of important computational problems that scientists and engineers would like to be able to solve in the 90's. These problems are not just mental exercises. Many of them are vitally important to very large numbers of people.

Global climate modeling which will make it possible to predict droughts and take steps to avoid famines is one of these. Better predicting the path of hurricanes and thus saving lives is another. Human genome mapping to predict and possibly cure diseases such as cancer is still another. Development of new medicines to replace those which no longer work because disease organisms have become resistant to them is a vitally important challenge. Simulating the airflow over cars or the wings of airplanes to determine the most fuel efficient designs and thus help reduce global warming is certainly a valuable contribution to the environment. The list goes on.

All of these "Grand Challenges" require several orders of magnitude more computational power than was available at the time the initial report was written. In an attempt to develop systems that meet these goals, various U.S. government agencies such as

ARPA, EPA, DOE, NASA, NSF, NIH, NIST, NOAA, and NSA have since then supplied massive funding to several university research teams and computer companies. Yearly reports summarize funding and achieved progress [HPCC94a].

The computational needs of the "Grand Challenge" problems are so great that a massively parallel array of several hundred or several thousand of the fastest currently available processors will be required to meet them.

As documented by Almasi and Gottlieb [Almas94a], and by Bell [Bell94a] , university researchers and computer companies have experimented with many, many different parallel computer architectures and parallel programming paradigms. Based on the many different systems currently in existence, it is a difficult task to determine the best way to connect a large number of processors and perhaps a more difficult task to write programs that efficiently utilize massive hardware parallelism.

My initial work focused on finding common parallel program operations that could potentially be done much more efficiently by specialized hardware, analogous to the way a numeric coprocessor speeds up floating point operations and a graphics coprocessor speeds up CRT display operations. This research disclosed that the operations most likely to benefit from hardware acceleration were global operations such as synchronization, broadcasting, reduction, and shared write-able variable access.

I then researched previous attempts at hardware assist for these operations and found that all of them had serious limitations. My hypothesis was that a secondary network with a wide tree topology and one or more centralized processors optimized for these operations could substantially decrease their execution time. To test my hypothesis, I developed the secondary network and Coordination Processor(COP) system described in this dissertation, modeled the major blocks of the design in VHDL, and simulated these blocks to verify their logic and get realistic timing values. This COP system can speed up a variety of common parallel program global operations by 2-3 orders of magnitude when added to any of several current commercial and research multicomputers or multiprocessors. This improvement helps increase the number of compute nodes that can be effectively applied to a task and/or helps increase the overall speedup available from a given number of compute nodes. Both of these are important in the pursuit of the "Grand Challenge" goals and in less ambitious parallel programming tasks.

I will show that the COP system has a greater combination of speed and versatility than any other and that it is very cost effective. To demonstrate the potential benefits of the COP system I will use the following standard methodology for evaluating new computer architectures.

To show the types of multicomputers and multiprocessors for which the COP system is intended, Chapter II gives brief overviews of the architectures of several current commercial and research machines. To give a thorough understanding of the global operations improved by the COP system, Chapter III describes in some detail how these operations are commonly performed on "Big Iron" multicomputers, on workstation cluster multicomputers, and on massively parallel multiprocessors. In Chapter III I also discuss major factors which contribute to the time required for these global operations, and based on this discussion I develop the justification of hardware support for global operations.

Chapter IV starts the detailed analysis of the COP system. This chapter first describes in detail the COP network architecture, compute node to COP network interface, and the COP itself. A discussion of how the target global operations are performed on the COP system follows. Also included in this chapter is an explanation of the software interface required for the COP system.

In Chapter V, I first give some COP system global operation performance data, based on timing values derived from VHDL modeling and simulation. In the second section of Chapter V, I compare the global operation performance of the COP system with the global operation performance of current commercial and research machines. Where possible, published data is used for these comparisons. Finally in Chapter V, I give examples of the overall speedup that a COP system can provide for some common types of scientific programs by decreasing the time required for their global operations.

Having thoroughly shown the benefits of the COP system, I then in Chapter VI compare and contrast other attempts to improve the efficiency of global operations on "Big Iron" multicomputers, workstation cluster multicomputers, and distributed shared memory multiprocessors. Finally, in Chapter VII, I give my conclusions and suggest some directions for my future work.

CHAPTER II

OVERVIEW OF SOME COMMON MIMD MACHINE ARCHITECTURES

SISD, SIMD, MISD, And MIMD Machines

The purpose of this chapter is to describe the architectures and network topologies of the types of computers for which the COP system is intended. Flynn's [Flynn72a] four categories for computer architectures are based on the number of concurrent instruction streams and the number of concurrent data streams. Single Instruction Single Data (SISD) includes the standard uniprocessor. The Single Instruction Multiple Data (SIMD) category includes machines such as the CM-2 or a systolic array in which the same instructions are broadcast to each processor in parallel. Each processor then synchronously executes these instructions on different data. The Multiple Instruction Single Data (MISD) category is mainly good for generating philosophical arguments because there seem to be no actual examples of this kind of machine. Finally, the Multiple Instruction Multiple Data (MIMD) category includes machines with multiple processors which can each execute independent instructions streams. As shown in Figure 1, the MIMD category can be further divided into three sub-categories: shared memory machines or multiprocessors, distributed memory machines or scalable multicomputers, and distributed shared memory machines or scalable multiprocessors. The COP system is applicable to distributed memory and to distributed shared memory systems, but for comparison I will briefly describe the characteristics of all three, give commercial and/or research examples of each type, and discuss some of the trade-offs in each type.

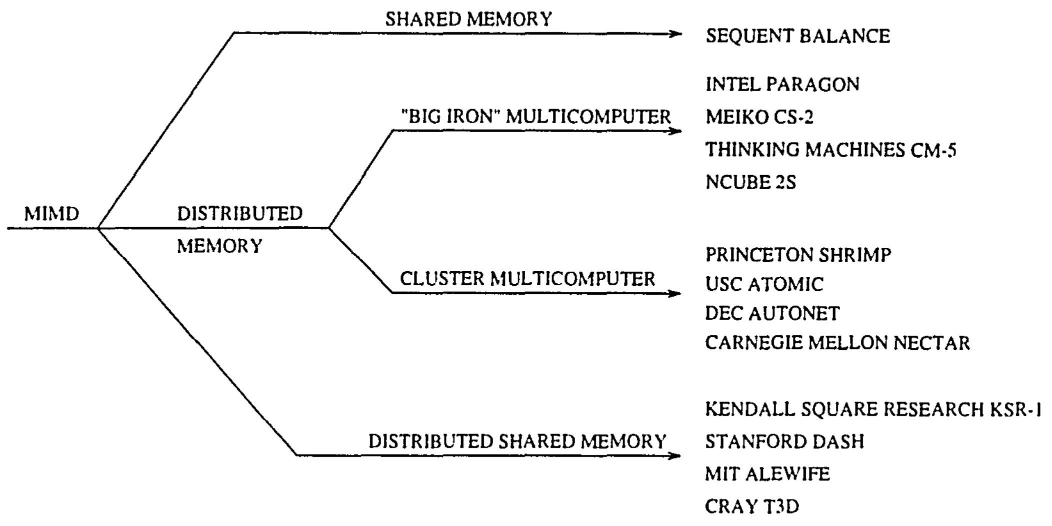


Figure 1. MIMD machine sub-categories and example systems

Shared Memory MIMD Machines

Figure 2 shows a block diagram for a simple shared memory MIMD machine such as the Sequent Balance. In this kind of machine processors access the shared memory over a common parallel bus. To reduce bus traffic to and from the shared main memory, each processor has a local cache which holds its current code and data sets. The single, global address space of this kind of machine makes it relatively easy to program because each processor can directly access all data. However, these machines have a several basic problems.

One problem is maintaining coherence of shared write-able variables. The ideal perhaps would be to have a separate memory for storage of just these variables. Assuming only one processor at a time could access this special memory, then strict coherence would be guaranteed because each processor would always see the last value written. An approximation to this is to declare shared variables non-cachable so that each processor always goes to main memory to access them. A difficulty with this approach is that it decreases the cache hit rate and thereby increases the average memory access time. Another difficulty with this approach is that it increases traffic and congestion on the main memory bus.

If shared write-able variables are allowed to be cached, then the main memory values and any copies in other caches must be invalidated or updated so that no processor uses stale values of the variables. Archibald [Archi86a] describes in detail many schemes for doing this. Common to several of these is to have the cache controllers monitor or "snoop" bus transactions. When a cache controller detects a write to a variable contained in its cache, it either invalidates or updates its copy of the variable. Invalidation is the simplest to implement, but invalidation means that each processor must read the value of the variable from main memory again when needed. An alternative is to in some way broadcast the new value of a variable to all the caches which hold copies of it.

Another major problem of simple bus-based shared memory machines is caused by the fact that main memory requests must be serviced sequentially. Thus as the number of processors in a system is increased, the average time that each processor has to wait for service also increases. This memory/bus contention reduces the time that each processor is doing useful work and thereby reduces the benefits of increasing the number of processors. Another way of expressing this is that bus and/or memory access saturation limits the ability of simple bus-based shared memory MIMD machines to scale to a large number of processors. Although they are relatively easy to program, the difficulty of scaling simple shared memory MIMD machines to large numbers of processors motivated development of the distributed memory MIMD machines discussed next.

Distributed Memory MIMD Machines

In a distributed memory MIMD machine each compute node has a processor and an independent memory. Compute nodes communicate with each other by passing messages

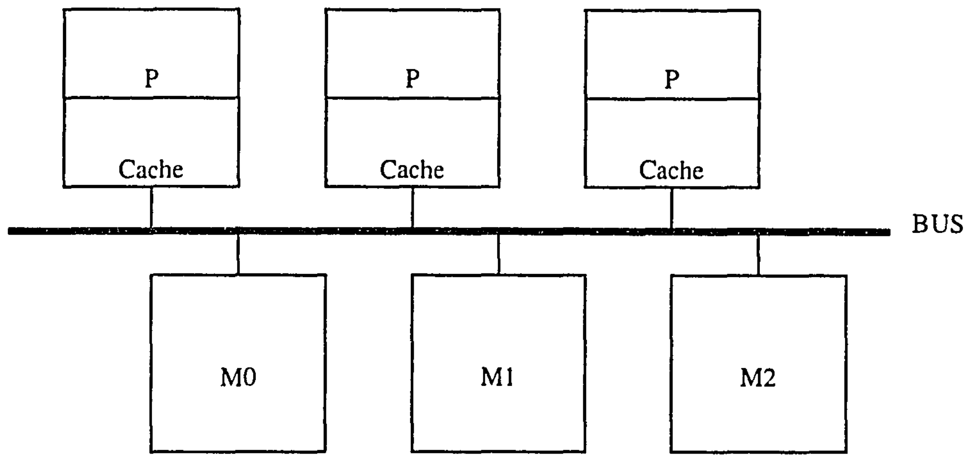


Figure 2. Common bus shared memory MIMD architecture

over some type of interconnection network. This kind of machine is often called a scalable multicomputer(sMC) or just a multicomputer(MC) because each node can execute programs independently.

As shown in Figure 1, there are two basic multicomputer categories. The first category is the so-called "Big Iron" machine where all the compute nodes are in one or several large, adjacent cabinets. The second category represents multicomputers consisting of a group of independent workstations connected by a network. In this type the compute nodes(workstations) may form a Local Area Network(LAN) based cluster within a room or building, or a Wide Area Network(WAN) based system spread around the whole world. Since my work relates closely to both "Big Iron" multicomputers and LAN based workstation cluster multicomputers, I will give a brief overview of some common multicomputer architectures and comment on their tradeoffs.

Common "Big Iron" Multicomputer Topologies

In an attempt to optimize message-passing efficiency, scalability, and program-machine compatibility, commercial companies and researchers have tried many different multicomputer topologies. For the sake of brevity, I will mostly confine my discussion to topologies that have enjoyed some commercial success and/or are directly relevant to my work. For a more extensive survey, see Almasi and Gottlieb [Almas94a].

One common multicomputer topology is the binary hypercube topology shown in Figure 3. Commercial systems which use this topology include the Intel iPSC/2, the Intel iPSC/860, and the nCube-2S [Zorpe92a]. Both the strength and the weakness of this topology are that the number of nearest neighbors for any processor is equal to the $\log_2(N)$ where N is the number of processors. This is a strength because the interconnection richness means that the diameter of the network, D, or the maximum number of message "hops" between any two nodes is just $\log_2(N)$. It is a weakness because the number of connections to each node must increase as N is increased. In a 64 node hypercube, for example, each node requires connections to 6 neighbors, but in a 1024 node hypercube each node requires connections to 10 neighbors. Because of this difficulty in scaling hypercubes to very large N, several recent systems such as the Intel Delta and the Intel Paragon [Zorpe92a] use a two dimensional mesh topology such as that shown in Figure 4.

The advantage of the 2-D mesh topology is that the maximum number of connections to neighboring nodes is 4 for any N. This makes system hardware design much easier because the number of network connections on each node does not have to change as the system is scaled to large values of N. Also, with a 2-D mesh, a relatively small number of additional nodes is needed to incrementally increase the size of a system while maintaining symmetry.

A disadvantage of the 2-D mesh topology is that the diameter of the network, D, is greater than that for an equivalent size hypercube. In a rectangular 2-D mesh where L is the number of nodes along one edge and W is the number of notes along an orthogonal edge, $D = (L-1)+(W-1)$ and the number of nodes, N, is equal to $L \times W$. In a square 2-D mesh, L and W are equal so D is $2\sqrt{N} - 2$. For a 32x32 node 2-D mesh, the maximum

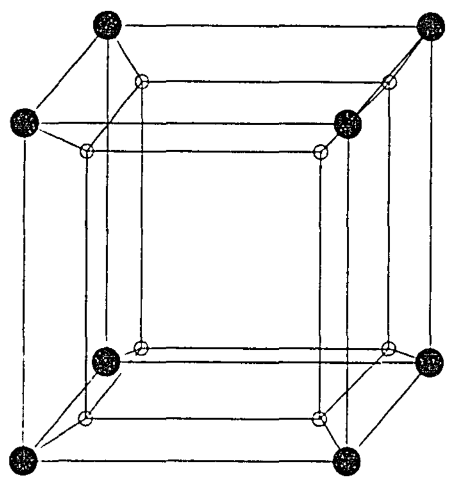


Figure 3. Hypercube multicomputer topology

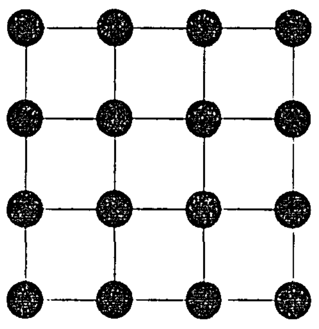


Figure 4. 2D-Mesh multicomputer topology

number of hops is then $2\sqrt{1024} - 2 = 60$. This is much greater than the number for a 1024 node hypercube where the maximum number of hops is $\log_2(1024) = 10$. However, Dally [Dally90a] and Chittor [Chitt90a] have shown that in large multicomputers mesh interconnection networks can have wider and faster bandwidth channels than hypercubes, if wire density limitations are considered. They further demonstrated that these higher bandwidth channels allow a mesh interconnection network to outperform a hypercube network, if contention for network channels is low. The "if" in this statement implies that to realize the benefit of the mesh topology, the contention for channels must be kept low. As we show later, the addition of our secondary network and coordination processors to a mesh connected MIMD machine reduces message traffic on the main interconnection network and thereby helps maintain the speed advantage of the more easily scalable mesh topology.

Another interesting commercial multicomputer topology is the "fat tree" topology used in the Thinking Machines CM-5 [Leise92a]. The term "fat" refers to the fact that the bandwidth is greater in the connecting links closer to the root. Kwan [Kwan93a] has shown that that this topology, or at least the CM-5 implementation of it, is communication limited for some common scientific applications such as two-dimensional FFT and Gaussian elimination, so the long range potential of this topology is perhaps in doubt.

Workstation Cluster Multicomputers

In the past, large scientific computations have been relegated to vector supercomputers such as CRAY Y-MPs, or the "big iron" multicomputers such as the Intel Paragon, Thinking Machines CM-5, and the N-Cube 2S machines that we described in the preceding section. Recently, however, considerable research effort has been directed toward using clusters of workstations connected by Ethernet or some other network to perform these computations. Ewing [Ewing93a] for example, has experimented with simulating the propagation of seismic waves on a network or IBM RS/6000 workstations, Manjikian [Manji16a] has successfully run large logic simulations on a network of Sun IPC workstations, and Castagnera [Casta94a] reports extensive work implementing the NAS benchmarks and other scientific programs on a cluster of Silicon Graphics, Inc. systems.

The advantages of using clusters of networked workstations as multicomputers are:

1. General purpose workstations can perform many different functions and because of this, realize an economy of scale from a cost standpoint.

2. The cumulative memory, disk space, compute power, and I/O capability of a workstation cluster can be very large.

3. Otherwise unused workstation clock cycles can be harvested for productive work.

Based on these advantages, Gordon Bell [Bell94a] predicts that, as the hardware and software problems of networked clusters are solved, clusters will in many cases replace the "Big Iron" multicomputers.

A major software factor which limits the performance of workstation cluster systems is the software "protocol stacks" which must be traversed in order to send a message on one machine and receive the message on another machine.

The major hardware based problem to overcome in developing this type system is low interconnection network performance. Many existing LANs still use Ethernet networks that have maximum bandwidths of only 10 Mbits/sec or 1.25 Mbytes/sec. In addition to its relatively low hardware bandwidth, a further limitation of Ethernet is its common bus topology which requires that workstations compete for network access. As the number of nodes on the network and/or the amount of message traffic increases, the effective bandwidth of the network decreases. FDDI increases the hardware bandwidth to 100 Mbits/sec or 12.5 Mbytes/sec, but the FDDI token ring topology suffers the same decrease in effective bandwidth as the number of nodes is increased. These bandwidths are much less than the bandwidth of current "Big Iron" multicomputer networks. For a bandwidth comparison, according to Rosing [Rosin94a] , the Intel Paragon multicomputer network currently has a node-to-node maximum message bandwidth of 200 Mbytes/sec.

One approach currently used to improve the communication performance of a workstation cluster is to connect the nodes in a mesh or similar topology with high-performance worm-hole routers similar to those used in "big iron" multicomputers. The

SHRIMP [Blumr93a] system, in fact, uses an Intel Paragon router backplane to connect a cluster of workstations in a 2-D mesh topology. Another example of a cluster system that uses a high-performance router based network is the Tandem Computer [94a] system. As still another example of this approach, the ATOMIC system [Cohen94a] at USC uses Caltech Mosaic [CLSei93a] smart router chips to implement a crossbar switch based network for a cluster of workstations. Carnegie-Mellon's Nectar system [Kung91a] and DEC's Autonet system [Rodeh93a] also use high performance crossbar switches to implement workstation networks.

Workstation cluster systems seem to be evolving toward networks and network topologies similar to those of some of the "Big Iron" machines. Later I show that the COP system is applicable and beneficial to both types of systems.

Distributed Shared Memory MIMD Machines

Simple shared memory multiprocessors such as that shown in Figure 2 are relatively easy to program because each processor has direct access to all data. However, their scalability is limited to a few tens of processors by bus saturation. Distributed memory multicomputers such as those discussed in the preceding section scale to thousands of compute nodes, but they are somewhat difficult to program. The reason for this is that, at the lowest level, a programmer must send and receive explicit messages for all remote data transfers, global operations, synchronization, etc. A great many attempts have been made to ease the task of programming distributed memory machines or at least to some degree insulate the programmer from the low level details of message passing. These efforts fall easily into four general categories: libraries of message-passing based functions that programmers can call, programming environments that use higher level constructs to make programming easier, programming languages or systems that implement a shared-memory programming paradigm on top of message passing, and machines that implement distributed shared memory in hardware.

Examples of basic message-passing libraries include the Basic Linear Algebra Communication Subprograms(BLACS) [Donga93a] and the Message Passing Interface(MPI) standard [Tennea]. These libraries are used in essentially the same way as other libraries in, for example, Fortran or C programs.

Higher level environments to assist in writing programs for workstation cluster multicomputers include Parallel Virtual Machine(PVM) [Manch94a], ParaSoft Corporation's Express [Ali94a], and p4 [Butle92a]. These environments have callable functions for creating, running, and stopping distributed processes, determining process status, synchronizing processes, and communicating between processes. They therefore lift the programmer from the basic message passing level to the process level.

Programming systems that implement a shared-memory programming paradigm on top of the message passing layer include Scientific Computing Associates' Linda [Carri89a], Kali [Koebe89a], Munin, [Benne90a], and Shiva [Li89a]. Nitzberg and Lo [Nitzb91a] give an excellent survey of the issues and tradeoffs involved in implementing shared memory on a distributed memory machine and then briefly compare the preceding and other approaches. Stumm and Zhou [Stumm90a] analyze the performance of several software algorithms for implementing distributed shared memory on message passing machines.

Several programming languages have been created to allow programmers to use a shared object-oriented programming model on parallel machines. Examples of this approach are Orca [Tanen92a], Emerald [Jul88a], and mentat [Grims93a].

The three types of programming aids described above all make the programmer's job easier, but they may actually make a program run slower than a program hand-crafted with basic message-passing primitives. One reason for this is that the generality required of library routines may introduce extra overhead. Likewise, the extra software layer required by high level program constructs or by the shared memory paradigm introduces additional execution overhead. To avoid this added overhead, several multiprocessor systems implement distributed shared memory directly in hardware.

An early example of a distributed shared memory machine was the IBM RP3 [Pfist85a]. Each processor-memory element in this machine contained a processor, a cache, some local memory, and a portion of the global memory. The global memories were connected to a multi-stage banyan network. Memory references not in a processor's local memory were routed through the network to the memory element that contained that global address. Memory coherence was maintained by declaring shared write-able variables uncachable. As will be discussed later, the RP3 also had a secondary network for synchronization and for combining multiple simultaneous reads directed at a single memory location.

An example of a commercially available distributed shared memory machine is the Kendall Square Research KSR1 [Saave93a]. The topology for this machine is a hierarchy of rings. Each level 0 ring can contain up to 32 processors. Each level 1 ring can have up to 34 level 0 rings attached to it. A significant feature of the KSR1 is its ALLCACHE memory hierarchy which includes just a 512 Kbyte first level cache, 32 Mbyte second level cache, and disk on each processor. Since processors have no fixed address main memory, this hierarchy is referred to as a Cache Only Memory Architecture or COMA. The entire memory is managed as one large virtual address space. Directories are used to keep track of the location and status of memory pages and maintain cache coherence. If a processor addresses a memory location which is not present in its local caches, a high speed "search engine" queries the directories around the ring(s) until it finds the desired page. The page is then transferred or copied to the cache of the requesting processor. All of the queries, copies, and transfers are actually done by passing messages, but a programmer simply sees a multiprocessor machine with a single large virtual address space.

The Stanford DASH Multiprocessor [Lenos93a] is another example of a distributed shared memory machine, but its architecture is very different from that of either the RP3 or the KSR1. The basic building block of the DASH system is a cluster containing 1-4 processors, each with its own cache. The processors within a cluster are connected on a common bus and use the snooping protocol described earlier to maintain cache coherence within the cluster. Each cluster also contains a block of the global distributed memory, a directory, and inter-cluster interface circuitry. A 2-D mesh Request network and a 2-D Reply network are used to connect clusters in the system. The maximum size of the current system is 16 clusters of 4 processors each.

The distributed directories in a multicluster DASH system have an entry for each 16-byte line of the global memory. A directory entry holds the address of the line and bits which indicate whether the line is uncached, shared, or dirty. Memory functions as a 4-level hierarchy. If a processor does not find a desired word in its local cache, it outputs the request on the cluster bus. If the word is present in one of the cluster caches, the appropriate snoopy controller supplies the line containing the word. If the word is not in the local cluster, a directory is used to find the "home" cluster for the line. If the line is not marked dirty, then the line will be supplied from the home cluster memory. If the line is marked dirty, the line will be supplied from the remote cluster cache where the dirty copy resides. Messages are used to access home and remote clusters. For future reference, the number of clocks required for accessing the four levels are 1:15:101:132 according to Lenoski [Lenos93a]. Another interesting research system, the MIT Alewife [Agarw90a] system, has a modified SPARC processor called a SPARCLE at each node of a 2-D mesh network. The Alewife system uses a directory based cache coherency approach, but to reduce the required directory size, the Alewife directories have entries for only a limited subset of the global memory lines and trap to software in the case of a directory miss. The SPARCLE processor is designed to cover this and other extended memory latency by rapidly swapping execution threads.

Other current research-based distributed shared memory systems include ASURA [Mori93a], PLUS [Bisia90a], and Gallactica Net [Wilso93a].

One of the latest commercial examples of a distributed shared memory machine is the Cray T3D. [Koeni94a]. This machine is actually a "backend" processor for a large Parallel Vector Processor host such as a Cray C90, rather than a stand-alone multiprocessor.

The 3-D torus topology used in the machine is basically a cube with the opposite faces connected so that rows form rings and columns form rings. Two processing elements are connected at each node in the torus, so an 8x8x8 torus can have up to 1024 processors. The three dimensions and the wrap-around connections drastically reduce the diameter of the network and thus decrease the maximum number of hops between any two processing elements. However, this topology requires more connections to each processing element than does, for example, a 2-D mesh topology. Also, this topology requires more nodes than a 2-D mesh to incrementally and symmetrically increase the size of a machine. As an example, an 8x8x8 torus requires 64 processing elements to increase to an 8x8x9 machine, but a 16x32 2-D mesh only requires 16 to increase to a 16x33 mesh.

Each processing element in the TD3 can have up to 64 Mbytes of DRAM memory which is part of the physically distributed, logically shared memory. Memory transfers are done with message packets containing payloads of one or four 64-bit words. The DEC 21064APX processor used in each processing element has an 8 Kbyte instruction cache and an 8 Kbyte direct-mapped write-through data cache. According to Koeninger [Koeni94a], the T3D has no hardware support for maintaining coherence of local data caches with respect to remote memory, so this is the responsibility of the Cray Research Adaptive FORTRAN(CRAFT) programming model developed for the system.

The possible programming advantage gained by implementing the distributed shared memory paradigm in hardware does not come cheaply. According to Lenoski [Lenos93a], for example, about 33% of the board area, 13.9% of the SRAM, and 13.7% of the DRAM in a DASH cluster is used for the cache directory circuitry. Although not specifically mentioned in the references, it seems that the global search engine in the KSR1 or the global memory access hardware in the T3D also requires considerable hardware to implement.

Chapter Summary

This chapter has described some of the current attempts to develop scalable computer systems with hundreds or thousands of compute nodes. Message-passing-only multicomputers such as the Intel Paragon or a workstation cluster are somewhat difficult to program at the lowest level, because the programmer must send and receive explicit messages for all remote memory accesses, global operations, etc. Furthermore, as will be shown in Chapter V, the combination of message latency and network topology makes some common global operations very costly on these systems. Parallel programming environments such as PVM, which run on top of the basic message-passing substrate of these machines make a programmer's job somewhat easier, but add considerable software overhead and do not remove the limitations imposed by the basic message-passing mechanism and the network topologies.

The difficulty of programming distributed memory message-passing-only multicomputers has motivated a large effort at implementing the shared memory programming paradigm on distributed memory machines. Software implementations of distributed shared memory built on message-passing substrates, however, suffer not only the message-passing and topology limitations but also the added overhead of an extra layer of software. Hardware implementation of distributed shared memory, as in the Stanford DASH system, for example, requires a large amount of circuitry to keep track of memory line locations, maintain cache coherency, etc. Also, although global shared memory makes the programmer's job perhaps easier, in Chapter V I show that some common global operations are unduly costly on this kind of machine and that a COP system can speed up these operations.

The task of the next chapter is to give a solid explanation of how common global operations are currently performed on "Big Iron" multicomputers, cluster multicomputers, and distributed shared memory multiprocessors so that later discussions of the COP system and comparisons with these systems will be more understandable.

CHAPTER III

MIMD MACHINE PROGRAMMING AND PROGRAM OPERATIONS

Requirements for SPMD Programs on MIMD Machines

Since MIMD machines have independent instruction streams, they can be used in Single Program Multiple Data (SPMD) mode where parts of a single large program are executed on multiple processors, or they can be used in Multiple Program Multiple Data (MPMD) mode where several independent programs are executed on the machine. In this section I will discuss the requirements for SPMD mode parallel programs and then in the following section I will use an example program to show the operations used to meet these requirements on current multicomputers and multiprocessors. After a short section discussing the additional system requirements for MPMD program execution, the final section of the chapter analyzes the cost of the previously described program operations on multicomputers and multiprocessors.

The first major consideration in adapting a uniprocessor program for execution on a MIMD machine is partitioning the overall task into processes that can be distributed among the compute nodes. One common way to do this is to create several identical processes, assign each process to a compute node, and have each process work on a subset of the data. Another common way is to create different processes which each perform various sub-tasks within the overall program.

The second major consideration in the program adaptation process is scheduling execution of processes on compute nodes. Two common methods are static scheduling and dynamic scheduling. With static process scheduling, processes are distributed to compute nodes at runtime and remain there for the duration of program execution. On a distributed memory MIMD system the data to be operated on by each process will likely be distributed to the appropriate compute nodes along with the process code. If the processes are identical and have equal data sets, then static scheduling provides a good load balance among the compute nodes. However, if the processes are very different or if the data does not divide easily among the processes, then dynamic scheduling may provide better load balancing. With dynamic scheduling, each compute node is programmed to seek new work when it completes its current work. Depending on the application, an idle node may simply grab a new chunk of data from an array, or it may acquire a new process from a queue of waiting processes.

A third important point to consider is implementation of global operations in which each compute node must participate. Several common parallel algorithms, for example, require that each process contributes a value and that the global result of the contributed values be returned to all the processes.

Synchronization is another major point to consider in adapting a uniprocessor program to run on multiple compute nodes. The access of multiple processes to shared writeable variables must be synchronized so that only one process at a time can read-modify-write the variable. Also, the execution sequence of multiple processes must be synchronized so that the result of a computation is the same as it would be on a sequential (uniprocessor) machine. A common form of this latter kind of synchronization requires that all processes reach a specific point before any are allowed to continue.

The following sections use a common numerical computation algorithm to illustrate the program operations used to satisfy these requirements in a MIMD machine program.

Implementing The Jacobi Algorithm On Parallel Processors

To illustrate the concepts of partitioning, scheduling, and synchronization and to show other operations commonly required by programs executing on MIMD machines, I will use the Jacobi Relaxation method for the numerical solution of Laplace's equation. As pointed out by Bertsekas [Berts89a], this algorithm is representative of a wide class of iterative numerical methods for solving systems of linear equations and is more easily parallelized than the somewhat similar Gauss-Seidel method. The Jacobi relaxation method can be used for such diverse applications as finding the temperature at specified points on a plate with fixed temperatures on the edges or finding the voltage at specified points on a metal plate with fixed voltages at the edges. In the latter case, for example, if v(x,y) is the function which represents the voltage at any point on a metal plate with fixed voltages at the edges, then the Laplace equation is:

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0$$

To solve this equation numerically with the Jacobi method, the plate is first divided into equally spaced regions with grid points as shown in Figure 5. An iterative approach is then used to produce better and better approximations for the voltage at each of these grid points. During each iteration an approximation for the voltage at each grid point is determined by averaging the values of its four nearest neighbors. In the first iteration the values for points near the center of the grid will all be zero, so most of the computed values may not be very good approximations. With successive iterations, however, the effect of the fixed boundary values propagates toward the center points and produces better approximations for all points, and the point values converge toward a solution. When the maximum change in value for any point from one iteration to the next becomes less than some desired precision, commonly called the maximum norm or the infinity norm, the process is stopped.

```
     0    1    2    3    4    5    6    7    8    9
0        (v)  (v)  (v)  (v)  (v)  (v)  (v)  (v)

1   (v)   .    .    .    .    .    .    .    .   (v)

2   (v)   .    .    .    .    .    .    .    .   (v)

3   (v)   .    .    .    .    .    .    .    .   (v)

4   (v)   .    .    .    .    .    .    .    .   (v)

5   (v)   .    .    .    .    .    .    .    .   (v)

6   (v)   .    .    .    .    .    .    .    .   (v)

7   (v)   .    .    .    .    .    .    .    .   (v)

8   (v)   .    .    .    .    .    .    .    .   (v)

9        (v)  (v)  (v)  (v)  (v)  (v)  (v)  (v)
```

Figure 5. Grid points for numerical solution of Laplace equation for metal plate with fixed voltages applied at edges

The example I will use here to show how this algorithm can be implemented on MIMD machines is an extension of one developed by Lester [Leste93a]. To start developing this example, Figure 6 shows a sequential pseudo-program for this algorithm. The program follows the verbal algorithm in the preceding paragraph quite closely but there are a couple of points that need clarification.

First, as shown in Figure 6 the data array is larger than the actual array of grid points so that there is room for the boundary values. The i and j loops which compute the value for each point only iterate over the actual grid points, so they do not change the boundary values.

Second, the new values computed for each point are put in a second array, rather than being written over the old values. This is necessary because the Jacobi algorithm explicitly specifies that all new values must be computed using values from the previous iteration. If new values were immediately written over old, some computations would use

```
do
   {
   max_delta = 0;
   for(i=1; i<n+1;i++)
     for(j=1;j<n+1;j++)
       {
        new[i][j] = 0.25 * (old[i-1][j] + old[i+1][j] +
                            old[i][j-1] + old[i][j+1]);
        delta = ABS(new[i][j] - old[i][j]);
        if(delta > max_delta)
           max_delta = delta;
       }
   for(i=1; i<n+1;i++)
     for(j=1;j<n+1;j++)
        old[i][j] = new[i][j];
   }
while(max_delta > 1E-7);
```

Figure 6. Body of sequential C program for Jacobi relaxation

old values and others a mixture of old and new. Note: The Gauss-Seidel method allows this, but for the reason stated above, I have used the Jacobi method here. Only after the new values for all points have been computed are they copied back to the original array to get ready for the next iteration.

Third, to check for convergence, the difference between the new value and the old value is determined for each point. When the maximum difference for all the points is less than the desired precision, execution is terminated.

The pseudo-program in Figure 7 shows in general how the sequential Jacobi program must be modified to execute on multiple processors. The two lines at the top of this program distribute the point computations among the processors by assigning the computations for one row of the matrix to each of n processors. Each processor also receives and executes a copy of the do-while block in the program. This is an example of static scheduling.

For the actual computation with this distribution, each processor just loops over the range of the column index variable, j. After computing the new value for a point, the processor determines the absolute value of the change in value for that point and updates the

```
for(i=0; i<n_row; i++)
assign row i to process i;
do
   {
   local_max_delta = 0;
   for(j=1;j<n+1;j++)
       {
        new[i][j] = 0.25 * (old[i-1][j] + old[i+1][j] +
                    old[i][j-1] + old[i][j+1]);
        delta = ABS(old[i][j] - new[i][j]);
        if(delta > local_max_delta)
           local_max_delta = delta;
       }

   BARRIER    /* Wait for all procesors to compute new[i][j] */

   for(j=1;j<n+1;j++)
    old[i][j] = new[i][j]; /* Copy new values to base array */

/* Accumulate global_max_delta */
   LOCK
   if(local_max_delta > global_max_delta)
       global_max_delta = local_max_delta;
   UNLOCK
   BARRIER   /* Wait until all processors have contributed */

   }
while(global_max_delta > 10E-7); /* Play it again, Sam? */
```

Figure 7. Pseudo parallel program for Jacobi relaxation

local_max_delta variable accordingly. When each processor finishes all of its j loop iterations, the variable local_max_delta will have the maximum change for all the points computed by that processor.

The BARRIER statement shown next in the program represents a synchronization construct devised by Jordan [Jorda78a]. A barrier forces each processor to wait until all processors have reached that point in the program. The purpose of the barrier here is to make sure that, as required by the Jacobi algorithm, all the new point values have been computed before any are copied to the old array.

The next phase in the pseudo program in Figure 7 is to accumulate the local_max_delta values from all the processors into one global value which will be used to decide whether to execute another do loop. For this operation each processor must access the global variable, perform a comparison with its local_max_delta, and write the result back in the global variable. If there were no control over when a processor could access the global variable, a major problem might occur. For example, if one processor reads the value of global_max_delta and then another processor reads the value of global_max_delta before the first writes a modified value, each processor will be comparing to the same value of global_max_delta. The result may be erroneous and is nondeterministic because of timing nuances. To prevent this problem, processor accesses to a global write-able variable must be synchronized in some way.

One way to synchronize accesses to a shared write-able variable is with a programming construct called a lock. If a processor examines the lock and finds it unlocked, the processor is allowed to lock the lock to exclude others, access the shared variable, and then, when done, unlock the lock so that another processor can access the variable. If a processor finds the lock locked, it must keep checking the lock over and over until it finds the lock unlocked.

The second barrier in the program in Figure 7 is required to make sure that computation of the global_delta_max is completed before any processor uses the value of global_data_max to decide whether to do another iteration or to terminate execution.

In summary, parallelizing a sequential Jacobi program requires a mechanism to make sure all processors have completed computation before transferring new values, a mechanism to synchronize access to the global_delta_max variable, and a mechanism which requires all processors wait for the final result of the global_delta_max calculation before continuing. The next section shows how these and some other commonly needed operations can be implemented on a message-passing 2-D mesh connected multicomputer.

Program Operations On A Message-Passing MIMD Machine

Figure 8 shows a Multi-Pascal [Leste93a] program I wrote to demonstrate how the operations required for the Jacobi algorithm can be implemented on a message-passing MIMD machine with a 2-D square mesh topology. This topology was chosen because of its direct applicability to "Big Iron" machines such as the Intel Paragon, newer cluster multicomputers such as Shrimp or ATOMIC, and distributed shared memory systems such as DASH or Alewife.

The program in Figure 8 roughly follows the pseudo-program in Figure 7. As an overview, each compute node executes a copy of the Updaterow procedure which exchanges values with neighboring rows, computes new values for the points in a row, and calls the Aggregate function to check for the termination condition. For simplicity this example has only 16 compute nodes and a 16 x 16 grid of data points, but by changing the values of the n, m, and L constants at the start of the program it can be used with larger systems and/or more data points.

```
PROGRAM Jacobi3; (* JACOBI FOR 2-D MESH *)
ARCHITECTURE MESH2(4);
CONST n = 16;       (*number of processors*)
      m = 4;        (*number of processors one edge*)
      L = 16;         (*number of points per row*)
      numiter = 1; (*iterations between termination checks*)
      tolerance = 0.1;
TYPE rowtype = ARRAY [0..L+1] OF REAL;
VAR A: ARRAY [0..n+1] OF rowtype; (* Data array *)
    i,row,col: INTEGER;
    upchan,downchan: ARRAY [1..n] OF CHANNEL OF rowtype; (*com ports*)
    inchan: ARRAY [1..n] OF CHANNEL OF BOOLEAN;
    snake: ARRAY [1..n] OF INTEGER;

FUNCTION Aggregate(mynum:INTEGER;mydone:BOOLEAN):BOOLEAN;
VAR     prow, dnproc, uproc: INTEGER;
BEGIN
%SEQOFF;
  prow := ((mynum-1) DIV m) +1;   (* Process row number *)
  IF prow = 1 THEN
    BEGIN
     dnproc := 2*prow*m+1-mynum; (* Compute down process number *)
     inchan[dnproc] := mydone; (* Pass mydone down column *)
     mydone := inchan[mynum];  (* Wait for broadcast return *)
    END
  ELSE IF (prow > 1) AND (prow < m) THEN
    BEGIN
```

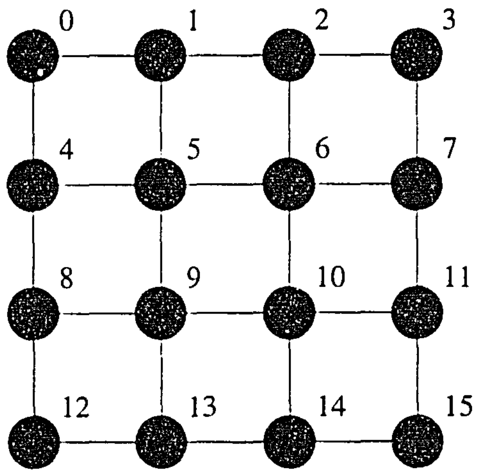
```
      dnproc := 2*prow*m+1-mynum; (* Compute down process number *)
      uproc := 2*(prow-1)*m+1-mynum; (* Compute up process number *)
      inchan[dnproc] := mydone AND inchan[mynum]; (* Pass result down *)
      mydone := inchan[mynum]; (* Wait for broadcast return *)
      inchan[uproc] := mydone; (* Pass broadcast up column *)
     END
   ELSE   (* prow = m *)
     BEGIN
      uproc := 2*(prow-1)*m+1-mynum; (* Compute up process number *)
      IF mynum = n+1-m THEN  (* Corner process in mesh *)
       BEGIN
        mydone := inchan[mynum] AND mydone; (* COL AND mydone *)
        inchan[mynum+1] := mydone; (* Pass across row *)
        mydone := inchan[mynum]; (* Wait for broadcast *)
        inchan[uproc] := mydone; (* Send up column *)
       END
      ELSE IF (mynum > n+1-m) AND (mynum < n) THEN (* Center processes *)
       BEGIN
        mydone := inchan[mynum] AND mydone; (* COL AND mydone *)
        mydone := inchan[mynum] AND mydone; (* COL, row, my result *)
        inchan[mynum+1] := mydone; (* Pass result across row *)
        mydone := inchan[mynum]; (* Wait for broadcast *)
        inchan[mynum-1] := mydone; (* Send across row *)
        inchan[uproc] := mydone; (* Send up column *)
       END
      ELSE  (* mynum = n *)
       BEGIN
        mydone := inchan[mynum] AND mydone; (* COL AND mydone *)
        mydone := inchan[mynum] AND mydone; (* Final result *)
        inchan[mynum-1] := mydone; (* Send across row *)
        inchan[uproc] := mydone; (* Send up column *)
       END
      END;
    Aggregate:= mydone; (* Pass result back to callee *)
%SEQON;
END;

PROCEDURE Updaterow(me: INTEGER; myrow: rowtype; VAR out: rowtype);
  VAR j,k,loops: INTEGER; maxchange, change: REAL;
      newrow,uprow,downrow: rowtype;
      done: BOOLEAN;
  BEGIN
    newrow[0] := myrow[0]; newrow[L+1] := myrow[L+1]; (*End points*)
    IF me = 1 THEN downrow := downchan[me]; (* Sent in main *)
    IF me = n THEN uprow := upchan[me]; (* Sent in main *)
    REPEAT
     FOR k := 1 TO numiter DO
      BEGIN
        IF me > 1 THEN
            upchan[me-1] := myrow; (* Send my row to me-1 *)
        IF me < n THEN
          BEGIN
            downchan[me+1] := myrow; (* Send my row to me+1 *)
            uprow := upchan[me];
          END;
        IF me > 1 THEN
           downrow := downchan[me]; (* Receive new downrow *)
        maxchange := 0;
        FOR j := 1 TO L DO
          BEGIN
            (* Compute average of adjacent points *)
            newrow[j] := (myrow[j-1]+myrow[j+1]+uprow[j]+downrow[j])/4;
```

```
             change := ABS(newrow[j]-myrow[j]);
             IF change > maxchange THEN maxchange := change;
           END;
         myrow := newrow;
       END;
       IF me = n THEN loops := loops + numiter;
       done:= Aggregate(me, maxchange < tolerance); (*Termination test*)
      UNTIL done;
    IF me = n THEN WRITELN('Loops to converge = ',loops);
    out := myrow; (*Write final answer back to A array*)
 END;

      BEGIN
      (* Set up process allocation array so adjacent processes *)
(* are on adjacent processors to minimize communication distances *)
  FOR i:= 1 TO n DO
    BEGIN
      IF ((i-1) DIV m) mod 2 = 0 THEN (* Even row *)
         snake[i] := i-1
      ELSE                            (* Odd row *)
         snake[i] := (((i-1) DIV m) +1)*m -1 - ((i-1) MOD m);
    END;

(* Set up data array with initial values*)
(* Number of data points = n*L *)
  row := 0;
  FOR col:=1 TO L DO
     A[row,col] := 100.0; (* Top boundary values = 100.0 *)
  FOR row:=1 TO n+1 DO (* Left and bottom boundary values = 0.0 *)
    FOR col:= 0 TO L DO
      A[row,col] := 0.0;
  col := L+1;
  FOR row := 1 TO n DO (* Right boundary values = 100.0 *)
     A[row,col] := 100.0;

(* Send fixed boundary values to channels *)
  downchan[1] := A[0]; upchan[n] := A[n+1];

(* Assign processes to processors, channels to processes, *)
   (* and array rows to processes *)
  FORALL i := 1 TO n DO
   (@snake[i] PORT upchan[i];downchan[i];inchan[i])
      Updaterow(i, A[i], A[i]);
END.
```

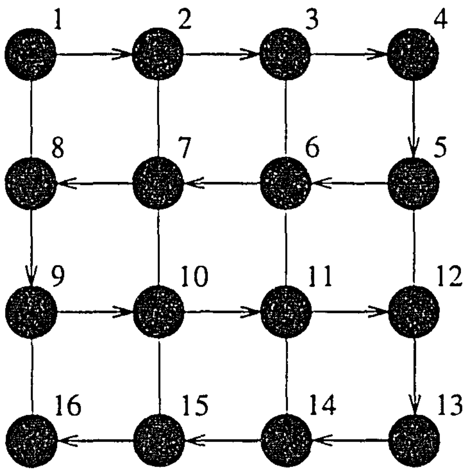
Figure 8. Multi-Pascal program which implements the Jacobi algorithm on a 2-D mesh MIMD machine.

Just to have some data to work on, a section near the end of the program initializes the top and right boundary values for the data grid to 100.0, initializes the left and bottom boundary values to 0.0, and initializes the actual data grid values to 0.0.

To implement static scheduling, the FORALL loop at the end of the program then creates a process containing the Updaterow procedure for each compute node, assigns input and output communication channels to each process, and assigns one row of the data grid to each process. Figure 9a. shows how the actual compute nodes are numbered and Figure 9b shows how processes 1-n are assigned to the compute nodes in a "snake" topology by the FORALL loop. The processes and grid rows are assigned to compute nodes in a snake pattern so that adjacent grid rows are always on adjacent nodes. This minimizes communication distances and congestion because all transfers of new row values require just one hop on the network. Assigning a complete row of data points to each compute node further reduces communication because each process has to send and receive only two messages during each iteration of the Updaterow procedure. If the grid data points were distributed to processes in blocks, then each compute node would have to send and receive four messages to exchange new values with adjacent processes.



a.



b.

Figure 9a,b. Compute node numbering for 2-D Mesh MIMD machine(a), Snake pattern of process assignment to compute nodes(b)

In Multi-Pascal, processes use channels to send messages to each other. At the start of the program in Figure 8, two channel variables, upchan and downchan are declared.

Identifying these channels as rowtype makes it possible to send a message containing an entire row of data points with a simple assignment statement of the form, upchan[me+1] := myrow. The message is received with another assignment statement of the form, uprow := upchan[me].

At the start of the Updaterow procedure, each process first sends its current row values to its two nearest neighbors and receives the current row values from its two nearest neighbors. Each process then computes the new value for all the points along its row and calculates the maximum change between the old value and the new value for all the points in its row.

To check for the termination condition, each process then calls the function Aggregate and passes it a Boolean called mydone which is true if maxchange < tolerance for its last computation iteration. Figure 10a shows how these local my_done Booleans are combined and accumulated at the highest numbered node. Each of the nodes along the top of the mesh sends a message containing its my_done to the next node down its column and waits to receive a message containing the global result. Each node in the next row down ANDs the received my_done with its local my_done, sends a message containing the result down to the next node in its column, and waits to receive a message containing the final result. At the bottom of the mesh, the my_done accumulation moves horizontally as shown until the final result is generated in the highest numbered node. This method of generating a global result is very widely used and is commonly referred to as a minimum spanning binary tree algorithm [Barne93a]. In the case shown in Figure 10a it is specifically a fanin tree.

Figure 10b shows how a minimum spanning fanout tree can be used to broadcast the final Boolean value to all the waiting compute nodes. To start, the highest numbered node sends messages containing the result to two other nodes. Each of these sends messages containing the global value on to one or two other nodes as shown. Eventually, all the waiting nodes receive the global value. As each node receives the global Boolean value, it exits from the Aggregate function and returns the global value to Updaterow. Updaterow uses the returned value to decide whether to loop again or quit.

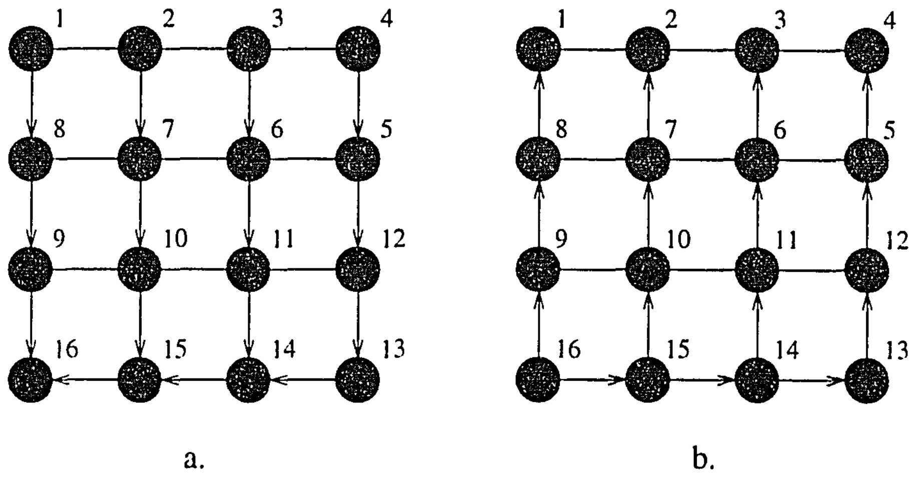


Figure 10a,b. Message flow for global aggregate on 2-D mesh MIMD machine (a), Message flow for global broadcast on 2-D mesh MIMD machine (b)

The fanin-fanout tree method used to accumulate and broadcast the global Boolean here implements both the second barrier and the lock shown in the pseudo-program of Figure 7. The fanin tree emulates the lock by providing controlled sequential accumulation of the global Boolean value. Also, it enforces a barrier because no process can receive the global result until all have contributed. Once the global result is available, the fanout tree efficiently releases all the processors from the barrier. Note that although I ran Boolean values instead of local_max_delta values through the fanin-fanout trees, the result is the same.

One way to reduce the overall time cost of convergence checking in a program such as this is to perform several computation iterations before checking for convergence. In the program in Figure 8, for example, if the value of numiter is changed to 4, then each process will perform four iterations of the computation loop in Updaterow before calling Aggregate. The result is that Aggregate only executes one quarter as many times as it does with numiter = 1. A larger value of numiter would further decrease the overall time contributed by execution of the Aggregate function. The tradeoff here is that if numiter is made too large, then extra and unnecessary computation iterations may be performed before the last convergence check.

A final point to make about the program in Figure 8 is how the Jacobi algorithm requirement that each iteration use only values from the previous iteration is met. The pseudo-program in Figure 7 shows how a barrier can be used to enforce this requirement. In the program in Figure 8, message send-receive operations provide the mechanism which enforces this requirement. During the first iteration in Updaterow each process sends its row values to its two neighbors and receives row values from its two neighbors. After that, when a process finishes one computation iteration and loops around to do the next, it sends messages containing its new row values to its two neighbors and waits until it receives messages containing new row values from each of its neighbors. Since a process cannot proceed with a new computation until it receives these new rows and it cannot receive them until they are sent by the process that produces them, the Jacobi requirement is satisfied.

The preceding program has illustrated how message passing can be used to implement synchronization, accumulate a global value, and broadcast a global value on a mesh-connected multicomputer. In this example we used fanin-fanout trees to accumulate a global Boolean, but the method is very general and can be used to generate and broadcast, for example, the result of a global sum operation. Johnsson [Johnsa] mentions that the global sum operation is an important part of the conjugate gradient method and Clark [Clark92a] shows that this operation is important in molecular dynamics programs. The fanin-fanout tree method can also be used to generate a global MIN, MAX, OR, EXOR, or AND, as required by a particular algorithm. In algorithms where only a one-to-all broadcast is needed, a single fanout tree algorithm can be used. van de Geijn [Geijn91a] points out that this kind of broadcast is an important part of LU factorization of a dense matrix.

In summary, this section has shown minimum spanning trees can be used to implement important global operations such as barrier synchronization, global sum, MIN, MAX, OR, EXOR, and AND. The number of steps required for a minimum spanning binary tree algorithm implemented on a network is equal to the diameter of the network, D. Each step requires a message send and receive operation, so the total time for the algorithm is proportional to the (message send-receive time) x D. If two trees are used to generate and broadcast a global sum, for example, the time is doubled. Later I show that for current machines, the cascaded message send-receive times along the branches of these trees contributes significantly to the overall execution time. Next is a discussion of how these global operations are commonly implemented on a hardware-based distributed shared memory machine.

Program Operations On A Distributed Shared Memory MIMD Machine

Conceptually, a Jacobi program for a shared memory machine follows the pseudo-program in Figure 7 very closely, so I did not write a separate program for it (see Lester [Leste93a] for one way to do it). On a shared memory machine, identical processes can be created on each processor and rows of grid points assigned to each process as in the preceding distributed memory example. Since all of the data are in a shared global address space, each processor can simply copy data values to its local cache as needed. The key points left to discuss then are how the synchronization lock and barriers shown in Figure 7 can be implemented on a distributed shared memory machine.

As mentioned earlier, a lock is a programming construct which allows only one processor at a time to access a shared resource. The simplest type of lock, commonly called a spin-lock, uses a lock variable which contains a 0 for the unlocked condition and a 1 for the locked condition. To gain access to the shared resource each processor uses an indivisible read-write operation to read the value of the lock variable and write a 1 to it. If the value read was a 0, the lock was unlocked, so the requesting processor will be granted access to a shared resource such as global_max_delta. The 1 written to the lock variable as part of the indivisible read-write operation will mark the lock as locked for other processors that access the lock variable. When a processor completes its use of the shared resource, it resets the lock variable to indicate the lock is open. If a processor reads the lock variable and finds it a 1, it must read the lock variable over and over or in other words "spin on the lock variable" until it reads a 0.

A common way to implement a simple barrier on a distributed shared memory system is with a count variable protected by a lock. At the start, the count variable is loaded with the number of processes that need to check in at the barrier. When each processor reaches the barrier, it spins until it is able to access the count variable through the lock, decrements the count value by one, and then releases the lock. When all the processes have checked in at the barrier, they are allowed to spin on a second lock which protects the count variable. To check out of the barrier, each process must acquire this second lock, increment the count variable, and release the lock.

One major problem here is that all of the processors are trying to access the lock variable at the same time, so the lock variable memory location becomes a "hot spot" of contention. The traffic to and from this "hot spot" may interfere with other memory accesses. A second problem is that successful acquisitions of the lock are of necessity sequential since only one processor at a time can hold the lock. Since each processor must acquire the lock twice, the total time for the barrier is equal to 2 x N x (the lock acquisition-release time), where N represents the number of processors. During this time processors spend most of their time spinning on the lock. Unless there are other processes available and the processors can efficiently swap to them, this spinning time is wasted.

One method of decreasing the contention for a single lock variable is to use a binary tree of lock variables. Processors at the leaves of the tree compete in pairs for a local lock and decrement a local count variable through it. When both have reached the local barrier, one of the processes is allowed to compete with another "local winner" for a lock at the next level up the tree. When all processes have reached the barrier, competition will reach the top of the tree. The tree can be used in the reverse direction to release all the processors from the barrier. In this case the time for the barrier is proportional to 2 x log N x (acquisition time). Although this method reduces the contention for a single lock variable and perhaps reduces the load on the cache coherency mechanism, it requires additional software overhead and still requires considerable spinning.

Still another way to reduce the contention for a lock such as this is to set up a queue of processes waiting for the lock. When one process unlocks the lock, it is automatically granted to the next process in the queue. The tradeoff here, of course, is the software complexity and time overhead of setting up and managing the queue.

Referring back to Figure 7, the first barrier can be be implemented as just described. Since a barrier contains a lock, the lock and the second barrier shown in Figure 7 can be combined. As each process unlocks the lock and accesses the count variable, it can also update global_max_delta as needed. Each process can read the final global_max_delta value during the barrier exit phase. Other global operations such as sum, MIN, OR, EXOR, and AND can be performed using the same approach.

In summary, this section has shown how access synchronization, barriers, and other global operations can be performed on a distributed shared memory machine. Later I show how the COP system can substantially reduce the time required to implement these operations. Next is a discussion of the additional requirements for executing multiple independent programs on a MIMD machine.

Additional Requirements for MPMD Execution on a MIMD Machine

In the preceding sections I described operations commonly needed for a multicomputer and for a multiprocessor executing in Single Program Multiple Data(SPMD) mode. In addition to these operations, a machine executing in Multiple Program Multiple Data (MPMD) mode has several other requirements.

First and foremost of these requirements is that each executing program or user must be assigned a set of processors which is reserved for its exclusive use. This process is commonly referred to as setting up a virtual machine for each program or each user. The usual way to assign a group of processors to a user is with an operating system call. On an Intel iPSC/2 [Corpoa], for example, the getcube (N) command instructs the NX/2 operating system to assign a group of N processors to the requesting user.

The assigned set of processors must also be protected from interference by any other programs executing on other processors. This need for protection makes programming and program execution considerably more complex. For example, on a message-passing machine each message sent by a user program must be checked to make sure that its destination is one of the processors assigned to that program. This check is usually performed by the operating system. For example, when a program on an Intel iPSC/2 uses the csend(type, buf, length, node, pid) system call to send a message to another processor, the NX/2 operating system checks to make sure the destination node is within the group assigned to that program and signals an error if it is not.

The software overhead of this checking adds to the time required for each message send-receive operation and thereby increases the overall program execution time. As will be discussed in the next chapter, the protection required for COP system operations is implemented with fast hardware to minimize its effect on execution time.

Initial Cost Analysis For Global Operations On MIMD Machines

Preceding sections of this chapter have demonstrated some programming constructs and global operations which are important in programs for message-passing multicomputers and in programs for distributed shared memory multiprocessors. These include: synchronization of access to shared variables; barrier synchronization to assure that all processors have reached a certain point before any are allowed to proceed; global operations such as sum, MAX, MIN, OR, AND, EXOR; and broadcasting one-to-all, or all-to-all. Also important are protection capabilities which allow assigning virtual machines for MPMD execution. In this section I show an initial analysis of the cost of these operations on multicomputers and distributed shared memory multiprocessors, then I make some projections on the effect these costs have on overall execution time and on the optimum number of processors that can be applied to a particular parallel task. These projections are the rationale for the COP system which is described in the next chapter.

Cost Analysis for Global Operations on "Big Iron" Multicomputers

Several of the current, massively parallel multicomputers use single, high-bandwidth networks for communication between compute nodes. Compute nodes pass messages on this interconnection network for all data transfers, synchronizations, global operations, etc. On a message-passing-only machine, assuming no contention for network channels, the time required to send a message has two main components. The first, commonly called the latency time or t_L, includes the message startup time, the time required to packetize the data to be sent, the time to add a header which specifies the destination for the message, the time to establish a path through the network, and the time to de-packetize and process the message at the receiving node.

The second component of the time to send a message is the total source to destination hardware transmission time, $M_B * R_T$, where M_B is the number of bytes in the message and R_T is the reciprocal of the network message bandwidth.

The total time to send and receive a message then is $t_L + M_B * R_T$. According to Rosing [Rosin94a], the Intel Paragon multicomputer currently has a minimum t_L of 25 microseconds and an R_T of 1/200 Mbytes/sec. Using these values, the time to send and receive an 8 byte message is 25 microseconds + 40 nanoseconds. For very large messages the latency term is swamped by the transmission time term, but for small messages the latency term clearly dominates.

As described earlier in the chapter, synchronization and other global operations are often implemented with fanin and fanout minimum spanning trees. New messages are generated at each level in these trees, so message-passing times cascade along the branches of the trees. Since the number of levels in either tree is equal to D, the diameter of the network, the total message passing time for one of these trees is equal to $D(t_L + M_B * R_T)$.

Using the Intel Paragon numbers of $t_L = 25$ microseconds and $R_T = 1/(200$ Mbytes/sec), this equation gives a time of about 350 microseconds to broadcast an 8-byte value from one node to the other 63 in a 64 node square mesh. For a 16 x 32 node mesh, the equation gives a one-to-all broadcast time of 1200 microseconds or 1.2 ms. This computed time agrees with a measured time recently reported by van de Geijn [Geijn94a] for a 16 x 32 node Paragon using the new InterComm Collective Communication Library. Barriers and similar global operations require traversal of two trees, so the times for these operations are about twice the time for a simple one-to-all broadcast.

Since current compute node processors can execute an instruction in 20 ns or less, the cascaded message times that a compute node must wait for a global result represents many thousands of instruction times. This time is wasted unless the program has enough parallelism to support another execution thread and the thread swap overhead is low enough to justify a swap.

Furthermore, if the number of compute nodes is increased in an attempt to exploit fine-grained parallelism in an application, the diameter of the network will increase. This will further increase the time required for each global operation and decrease or possibly eliminate the potential gain expected from using a greater number of compute nodes. Driscoll and Daasch [Drisc95a] have shown, in an extension of the work of Amdahl [Amdah67a] and Gustafson [Gusta88a], that if the serial component of an application increases linearly with the number of compute nodes, the optimal number of compute nodes can be represented as:

$$N_{OPT} = \sqrt{\frac{Par_0}{a_S}}$$

where Par_0 represents the time to execute the parallelizable portion of the application on a single machine, and a_S is the serial component of the application.

The number of levels in a binary spanning tree increases logarithmically with the number of processors and therefore the serial component introduced by cascaded message times increases logarithmically with N rather than linearly. However, this expression clearly shows that if global operations which use these trees are a significant part of the application, as they are in many scientific computations, then the sequential time represented by the cascaded message latencies limits the number of compute nodes which can be effectively utilized in the computation.

Cost Analysis for Global Operations on Cluster Multicomputers

For current workstation cluster multicomputers the time costs of global operations are even greater than those for "Big Iron" machines, because the message-passing latency times are much longer. Steenkiste [Steen94a] reports that the measured time for a single 16-byte message on the Nectar cluster system is 97 to 234 microseconds, depending on the protocols used. Cohen [Cohen94a] reports that the latency between user memory on one node and user memory on another node in the ATOMIC cluster system is about 1500 microseconds. The main reason that these times are so long is that high level operating system protocols are used to send and receive messages. This is done to maintain protection, implement error detection, and maintain software compatibility. The point remains, that since global operations require cascaded messages, they are very expensive time-wise on current cluster multicomputers.

Cost Analysis for Global Operations on Distributed Shared Memory Multiprocessors

If many processors in a distributed shared memory system are trying to access a cached lock variable at the same time, the synchronization variable may "ping-pong" around the system caches. This is a worst-case scenario for most cache coherent systems, because after each write, the result will be dirty in a remote cache. As cited in Chapter II, Lenoski [Lenos93a] notes that on the DASH system this type access requires 132 clock pulses as compared to only one clock pulse for a local cache access. Even if the lock variable is not cached, each access is still remote and takes appreciably more time than a local cache access.

Since accesses to a lock variable for implementing a barrier, a reduction, or some other global operation are of necessity sequential, the access times are additive. If a single lock variable is used, Lenoski [Lenos93a] has shown that the total time is a linear function of the number of processors. If software combining trees are used to implement these operations as described in an earlier section of the chapter, the total time in O(log N).

In either case, these operations introduce a sequential time component which increases with the number of participating processors and, as previously cited from Driscoll and Daasch, strongly affects the optimum number of processors which can be applied to a particular task. Also, as discussed earlier in the chapter, most of this sequential time is wasted spinning on locks.

Chapter Summary And Conclusions

Global operations introduce a substantial sequential time component into computations on both multicomputers and distributed shared memory multiprocessors. As stated in Chapter I, my hypothesis is that the addition of a high-speed secondary interconnection network with a wide tree topology and one or more Coordination Processor(s) or COP(s) to either type of machine could reduce the time required these operations by as much as two or three orders of magnitude. In my research I found no other multicomputer or multiprocessor enhancements that give the combination of speed, versatility, and cost effectiveness potentially provided by the COP system.

In the next chapter I describe in detail the architecture, operations, and software interface of the COP system. Then in Chapter V, I do a detailed performance analysis of the COP system, compare the performance of a COP enhanced system with the reported performance of current research and commercial machines, and show some examples of the overall program speedup provided by the COP system. With the picture of the COP system complete, I then in Chapter VI describe and compare other attempts to improve global operation performance.

CHAPTER IV

COP SYSTEM ARCHITECTURE, OPERATIONS, AND PROGRAMMING

Goals

The goals of the COP system were:

1. Be applicable to "Big Iron" multicomputers, workstation cluster multicomputers, and distributed shared memory systems.
2. Improve the efficiency of a wide variety of common parallel programming operations so as to better justify the cost of implementation.
3. Retrofit easily to the hardware of current generation machines so that it would not be necessary to wait for the next generation of machines to gain the benefits.
4. Require minimum modification of existing programs and programming paradigms so as to not waste the massive efforts that have been invested in them.
5. Be compatible with MPI, PVM, BLACS, and other current efforts to insulate programmers from low level system details.
6. Be compatible with advances such as thread scheduling and object oriented parallel programming that are likely to be included in future machines.
7. Have a high benefit-to-cost ratio.

As I discuss the architecture and operation of the COP system in the following sections, I will point out some of the decisions made to help meet these goals.

Top Level View

Figure 11 shows the network topology for a single level COP system. As shown, each compute node in a group of 64 is connected to a coordination processor (COP) by an independent high-speed, half-duplex serial data link. Upon seeing 64 compute nodes connected to one COP, the first word that probably comes to mind is "bottleneck". However, as I show later, it is the centralized nature of the COP which helps provide the benefits of the system and the COP is not a bottleneck for its intended operations. Everyday life provides many examples of the benefits of a balanced, centralized system such as this. An airport control tower, for example, seems a better way to control plane landings than having each plane attempt to land on its own and back off each time a collision seems imminent.

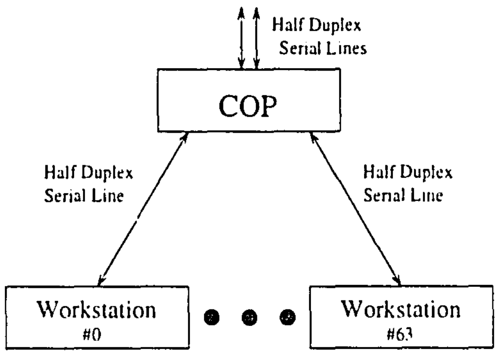


Figure 11. Single level COP system topology

Since the communication links between compute nodes and a COP are independent, all the compute nodes in a group can send data words or synchronization signals to their COP simultaneously. With these dedicated direct links, the source and destination are hardwired, so no complex message formatting is required. To send a word to its COP, a compute node simply does a write to its COP network interface port. The dedicated signal lines also mean that no time is required to establish a connection with the COP, and there is no network contention. The result of these capabilities is that each compute node can transmit a synchronization signal or data word to its COP in a very short constant time. The independent communication links also mean that a COP can broadcast a synchronization signal or data value to all the compute nodes in its group simultaneously.

Bit-serial data transmission was chosen to minimize the number of conductors in each link and for compatibility with relatively inexpensive, non-multiplexed fiber-optic data transmission.

The decision to assign 64 compute nodes to each COP was made partially so masks, bit-vectors, etc. are compatible with the data path widths of the latest compute node processors. Based on my personal programming experience, this reduces "bit-twiddling" and thus simplifies the software interface with the COP. Also, assigning 64 compute nodes to each COP means that a two level hierarchy of COPs can service up to 4096 compute nodes as shown in Figure 12. Keeping the number of levels low reduces the number of COPs and the number of connecting links for a given size machine. Keeping the number of levels low also reduces the overhead involved in traversing the tree for global operations in which a large number of compute nodes participate. To broadcast a value to all 4095 other nodes, for example, compute node 0 sends the value to COP 0, COP 0 sends the value to the Level 2 COP, the Level 2 COP broadcasts the value to all the Level 1 COPs, and each of these COPs broadcasts the value to its 64 compute nodes. The whole process requires only three passes though a COP level.

A very important point here is that the topology of the COP system is independent of the topology of the underlying machine. This means that the COP system is equally applicable to "Big Iron" multicomputers, cluster multicomputers, and distributed shared memory multiprocessors. Note that the COP system will be most efficient if a particular physical partition or "virtual machine" is created with all its compute nodes connected to one COP, but this is not required. In this case only the level 1 COP is used for all

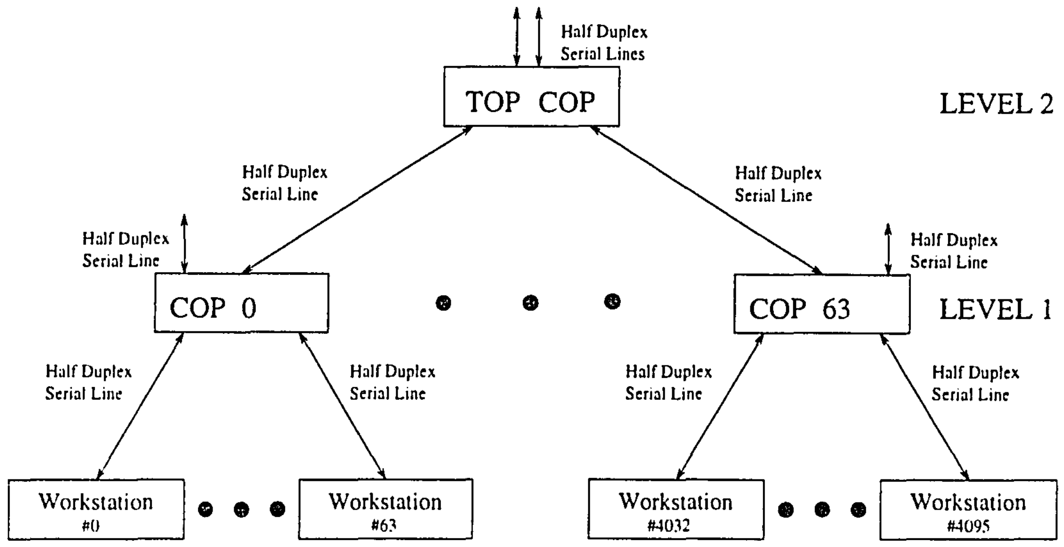


Figure 12. Two level COP system topology

operations within the partition. For a two level system it is somewhat more efficient to assign the compute notes of a partition to level 1 COPs that are connected to adjacent input channels on the level 2 COP, but again, this is not required.

The software receive latency for a COP broadcast is usually very low because the receiving node is waiting for the control or data word and immediately reads it from the COP network interface as soon as it arrives. In the case of a global sum operation, for example, a compute node would most likely write its data value and the appropriate op code to its COP interface port, poll the COP interface port Data Ready strobe until the global sum arrives, and then immediately read the sum from the port.

Each COP has two extra serial channels in addition to those used to connect to 64 compute nodes or to lower level COPs. One of these is used to connect to a higher level COP if present. The second extra serial channel can optionally be used to export performance and debugging data to external recording equipment.

A 64-bit integer ALU in each COP is used for performing global integer sums, MIN, MAX, bitwise AND, bitwise OR, and bitwise EXOR operations. A Floating Point Unit in each COP is used for performing global floating point sum, MIN, and MAX. Each COP also contains a bank of very fast SRAM which is used to hold intermediate results and shared write-able variables, to function as a global name space, and to accumulate performance data. A second, smaller bank of RAM holds broadcast/multicast masks. A third, small bank of RAM holds bit vectors which identify the compute nodes participating in a barrier or other global operation. As a brief, introductory example of how a COP system works, we will use a global sum operation.

To start, each compute node writes a command consisting of a data value and the appropriate op code to its COP network interface port. The COP network interface controller then automatically transmits the command to the COP. Arrival of a command at the network port of a COP sets a DATA_RDY flag for that input channel. The COP cycles through the input channel service requests on a round-robin basis. When the COP services a channel, it adds that channel's contribution to the intermediate result and resets the appropriate bit in the bit vector which identifies the compute nodes participating in the operation. When all the compute nodes have contributed, the COP broadcasts the sum simultaneously to all the participating compute nodes.

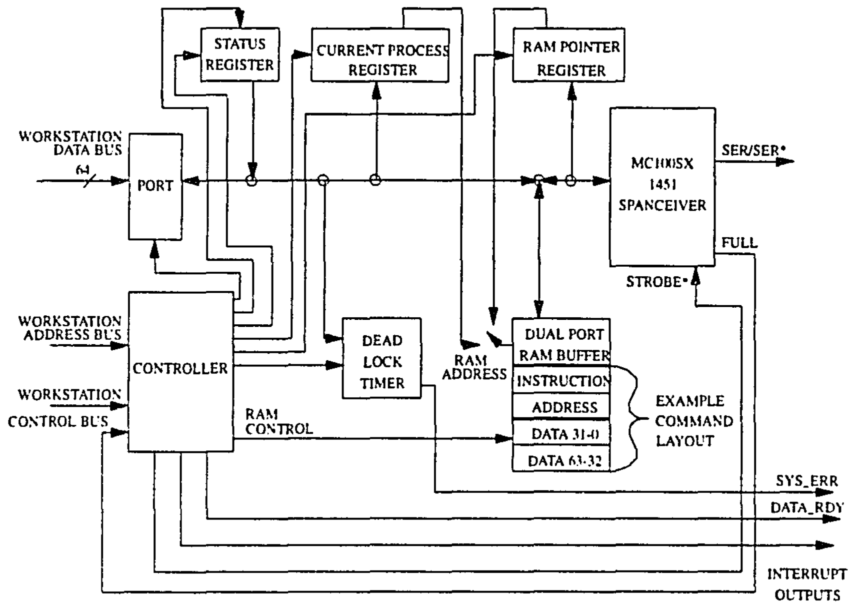
Operations that a COP system can directly perform include: synchronized read-modify-write access to global variables; barriers; integer and floating point global sum, MIN, MAX; bitwise AND, OR, EXOR; one-to-all broadcast or multicast; and all-to-all broadcast or multicast. As mentioned in the last chapter, the COP system includes hardware-based protection mechanisms so that it is compatible with MPMD execution as well as with SPMD execution.

The Compute Node To COP Network Interface

Figure 13a shows a block diagram of the compute node to COP network interface. This interface is basically just a serial port with receive buffering, deadlock detection, and virtual machine protection capabilities. The compute node can interact with the interface on either a polled or an interrupt basis. To the compute node the interface appears simply as a 64-bit read/write port.

Each communication between a compute node and a COP consists of one to four 32-bit words as shown by the command layout example in Figure 13a. The first word in a COP command is always an instruction word with the format shown in Figure 13b. Depending on the particular command, this instruction word is inserted by a user instruction, an operating system command, or hardware. The U/S bit in the instruction word indicates whether the command is a user level command or a supervisor level command which can only be invoked with an OS call. The PPN in the instruction word is a number which identifies the physical partition to which the processor has been assigned. All the compute nodes assigned to a particular user will have the same PPN. The X bit is used to indicate whether the physical partition extends beyond the local, Level 1 COP. In other words, the X bit specifies whether a COP command should be passed on to a level 2 COP and applied to more than one level 1 COP. The PID in the command word includes context, group, and rank numbers which identify the process sending or receiving the command. This process identification mechanism was chosen for compatibility with the Message Passing Interface Standard. [Tennea]

The OP Code bits in the instruction word specify the operation to be performed. The MASK bit in the instruction word is used to select one of two programmable masks in the COP Mask RAM. One of these masks might be programmed for a one-to-all broadcast to all the other compute nodes in the partition, and the other mask might be programmed for



(a)

U/S - 1	X - 1	OP CODE - 7	MASK - 1	PPN - 7	PID - 7	UNUSED - 8

(b)

Figure 13a,b. Compute node to COP network interface circuitry(a), format for COP instruction word showing number of bits for each field(b)

a multicast to some subset of the compute nodes in the partition. Additional bits in the instruction word are reserved for future use.

If required, the second word in a command contains an address which is used to access a global shared variable or partial result in the COP Data RAM. The third and fourth words in a command are used for 32-bit data values, 64-bit data values, or 64-bit mask values.

From the compute node, the COP network interface appears as a read/write port with two different and separate addresses. One address maps the port into protected I/O or memory space where it can only be accessed through the operating system. This access is used to initialize broadcast and barrier masks in the COP, change the value in the Current Process Register, initialize the deadlock timer, and perform various tasks during a process swap. For this type of access, the value in the U/S bit in the instruction word is determined by the bit written to the port in that position.

The second interface port address is mapped into the user addressable memory or I/O space and allows user programs to directly access the COP. When the interface is accessed through this address, the U/S bit is hardwired with the User value so there is no chance a user program can mistakenly issue a Supervisor level command.

During system startup the operating system writes a physical partition number and the PID for the first process to the Current Process Register, and writes an initial count to the deadlock timer. When a user writes a command to the interface, the PPN and PID numbers are automatically inserted in the instruction word from the Current Process Register as the word is transferred to the Dual-Port RAM Buffer. This hardware mechanism assures that a command is linked with the currently executing process. During a process swap, the operating system writes the PID number for the new process in the Current Process Register.

If a command consists of more than one word, the additional words are transferred to the Dual-Port RAM Buffer as they are written to the interface by the compute node. However, as soon as an instruction word is written to the buffer, the interface controller transfers it to the UART and the UART automatically sends the word on to the COP. Additional words of a command are transferred to the UART and sent to the COP in sequence.

The actual UART sections of both the compute node to COP network interface and the network to COP interface use Motorola MC100SX1451 Autobahn Spanceivers [Produ93a] rather than custom modeled devices. These devices not only fill a need in the COP system, but also demonstrate that 200-400 MByte/sec serial transmission is possible with currently available commercial technology.

The Spanceiver serializes each 32-bit word and transmits it over a Positive Emitter Coupled Logic (PECL) differential transmission line. Current versions of the device transmit at a programmable data rate of up to 200 Mbytes/sec. Second-silicon versions are expected to work at up to 400 Mbytes/sec. High quality triaxial cable can be used to connect Spanceivers that are within 10 feet of each other. [Blood88a]. For longer distance connections between Spanceivers, the PECL signals can be converted to non-multiplexed optical signals and transmitted over relatively inexpensive multi-mode fiber optic cables. The second-silicon version of the Spanceiver, in fact, includes fiber optic interface circuitry.

Using bit-serial data transmission for the initial design also provides a worst case test of the feasibility of the COP system from a timing standpoint. Later versions might use byte-serial or word-serial transmission if justified by the benefit/cost ratio. Recent research by Smith [Smith94a], indicates that it will soon be feasible to use parallel ribbon fiber interconnects between multicomputer cabinets over distances up to a few tens of meters.

Figure 14 shows a block diagram for the Spanceiver transmit and receive sections. Parallel data words to be sent are written to the Spanceiver transmit register using a FULL/STRB signal handshake protocol. Once a 32-bit word is written to the transmit section of the spanceiver, the word is automatically serialized and sent out on the ser/ser* differential outputs.

When the Spanceiver is not transmitting, it monitors the ser/ser* pin for signal activity. In response to a detected signal transition the receiver section automatically synchronizes on the incoming data, shifts the data bits into the SIPO register, and transfers the received word in parallel to the receive buffer register. Since the Spanceiver can only buffer one 32-bit word internally, each word is transferred to the external RAM buffer. As

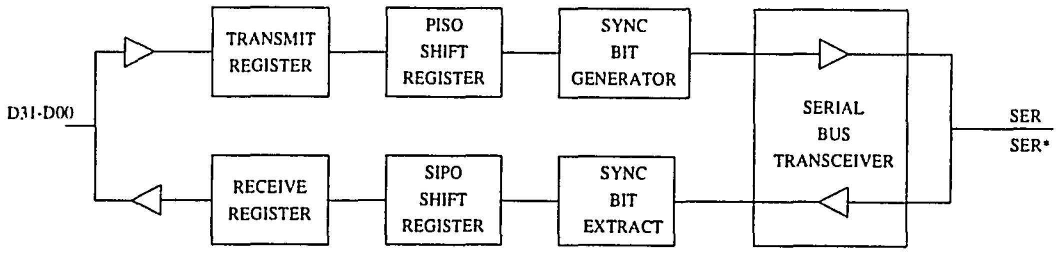


Figure 14. MC100SX1451 Autobahn Spanceiver transmit and receive blocks

an instruction word is transferred from the Spanceiver, the PPN and PID are extracted from it and written to the Buffer Pointer Register. The contents of this register are used as part of the address for the RAM location where the received command will be written.

After all the words for a received command are written to the RAM Buffer, the interface controller compares the Current Process Register with the Buffer Pointer Register to determine if the received message is intended for the current process. If so, the controller sets a bit in the interface status register and/or asserts an interrupt signal to the compute node. If the command is intended for a swapped out process, the interface controller may immediately set a status bit and/or generate an interrupt signal if programmed to do so, or it may wait until the destination process is swapped back in to do this. The key here is that when a process is swapped back in, the interface controller checks the RAM buffer to determine if there are any COP commands waiting for the process. This mechanism assures that a command returning to a process from a COP will not be lost if it arrives at the interface while the process is swapped out.

If a Spanceiver detects any errors while receiving a data word, it will assert its Error signal. Later, after I discuss the operation of the COP end of the data link I will describe the protocol that the COP system uses to handle errors detected by the Spanceiver.

When the compute node accesses the COP port to read the received message, the contents of the Current Process Register are used as a pointer to the RAM Buffer. This further assures that a compute node can only read a message intended for the currently executing process.

The deadlock timer on the interface can be used to trap to the operating system if a compute node sends a command to its COP and does not receive a reply within some programmable time interval.

In summary, the compute node to COP network interface provides fast data transfer with the protection features required for virtual machine operation. The relative simplicity and standard bus connections of this interface allow it to be implemented as a small daughter board which can be added to an existing compute node for performance enhancement or can be easily included in a new system design.

COP Architecture And Operations

The COP to COP Network Interface

Figure 15 shows a block diagram of a COP. Each of the COP to network interfaces has a Motorola MC100SX1451 Spanceiver, four 32-bit registers for buffering words received from a compute node, four 32 bit registers for buffering words to be sent to a compute node, and a mini-controller. The first register in each set is an instruction register. The second register in each set is used for Data RAM addresses which identify global shared variables or partial results. The third and fourth registers in each set are used for 32-bit or 64-bit data words and 64-bit masks.

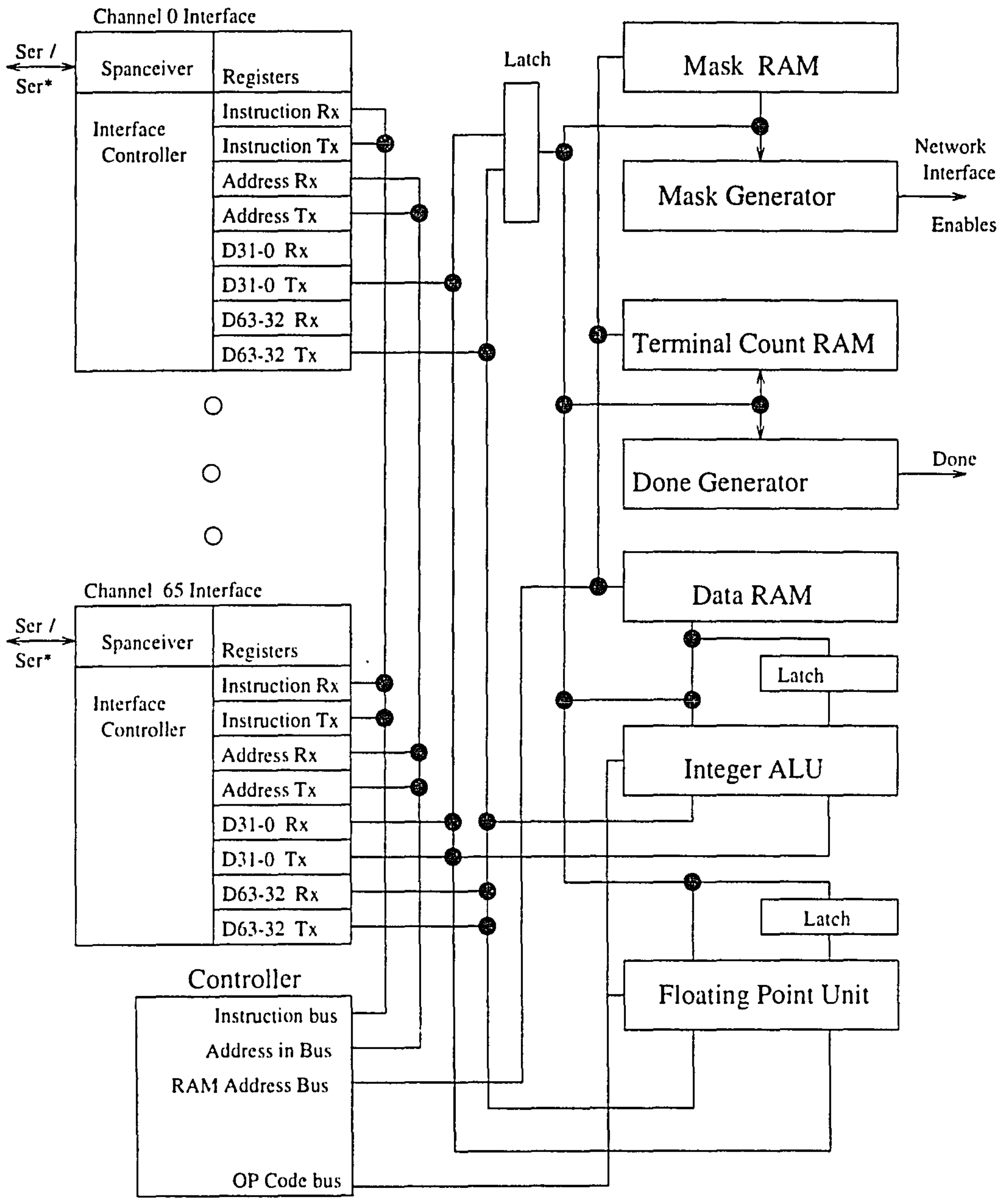


Figure 15. Block diagram of a Coordination Processor

When the first word of a command arrives at the COP board, the interface controller transfers the word to the first buffer register, and extracts two bits which specify the number of words in the command. As additional words are received, they are transferred to the appropriate buffer register.

If the Spanceiver detects an error while receiving a word, it will assert its Error signal. If this signal is asserted, the interface controller aborts the receive operation and, as soon as the Spanceiver is available, writes a "resend" command to the Spanceiver for transmission back to the compute node. In response to a resend command, the compute node interface controller resends the entire command which is still in the RAM buffer on the compute node interface. After some number of unsuccessful attempts to receive a message from a compute node, the buffer controller sends an error word which causes a trap to the operating system on the compute node. A major advantage of this approach is that the compute node to COP link can cycle through multiple attempts to deliver a message without involving the COP controller. This reduces the COP controller overhead.

When all of the words of a command have been received without errors, the COP interface controller asserts a DATA_RDY signal. The main COP controller polls the DATA_RDY signals of the 64 network interfaces on a round-robin basis and services ready interfaces in sequence. As soon as the COP controller reads a command from a ready interface, the interface controller writes an Acknowledge word to the Spanceiver for transmission back to the compute node. Arrival of this Acknowledge word at the compute node indicates that the receive registers on the COP end of the link are available. Requiring that the compute node interface wait for this Acknowledge prevents overwriting the receive buffers on the COP and assures that a command is still available in the compute node interface RAM buffer for resending in case of an error.

To send a command to a compute node, the COP controller transfers the command, address and data components of the command in parallel to the four transmit buffer registers in the interface. The interface controller then transfers the buffered words to the Spanceiver in sequence for transmission. If the compute node interface detects an error in a received word, it will direct the COP interface controller to resend the command. After some number of unsuccessful communication attempts, a trap to the operating system is generated. Again, placing the responsibility for reliable communication on the interface controllers rather than on the main COP controller removes the overhead of error handling from the main COP controller and thereby improves its efficiency.

Another important point here is that the COP network communication links are asynchronous. This means that no global clock is required and that cables do not have to be cut to specific lengths in order to synchronize transmitters and receivers. This makes it easier to use the COP system with cluster multicomputers.

Overview of COP Operations

As mentioned previously and as also shown in Figure 15, a COP contains a 64-bit integer ALU, a double precision Floating Point Unit, three banks of 64-bit wide, very fast SRAM, and a hard-wired controller. The Data RAM can be used to hold shared writeable variables, hold intermediate computational results, function as a global name space, and accumulate performance data. The Mask RAM holds programmable masks which are used to enable the desired output channels during broadcast and multicast operations. The Terminal Count RAM holds the bit vectors that are used to keep track of which compute nodes have participated in a global operation. In this section I will give an overview of how these parts work together during various COP operations and in following sections discuss in detail the operation of each major block of COP circuitry.

As a first example of how a COP operates, suppose that one compute node needs to broadcast a data value to all or a subset of the other nodes in its partition. To do this the compute node sends the data word and the appropriate instruction word to its COP. When the COP controller reads the command, it will use the PPN, PID, and a couple of other bits in the instruction word as a pointer to the Mask RAM. The mask read from this RAM will enable the transmit buffers of the channels that are to receive the broadcast data word. After the transmit buffers are enabled, the controller writes the data word to all of them simultaneously. The interface controllers then transmit the data word to all the destination compute nodes at the same time.

If the partition is larger than can be serviced by a single COP, then the local, Level 1 COP forwards the command on to the Level 2 COP. The level 2 COP uses a mask in its Mask RAM to broadcast the command to the appropriate Level 1 COPs and each of these then uses a mask from its Mask RAM to broadcast the data value to the desired compute nodes.

To enable a compute node to efficiently broadcast vectors longer than the eight-byte maximum for a single broadcast command, the COP controller has a channel lock capability. When a COP controller receives a lock command, the round-robin servicing of input channels is disabled so the controller continues servicing the locked channel until it receives an unlock command. This feature allows the node associated with the locked channel to pipeline back-to-back sequences of words through the COP.

Two protection mechanisms are available for broadcast operations. First, an OS call is required to write a mask to a Mask RAM, so the OS can check that a user program is not attempting to write an illegal mask. Second, using the PPN and PID to access a mask assures that the mask belongs to the currently executing process.

The COP system can also be used to implement a barrier very efficiently. As mentioned earlier, the Terminal Count(TC) RAM in the COP is used to hold bit vectors which identify the compute nodes participating in global operations such as barriers, reductions, etc. Each bit in one of these vectors corresponds to one of the attached compute nodes. To maintain protection, an operating system call must be used to write a bit vector in one of the TC RAM locations.

When each participating compute node reaches the barrier, it sends a single 32-bit command word to the COP. In response to this word the COP resets the corresponding bit in the barrier bit vector and determines if all the bits are reset. If all the bits in the barrier vector are reset, the Done signal is asserted. In response to the Done signal, the COP controller writes a barrier exit command word to all the network interfaces which are enabled by the corresponding mask from the Mask RAM. The barrier exit command is thus broadcast to all the participating compute nodes simultaneously rather than sequentially.

Global sum and other similar global operations can also be performed very efficiently by a COP system. For this operation each compute node sends a contribution to its COP. The COP adds each contribution to a partial result stored in a Data RAM location. When all the values have been added, the COP uses a mask from the Mask RAM to broadcast the result to the participating compute nodes. For protection, the PPN and PID in the instruction word are used as part of the address for the temporary result in the Data RAM and for the mask in the Mask RAM. Note that each intermediate result could be broadcast to all or to a subset of the participating compute nodes at the same time as it is written back to the Data RAM, if this were required by the particular algorithm.

Still another type of operation that a COP system can easily perform is global shared variable access. For simple read access, a compute node sends the appropriate command and a variable identifier (address) to its COP. The COP uses the variable identifier, the PPN, and the PID received from the compute node to address the desired location in its Data RAM and sends the addressed data value back to the compute node. In a case where it is important that a compute node very quickly read a series of values from the Data RAM or write a series of values, the channel lock feature can be invoked.

Read-Modify-Write access to the COP Data RAM is essentially the same, except that as the value read from the Data RAM is being copied to an interface transmit buffer, it is passed though the ALU, modified as specified in the command, and then written back to the Data RAM.

With the overview of COP operations fresh in mind, let's now take a closer look at the Mask Generator, Terminal Count, ALU/FPU, and Controller blocks in a COP.

Operation of the Mask Generator Block

Figure 16 shows a detailed block diagram of the Mask Generator block. This block has three major functions. The first function is to supply a 1-of-64 signal which enables one network interface unit at a time for servicing. The second function of this block is to supply masks which enable the desired channel interfaces during broadcast and multicast operations. The third function is to produce the XMIT_READY signal when all the transmit interfaces specified by a particular mask are available.

The CHAN DECODE block in the lower left corner of Figure 16 decodes a 6-bit count from the controller to produce the 1-of-64 signal needed to enable a single network interface unit for servicing. This 1-of-64 signal is also used to enable an interface unit for returning a Data RAM value to a requesting compute node. In either case, the CHAN DECODE signal is passed through the 64 x 2:1 multiplexer in the lower right corner of Figure 16 so the enable signal(s) can be asserted at the correct times.

During a broadcast or multicast operation the specified mask is read from the Mask RAM and at the correct time passed through the 2:1 multiplexer to enable the desired interface units. The mask read from the MASK RAM is also applied to the XGEN circuitry where it is compared on a bit-by-bit basis with the TX_EMPTY signals from the interface units. If the transmit buffer registers are empty on all the interfaces specified by the mask, then the XMIT_READY signal will be asserted. In response to this signal, the controller enables the 2:1 multiplexer to pass the mask on to the enable inputs of the interfaces. If one or more of the interfaces is still working on transmitting data words from a preceding operation, the COP controller will simply insert wait states until the transmission(s) are complete and the XMIT_READY signal is asserted.

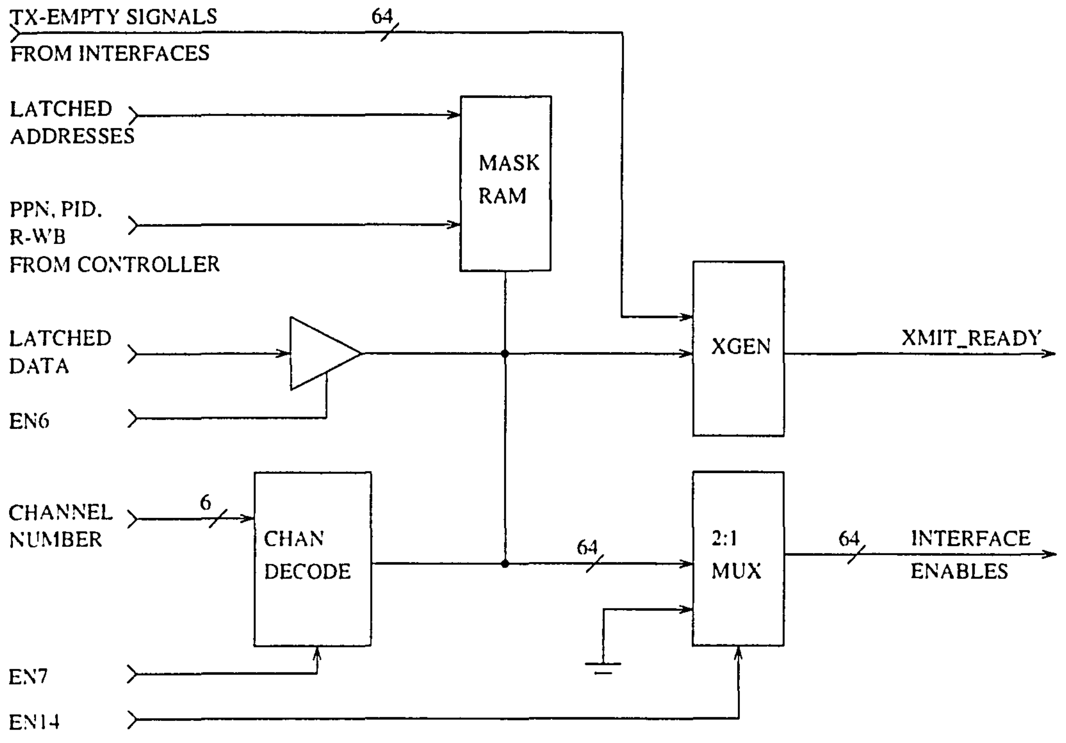


Figure 16. Block diagram of the Mask Generator Block

Operation of the Terminal Count Detector Block

As indicated by its name, the function of the Terminal Count (TC) Detector block shown in Figure 17 is to keep track of which nodes have participated in a particular global operation and generate a Done signal for the controller when all have participated. Nodes participating in a particular operation are represented by bits in a bit vector stored in the TC RAM. A bit vector is initialized with a specific Write-to-TC-RAM COP command.

When a node participates in a global operation related to that bit vector, the bit vector is transferred to the DONE REGISTER. There the bit corresponding to the participating node is reset, and the resultant bit vector is written back to the TC RAM. If the resultant bit vector contains all zeros, the DONE signal is asserted. In response to this signal, the controller transfers the result to the network interface units enabled by the corresponding mask from the Mask Generator Block.

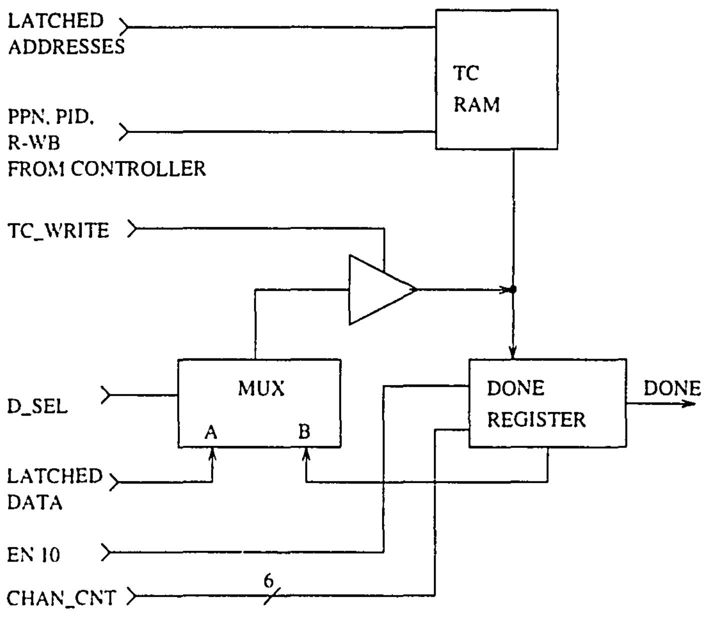


Figure 17. Block diagram of Terminal Count Detector block

Operation of the ALU and FPU blocks

Figure 18 shows a more detailed look at the ALU block of a COP. The overall architecture of the Floating Point Unit block is the same as that of the ALU block and, except for timing, the operations are essentially the same as those for the ALU block. Therefore, to avoid redundancy, I will just describe the ALU block.

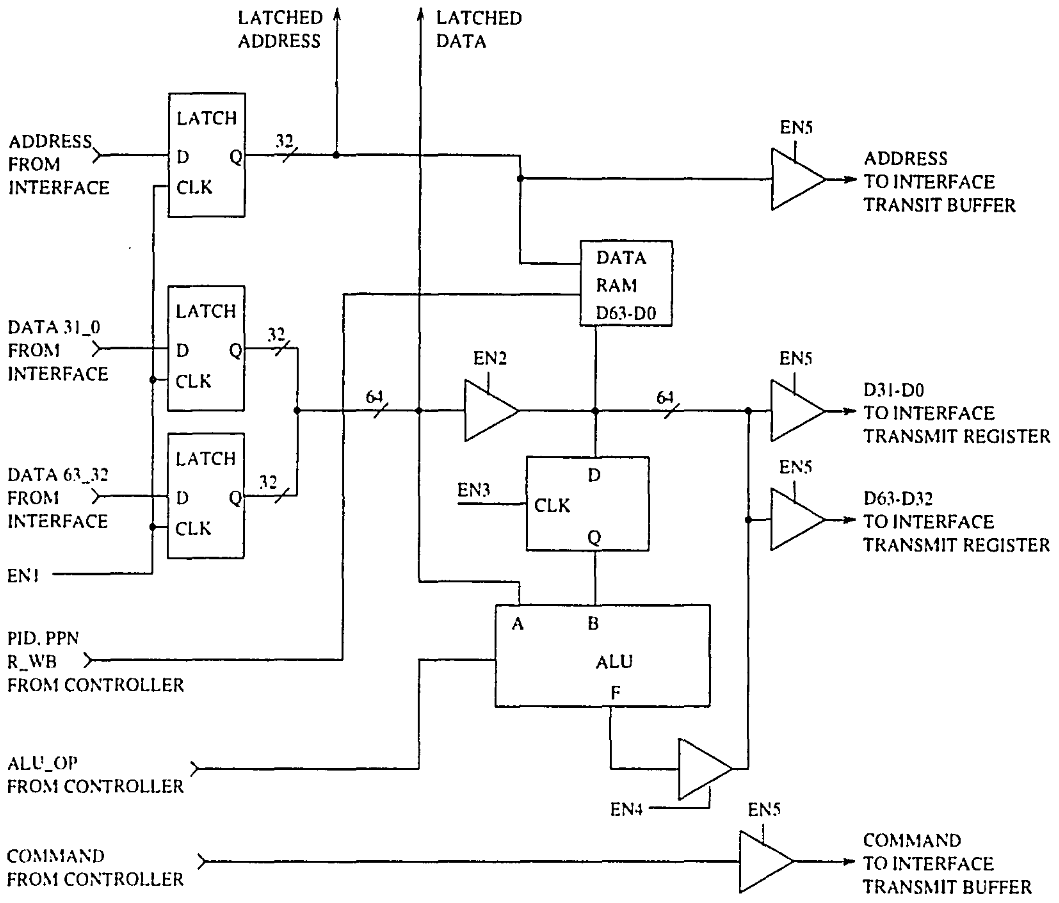

LATCHED ADDRESS
LATCHED DATA
ADDRESS FROM INTERFACE
LATCH
D Q
CLK
32
EN5
ADDRESS TO INTERFACE TRANSIT BUFFER
DATA RAM D63-D0
DATA 31_0 FROM INTERFACE
DATA 63_32 FROM INTERFACE
EN1
EN2
64
D31-D0 TO INTERFACE TRANSMIT REGISTER
D63-D32 TO INTERFACE TRANSMIT REGISTER
EN3
PID, PPN R_WB FROM CONTROLLER
A B ALU F
ALU_OP FROM CONTROLLER
EN4
COMMAND FROM CONTROLLER
COMMAND TO INTERFACE TRANSMIT BUFFER

Figure 18. Block diagram of COP ALU block

In the current COP design, the ALU block requires two clock cycles to service a channel and do a simple RAM read or write, three clock cycles to service a channel and perform one element of a global integer or barrier operation, and 7 clock cycles to service a channel and perform a floating point operation. Probably the best way to explain how the ALU block works is to briefly describe the control and data flow during a few representative operations.

To start, a channel request to read the value of a shared variable in the Data RAM takes two clock cycles and proceeds as follows. During the first clock cycle, the four receive buffer registers on the selected interface are enabled and the contents transferred to the appropriate COP latches. The address and data registers are transferred to the corresponding latches shown in Figure 18 and the instruction word is transferred to a latch in the controller where it is immediately decoded. As mentioned in an earlier section, as soon as the COP Controller transfers a command, the interface controller sends an ACK word back to the compute node to indicate that the receive buffers on the COP end of the link are available.

Note that the address and data are always latched, rather than waiting to see if they are needed before latching them. Also, the latched address is always immediately applied to all three RAM blocks, and the latched data is always moved as close to the RAMs as possible. This approach costs nothing extra and helps get the address and data to RAMs as soon as possible. The PPN and PID parts of a memory address from an instruction word simply pass through the controller to the appropriate address input on the RAMs.

During the second clock cycle RAM is enabled and, if the XMIT_READY signal from the Mask Generator is asserted, the En5 signal will be asserted to transfer the data word to the interface for the requesting channel. Once the word is transferred to the interface, the interface controller automatically transmits the data word to the requesting node. Note that a command word and optionally an address word are returned along with the data word. The command word contains the PPN and PID required to identify the process to which the data word belongs. The returned address could represent a handle or a shared object identifier.

For a simple write to the Data RAM, the data latched during the first clock cycle is written to the RAM during the second clock cycle. This operation does not require a separate command be returned to the sending node because an ACK word was already returned when the command was read from the interface buffer registers.

A one-to-all broadcast operation seems a good way to further illustrate how the Mask Generator block and the ALU Block work together. As shown in Figure 19, during the first clock cycle the command words are transferred from the interface and the command decoded. During the second clock cycle, the addressed mask is read from the MASK RAM and En2 is asserted to bring the data word one step closer to the transmit buffers. If the XMIT_READY signal is asserted, then in the third clock cycle the data word is transferred to the transmit buffers of all the interfaces that are enabled by the mask read from the MASK RAM. Again, a command word is transmitted along with the data word in order to identify the process(es) for which the broadcast data word is destined.

Clock Cycle	1	2	3
	Enable Channel. Latch address, data, and command. Decode instruction.	Read Mask RAM	Transfer latched data, command, and address to transmit buffers. Increment channel count if not lock.

Figure 19. Activities during each clock cycle for a one-to-all broadcast

As an example of operations involving the ALU Block and the Terminal Count Detector Block, Figure 20 shows the activities that occur when a compute node sends a barrier entry command. During the first clock cycle, the command is transferred, latched, and decoded as before. In the second clock cycle, the addressed TC vector is read from the TC RAM and the appropriate bit reset. Also, during the second clock cycle, the mask which identifies the compute nodes participating in that specific barrier is read from the Mask RAM. If resetting the bit in the TC vector left all zeros, then during the third clock cycle a barrier exit word will be transferred to all the interfaces enabled by the mask. If the result was not all zeros, the modified TC bit vector will simply be written back to the

TC RAM during the third clock cycle.

Clock Cycle	1	2	3
	Enable Channel. Latch address, data, and command. Decode instruction.	Read TC RAM & reset bit for channel. Read Mask RAM in case needed.	If Done, broadcast barrier exit command to enabled channels.

Figure 20. Activities during each clock cycle for a barrier entry/exit operation

A global reduction (sum) and broadcast operation likewise involves all three COP blocks. Figure 21 summarizes the activities that take place during each clock cycle for this operation. Latching and decoding in the first clock cycle are the same as for other operations just described. At this time, the data word from the compute node will also be passed on to one set of inputs on the ALU.

During the second clock cycle, the bit vector will be read from the TC RAM, the appropriate bit reset, and the DONE signal asserted if this is the last contribution to the sum. Likewise during the second clock cycle, a mask will be read from the MASK RAM to be ready for a broadcast if this is the last contribution to the global result. Also, during the second clock cycle, the Data RAM supplies the previous result as the second operand to the ALU. The sum is produced at the very end of the second clock cycle.

At the start of the third clock cycle, the operand from the Data RAM is latched to release the RAM so that the result of the addition can be written back to RAM. If the DONE signal is asserted, indicating that all the involved nodes have participated, then the controller will assert the En5 signal to transfer the result to all the interfaces enabled by the mask from the Mask Generator.

For a global floating point reduction, the activities in the first and last clock cycles are the same as those for the integer reduction described in the previous section.

However, after the second operand is transferred to the FPU, the controller spins for five clock cycles until the result is ready, then transfers the result to the interface transmit buffers if appropriate. Note, I did not model the FPU, but simply assumed 5 clock cycles for this operation based on a discussion in Hennessy and Patterson [Henne90a].

Clock Cycle	1	2	3
	Enable Channel. Latch address, data, and command. Decode instruction.	Read Data RAM. Data though latch to ALU B input. Read TC RAM & reset bit. Read Mask RAM in case needed. Calculate sum.	Latch ALU B input. Write sum back to Data RAM. Write bit vector back to TC RAM. Broadcast sum if Done. Increment channel count if not lock.

Figure 21. Activities during each clock cycle for a global sum operation

The COP Controller Block

Figure 22 shows a block diagram for the COP Controller section. The 6-bit counter in this section provides the CHAN_CNT signal to the Mask Generator Block. As discussed earlier, CHAN-CNT is decoded to produce the signal which enables a particular network interface unit for servicing. If all of the interfaces are requesting service, then the counter cycles through a normal binary count sequence and the interfaces will be serviced on a round-robin basis. However, if one or more interfaces does not require service, the controller will jump the counter in sequence to the next interface that does need servicing. This jump requires one clock cycle, but in the case where only a few interfaces need servicing, it is cheaper than cycling through all the channels.

If the COP receives a lock command, this counter is simply stopped so the selected interface is serviced repeatedly until an unlock command is received. Except in the case of this Lock operation, the round-robin scheduling is fair because it services interfaces in orderly sequence and assures that each interface is only serviced once for each cycle through them all.

The counter can also be jumpered so that it cycles through 2, 4, 8, 16, 32, or the full 64 counts. This "short cycle" feature further improves performance in systems which do not utilize all of the interfaces on a COP.

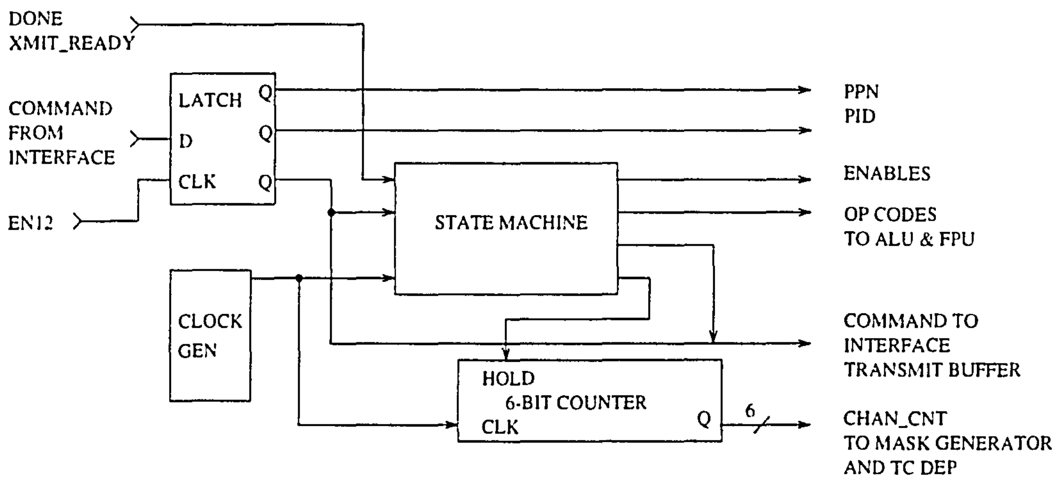


Figure 22. Block diagram of the COP controller block

Another major part of this block is a traditional hard-wired state machine controller. Table I shows the list of commands that the initial version of the COP controller is programmed to implement. One subset of commands left out of this first version of the COP was parallel prefix operations [Almas94a], where the partial result at each step in a global operation is returned to the contributor. The COP has many unused op codes available and the basic hardware configuration makes it very easy to add these operations. In the case of the global sum operation shown in Figure 21, for example, the controller can be programmed to send the partial sum to the contributor at the same time as it is written back to the Data RAM. The next design of the COP controller will include these and any other commands deemed useful.

TABLE I

COMMANDS AND OPCODES

Command	Opcode	Description	Privilege
RSRAM32	1011100	Read 32-bit data from Data RAM	U/S
WSRAM32	1011101	Write 32-bit data to Data RAM	U/S
RMW32	1011110	Read-Modify-Write 32-bit data	U/S
SUMI32	1010000	32-bit integer global Sum	U/S
MAXI32	1010001	32-bit integer global Maximum	U/S
MINI32	1010010	32-bit integer global Minimum	U/S
OR32	1011000	32-bit global logical OR	U/S
AND32	1011001	32-bit global logical AND	U/S
EXOR32	1011010	32-bit global logical EXOR	U/S
RSRAM64	1111100	Read 64-bit data from Data RAM	U/S
WSRAM64	1111101	Write 64-bit data to Data RAM	U/S
RMW64	1111110	Read-Modify-Write 64-bit data	U/S
SUMI64	1110000	64-bit integer global sum	U/S
SUMF	1110100	64-bit floating point global sum	U/S
MAXI64	1110001	64-bit integer global Maximum	U/S
MAXF	1110101	64-bit floating point global Maximum	U/S
MINI64	1110010	64-bit integer global Minimum	U/S
MINF	1110110	64-bit floating point global Miminim	U/S
OR64	1111000	64-bit global logical OR	U/S
AND64	1111001	64-bit global logical AND	U/S
EXOR64	1111010	64-bit global logical EXOR	U/S
BENTRY	0000001	Barrier entry and broadcast	U/S
LOCK	0000010	Lock a channel	U/S
UNLOCK	0000011	Unlock a locked channel	U/S
XMIT	0000100	Retransmit a message	U/S
ACK	0001000	Acknowledgement for a message	U/S
BCAST32	0011100	Broadcast 32-bit word	U/S
SETBM	0111110	Set broadcast mask	S
SETTC	0111111	Set TC Bit vector	S
BCAST64	0111100	Broadcast 64-bit word	U/S

Note that with the COP design there is no conflict if, for example, one node sends a global sum command and another sends a Data RAM read command. Assuming the controller services the interface with the global sum command first, the COP will simply add that interface's contribution to the intermediate sum in the Data RAM. If the global sum is now complete, the COP will broadcast it to all the nodes waiting for the sum. As mentioned earlier, an instruction is is broadcast along with the sum to identify the sum for the receiving nodes. If the global sum was not complete, the COP will just go on to the interface requesting a Data RAM value and service that request.

In addition to the direct COP commands, there are several other control features which need further explanation. First, the transmit and receive interfaces to higher level COP are enabled directly by the controller, rather than through the Mask Generator block. If the X bit is set in a command, the Controller will assert the enable signals required to transfer the words of the command directly to the transmit buffers of the interface which connects to a higher level COP. Likewise, the Controller directly monitors the higher level COP interface to detect commands coming in from the higher level COP.

A final point about the Controller, or actually about the overall COP design, relates to how I divided the activities among clock cycles for different operations. As shown in the preceding sections the clock by clock activities are very similar for most COP operations. While this makes the controller simpler and easier to implement, it sacrifices some efficiency for specific operations. For example, as I discuss in Chapter VI, several research projects and the Cray T3D have added a secondary network and essentially a big AND gate just to implement barriers. This approach can obviously be very fast, but it is also very limited. My feeling is that the versatility, protection features, and ease of programming provided by the COP approach justify the small additional time cost.

Software Interface For The COP System

In accordance with the goal that existing programs require minimum modification to utilize a COP, the COP does not change the basic parallel programming operations. It simply provides an alternative hardware mechanism to implement them. This is analogous to the way a numeric coprocessor provides an alternative hardware based mechanism for implementing floating point computations on an 80x86 based PC system. The overall software approaches for the COP system can be similar to those used for a numeric coprocessor.

A very important point about the COP system is that User/Supervisor and process level protection are both enforced in hardware. This means that application programs can safely be given direct, low-overhead access to the User COP port. For programming at the lowest level then, the COP system gives the programmer the additional set of commands shown in Table I. An assembler or compiler can translate each COP command into the short sequence of machine instructions actually needed to implement it. Perhaps, a reasonable scenario would be to write a program so that it contains both standard mechanisms and COP based mechanisms for relevant operations. Upon startup, the program could then determine if the system contains a COP. If so, the program could use the more efficient COP based code sections for the relevant operations. This approach is commonly used in math-intensive application programs intended to run on 80X86 based PCs.

At a somewhat higher level of programming, COP enhanced code could be included as the same type of alternative in programming libraries such as MPI [Tennea]. The Collective Communication section of the MPI library, for example, contains functions for one-to-all broadcast, barriers, global operations, all-to-all broadcast, parallel-prefix, etc. At the actual implementation level, the library could contain both standard message-passing code and COP-based code for these operations.

At a still higher level of programming, Li [Li91a] has done considerable work with compiling shared memory programs to run on distributed memory machines. His approach involves partitioning program data for efficient communication, syntactically analyzing a program to recognize the required communication patterns, and at runtime choosing the least cost routine to implement the required communication on the target machine. It seems that a sophisticated compiler could likewise analyze programs to determine functions that can be implemented efficiently by the COP system and insert the hooks necessary to call the appropriate routines at runtime if the system was found to contain a COP.

A final point here is that the COP system does not prevent use of the standard features on a given system. It simply provides a more efficient execution mechanism for global operations.

CHAPTER V

COP PERFORMANCE ANALYSIS AND COMPARISONS

COP Performance

Introduction

In the last chapter, Figures 19, 20, and 21 showed the activities that take place in the major COP blocks during each clock cycle. In this section I extend the analysis of the COP system, derive equations for the total times required for various global operations, and use simulation results to project some actual execution times for these operations.

In order to determine the feasibility and hardware performance capability of the COP system, I modeled the major portions of a single-level COP system in VHDL and simulated the resulting models using the Mentor Graphics 1076 VHDL System and QuickSimII simulator. Although I only modeled and simulated the functional blocks for a single level COP system, I feel that that the basic symmetry of the COP system allows the results to be extended to a two level system with a high degree of confidence.

For timing parameters in the VHDL models, I used values from currently available commercial products so that I could verify the feasibility of building the system with existing technology. In the RAM model, for example, I used the parameters of the NEC μPD46258 which has a maximum address access time of 6 ns and an output enable access time of 4 ns. For latches, buffers, multiplexers, counters, etc. I used the propagation delay values for these devices implemented in the Motorola ECLinPS [Motor94a] family which is compatible with the Spanceiver. Maximum propagation delay for a single gate device in this family is about 500 ps. Other maximum delay values for devices in this family are 700 ps for a D latch, 850 ps for a 2-input multiplexer, and 1100 ps for a 6-bit binary synchronous counter. Using these timing parameters, the COP system design described in the last chapter simulated correctly with a 10 ns clock period.

To simplify later performance projections, I also assume a 10 ns compute node or workstation clock period. This assumption is justified by several commercially available machines. The DEC AlphaStation 200 4/233 workstation [Lee94a], for example, has a 4.3 ns clock period, and the Cray T3D Multiprocessor discussed in Chapter II has a 6.7 ns clock period. Because it is an important part of many other operations, I will analyze the COP One-To-All broadcast operation first.

One-To-All Broadcasting with a COP

To perform a one-to-all broadcast in a multicomputer with a COP system, a compute node writes the appropriate instruction word and data word(s) to the COP network interface. The interface then sends these to the local COP. The number of clocks required for a compute node to output a word and for the interface to transmit it to a COP will be represented as t_{CN-COP}. Assuming the compute node can write a 64-bit word to its COP port in two clock cycles, $t_{CN-COP} = 11$ clocks for a command word, address word, and a 64-bit data word.

When the network interface on the COP correctly receives all the words of a command, it will assert its Data Ready output. The COP controller checks the Data Ready signals on a round-robin basis. Assuming the worst case condition that all channels are requesting some type of service, the total number of clocks to cycle through all the other channels and get to the channel requesting a broadcast is $C * t_{OP}$. C in this expression represents the number of channels in a COP and t_{OP} represents the number of clocks that a COP requires to service each input channel. As shown in Figures 19, 20, and 21, t_{OP} is 3 for integer and Data RAM operations, but increases to 7 for floating point operations.

To implement the broadcast operation, the COP uses a mask from the Mask RAM to enable the transmit buffers for the desired destination nodes and writes the data word to all these buffers at the same time. The interfaces then transmit the data word to all or a desired subset of the compute nodes simultaneously. Again assuming a compute node can read a 64-bit value from its COP port in two clock cycles, 17 clock cycles are required for the network interfaces to transfer a broadcast data value and the receiving compute node to read the word. This time will be referred to as t_{COP-CN}.

Figure 23 shows the timing for these operations in diagram form. Intuitively or from this diagram the maximum number of clocks required for a single one-to-all broadcast of an 8-byte data value on a single level COP system is $t_{CN-COP} + C * t_{OP} + t_{COP-CN}$. The total time required then is: $t_{1-All} = (t_{CN-COP} + C * t_{OP} + t_{COP-CN})P_{CLK}$, where P_{CLK} is the period of the COP system clock. Using the previously stated values of $t_{CN-COP} = 11$, $t_{OP} = 3$, $t_{COP-COP} = 17$, and a 10 ns clock period, gives a time of 2.2 microseconds to broadcast an 8-byte value to 64 compute nodes. For comparison, later I show that a 2-D mesh, message-passing multicomputer such as the Intel Paragon requires about 350 microseconds for the same type of broadcast.

One small note here is that from the Spanceiver data sheet I was not able to determine if its design allows for metastability settling time. If not, this time can be provided by adding an extra flip-flop to each interface. With this addition, t_{CN-COP} and t_{COP-CN} will each be increased by one. This will increase the total time for a one-to-all broadcast from 2.20 microseconds to 2.22 microseconds. Since this change is less than 1% and less than the precision of the software timing assumptions, I will ignore it in the analyses that follow.

The COP timing for a multicast to, for example, some number of nearest neighbors connected to the same COP is the same as that for a one-to-all broadcast, because the broadcast or multicast word is transferred to all of the enabled transmit buffers at the same time.

Note that some small additional overhead is incurred in initializing the broadcast/multicast mask. However, in iterative algorithms the same mask will be used multiple times and this startup overhead will thus be amortized over many broadcast operations.

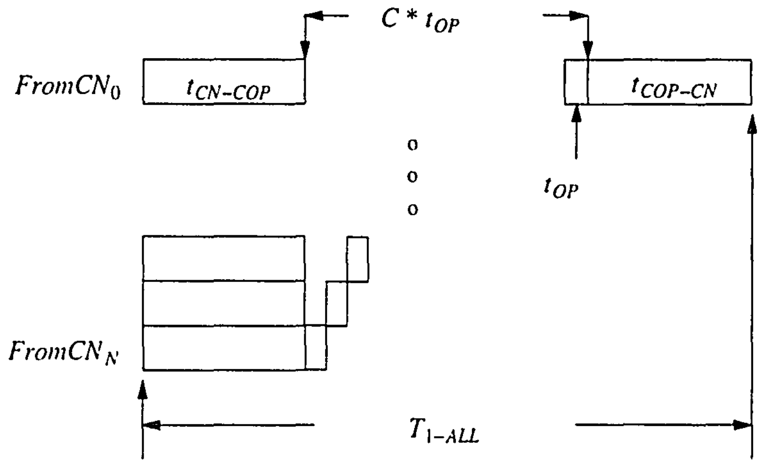


Figure 23. Timing for One-To-All broadcast with single level COP system

For one-to-all broadcast of larger vectors, the COP channel lock feature can be invoked. Figure 24 shows a diagram for this kind of operation. In the worst case, the COP controller may cycle through servicing all the other channels before it gets to the channel desiring to broadcast, but once the channel is locked, successive broadcasts can proceed in "pipeline" fashion as shown. The key here is that as soon as the COP reads a command, a one word ACK is returned to the sending compute node to indicate that it can send the next command. If the broadcast mask is set up so that the COP does not send the broadcast word(s) back to the sending node, then a new command can be coming into the COP from the sending node at the same time that the broadcast word is being transmitted to the receiving nodes. Once the pipeline is running, only $t_{OP} + t_{COP-CN}$ clocks are required for each 8-byte block. The total time to broadcast a vector with M_B bytes then is

$$t_{M_B} = [t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN} + (\frac{M_B}{8} - 1)(t_{OP} + t_{COP\text{-}CN})]P_{CLK}$$

Substituting the previously stated values in this expression reduces it to about T_{M_B} = 2.2 microseconds + M_B x .025 microseconds/byte. Assuming the vector is long enough to swamp out the 2.2 microseconds, this gives a broadcast(not just point-to-point) bandwidth of about 40 Mbytes/sec. Broadcasting a 64K-byte vector in this way requires about 1600 microseconds. Note that some small additional startup overhead is incurred in issuing the lock and unlock commands.

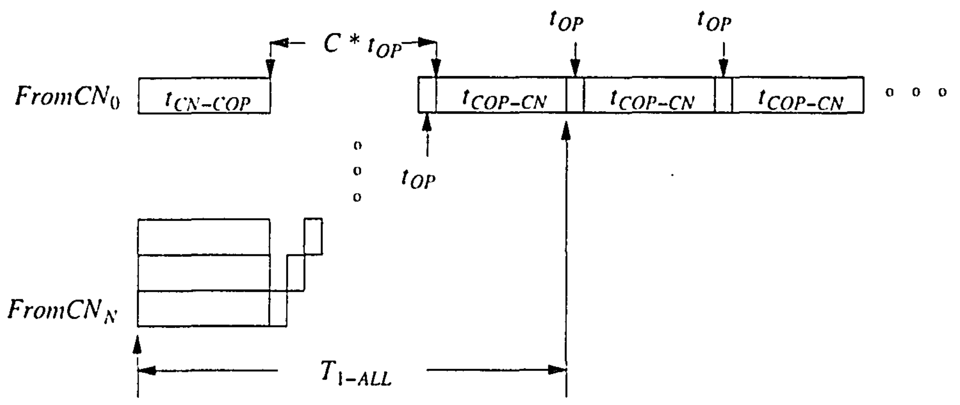


Figure 24. Timing for One-To-All broadcast of long vectors with single COP

Figure 25 shows the timing for a one-to-all broadcast with a two-level COP system. For this operation the sending compute node transmits the command to its local COP and the local COP passes the command on to the Level 2 COP. The Level 2 COP broadcasts the command to all the Level 1 COPs that need to receive it. Each of the level 1 COPs broadcasts the value to the compute nodes enabled by the appropriate mask from its MASK RAM. For analysis of a two level system it helps to introduce an additional variable, L, which represents the number of COP levels in a system. If C represents the number of channels in each COP and N is the number of compute nodes, then L is equal to

$\log_C(N)$. The time required for a two-level COP-assisted one-to-all broadcast is:

$$t_{1\text{-}ALL2} = [t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN}]P_{CLK}$$

This expression assumes the worst case situation where each of the COP controllers has to cycle through servicing all the other channels before it services the channel which has the data to be broadcast. Assuming $t_{COP\text{-}COP} = t_{CN\text{-}COP}$ and using the previously stated values from our simulation, this expression indicates that a two-level COP system will require a worst case time of about 6.3 microseconds to broadcast an 8-byte value to as many as 4096 compute nodes.

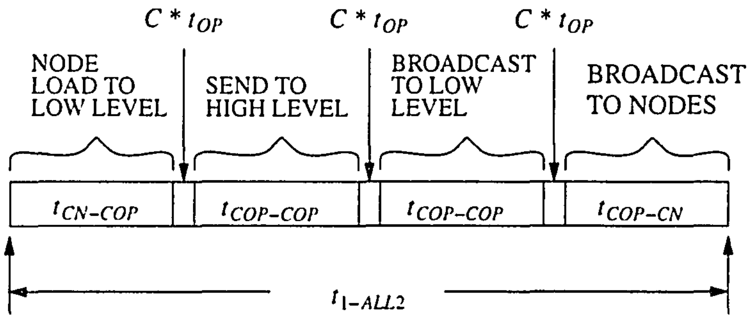


Figure 25. Timing for One-To-All broadcast with two-level COP

Note that the expression in the preceding paragraph assumes the maximum size two level system where all the level 2 COP inputs have level 1 COPs that must be serviced. For systems with less than 64 level 1 COPs, the short cycle feature of the level 2 COP can be used to minimize the number of channels checked, and thus reduce the effective value of C for the level 2 COP. The expression for the required time in this case is:

$$t_{1\text{-}ALL2} = [t_{CN\text{-}COP} + ((2L-2)C_1 + C_2)t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN}]P_{CLK}$$

For a 256 node system which uses only 4 channels on the level 2 COP, this gives a worst case time of 4.5 microseconds to broadcast an 8-byte value to all 256 nodes.

For one-to-all broadcast of longer vectors to a large number of compute nodes, the COP channel lock command can be invoked on all COPs. Once the channel locks are in place, successive words can be pipelined through the two COP levels and broadcast to all the nodes. The pipeline is longer in this case, but once the pipeline is flowing, the data rate is the same as that for the single level COP system. Assuming 6.3 microseconds to get the channel locks in place and another 6.3 microseconds to broadcast the first word, the time for an M byte vector is about T_{M_B} = 6.3 microseconds + 6.3 microseconds + M_B x .025 microseconds/byte. This expression predicts a time of about 1650 microseconds to broadcast a 64 Kbyte vector to as many as 4096 compute nodes.

Global Sum and Other Global Operations

A comparison of Figures 19 and 21 shows that t_{OP} is the same for both operations. Assuming that all the participating compute nodes send a contribution to their COP at the same time, the overall timing diagram for a global integer or bitwise logical operation performed with a single level COP is the same as that shown in Figure 23 for a COP implemented one-to-all broadcast. Therefore, the expressions for the total times are the same. Likewise, the timing for a global sum or bitwise logical operation with a two-level COP system is the same as that shown for a two level one-to-all broadcast in Figure 25. A COP assisted 64-bit integer reduction or bit-wise logical operation then requires a maximum of 2.2 microseconds on a single level cop system with up to 64 compute nodes and a maximum of about 6.3 microseconds on a two level COP system with up 4096 nodes. For a double-precision floating-point reduction t_{OP} is 7 clocks, so the total time is 4.7 microseconds for 64 compute nodes and about 14 microseconds for 4096 compute nodes.

The times for producing global results for longer vectors on a COP system are calculated by cascading the times for single word operations. For example, to produce the sum vector for 1024 element integer vectors over 64 compute nodes requires 2.2

microseconds x 1024 = about 2200 microseconds. The time to produce a double-precision floating point sum vector for 1024 element vectors over 4096 compute nodes is 14 microseconds x 1024 = about 14,000 microseconds.

Barrier Implementation

When a compute node in a COP enhanced system reaches a barrier point, it sends just an instruction word to its COP. As described earlier, the COP controller uses the PPN and PID in the command to access the appropriate TC bit vector in the TC RAM and the appropriate mask in the MASK RAM, then resets the bit for that compute node in the TC vector. If all the TC vector bits are reset, the barrier exit command is broadcast simultaneously to all the participating compute nodes.

The timing expressions for COP implemented barriers are the same as those for one-to-all broadcasts, but since only command words are sent in each direction, the actual times are slightly less. Ignoring this small difference and using the COP values from previous examples, gives a worst case time of 2.2 microseconds for a barrier with up to 64 nodes and a worst case time of 6.3 microseconds for a two level COP barrier with up to 4096 nodes.

Note that the mask mechanism in the COP system allows any subset of the compute nodes to participate in a barrier. Also, it permits multiple barriers to be in force within a partition at the same time and thus provides for finer grained synchronization than that provided by a single global barrier. Yeung [Yeung92a] has shown that finer grained synchronization improves performance on conjugate gradient problems.

All-To-All Broadcast

Figure 26 shows the basic timing diagram for an all-to-all broadcast of a 8-byte word using a single level COP. During the first phase, all the compute nodes simultaneously transmit a value to their COP. During the second phase, the COP sequentially reads each of the received values and broadcasts each to the mask selected set of compute nodes. This phase is just a series of one-to-all broadcasts. The number of clocks for each of these broadcasts is $t_{OP} + t_{COP-CN}$ so the number of clocks to receive and broadcast all the values is about:

$$t_{A2A} = t_{CN-COP} + C(t_{OP} + t_{COP-CN})$$

Using the COP values from previous examples, the time required for a COP assisted all-to-all broadcast of a 8-byte word with 64 processors is about 13 microseconds.

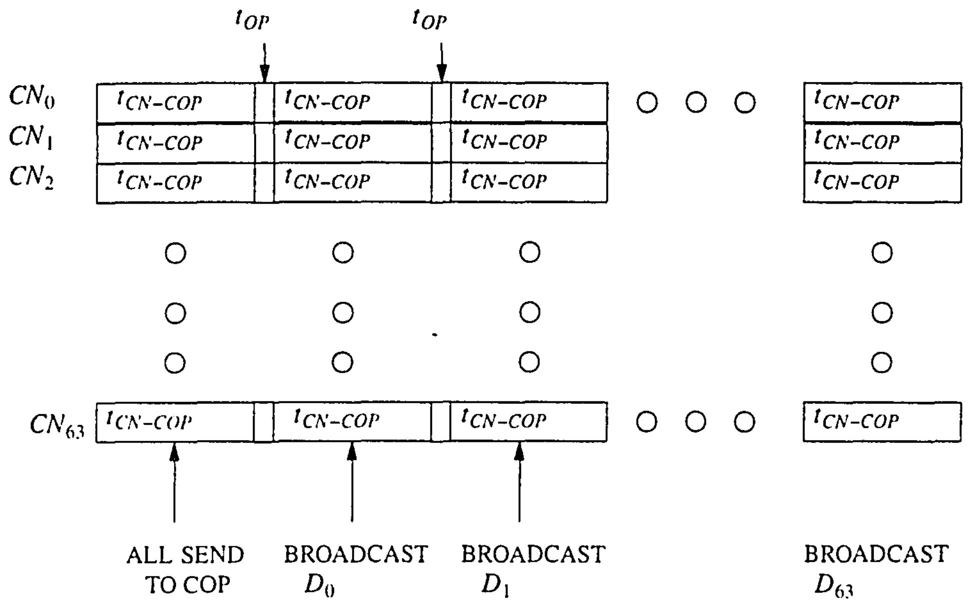


Figure 26. Timing for All-To-All broadcast with single-level COP

An all-to-all broadcast with a two-level COP system requires several phases. The details of these phases depend on the particular type of all-to-all broadcast. It can be shown that the time for the worst case, where each node broadcasts an 8-byte value to all the other nodes is:

$$t_{A2A2} = [t_{CN-COP} + C((t_{OP} + t_{COP-COP}) + C(t_{OP} + t_{COP-COP}) + C(t_{OP} + t_{COP-CN})]P_{CLK}$$

Assuming for simplification that $t_{CN-COP} = t_{COP-COP} = t_{COP-CN}$, then t_{A2A2} can be approximated as:

$$t_{A2A2} = (2C^2 + C + 1)(t_{OP} + t_{COP-CN})P_{CLK}$$

With 64 channel COPs and the same timing parameters used in the preceding examples, this expression predicts the time for an all-to-all broadcast of a 8 byte word with 4096 compute nodes to be about 1,650 microseconds or 1.65 ms.

Shared Variable Access

In a COP system, global shared variables can be stored in the Data RAM. To service a shared variable read, write, or read-modify-write request, the COP uses a decoded combination of the address, PPN, and PID received from the compute node to access the variable in the Data RAM. In the case of a simple read, the addressed value is simply transmitted back to the requesting node. For a simple write, the data value received as part of the command is written to the addressed Data RAM location. In the case of a Read-Modify-Write Access, the value read from the Data RAM is transferred to the transmit buffer for the requesting node and at the same time applied to the ALU. The ALU performs the specified operation on the data value and the result is written back to the Data RAM for the next access.

Assuming simultaneous requests and a single COP level, the maximum time for all of the requesting nodes to receive individual values is:

$$t_R = [t_{CN-COP} + C * t_{OP} + t_{COP-CN}]P_{CLK}$$

Substituting the previously stated COP values in this expression indicates that a single level COP can supply different or the same 8-byte variable(s) to 64 compute nodes in about 2.2 microseconds. The node serviced first gets its value in about 300 ns and the last gets its value after 2.2 microseconds, so the average time to service a request is about

1.25 microseconds. Note that the COP shared memory provides sequential consistency in that the value read by any compute node will be the last value written.

The expression for the worst case time to supply a shared variable to each of 4096 compute nodes can be shown to be approximately:

$$t_{R2} = (2C^2 + C + 1)(t_{OP} + t_{COP\text{-}CN})P_{CLK}$$

Using the previous COP values, this expression predicts a time of about 1650 microseconds to supply the same value or successively modified values of a shared variable to up to 4096 compute nodes with a two-level COP system.

Performance Summary

Table II summarizes the performance data calculated for various global operations using the expressions shown and the timing values derived from VHDL modeling and simulation. Note that these expressions assume the worst case where all of the COP inputs are connected and all require servicing. Timing values for smaller systems can be derived as shown previously for a one-to-all broadcast on a two-level COP system with 256 nodes. Also note that I did not include wire or fiber-optic cable delay of 1.6 ns/foot because the time this contributes would be about the same for all systems.

As I show in the next major section of the chapter, the performance numbers shown in TABLE II are quite impressive when compared with those for the same operations on "Big Iron" and other multicomputers. First, however, I will briefly discuss some topology tradeoffs for a COP system while the preceeding analyses are still fresh.

COP Topology Tradeoffs

For the discussions thus far I have assumed a 64-ary tree topology for the COP system, but the timing values from the VHDL simulation and the analyses in the preceding section provide a mechanism for evaluating the tradeoffs in using different width trees for

TABLE II

COP GLOBAL OPERATION TIMES AND EXPRESSIONS

Operation	Execution Time (μS)	Expressions for execution times one level and two level COP systems
Read/Write	2.2	[L=1] $(t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN})P_{CLK}$ (64 nodes)
	1400	[L=2] $(2C^2 + C + 1)(t_{OP} + t_{COP\text{-}CN})P_{CLK}$ (4096 nodes)
One-to-All	2.2	[L=1] $(t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN})P_{CLK}$
	6.3	[L=2] $(t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN})P_{CLK}$
Global OP Integer	2.2	[L=1] $(t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN})P_{CLK}$
	6.3	[L=2] $(t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN})P_{CLK}$
Global OP Double Float	4.7	[L=1] $(t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN})P_{CLK}$
	14	[L=2] $(t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN})P_{CLK}$
Barrier	2.2	[L=1] $(t_{CN\text{-}COP} + C * t_{OP} + t_{COP\text{-}CN})P_{CLK}$
	6.3	[L=2] $(t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN})P_{CLK}$
All-to-All	13	[L=1] $(t_{C_n\text{-}COP} + C(t_{OP} + t_{COP\text{-}CN}))P_{CLK}$
	1,650	[L=2] $(2C^2 + C + 1)(t_{OP} + t_{COP\text{-}CN})P_{CLK}$

the COP system topology.

As stated previously, the number levels in a given COP system is $L = \log_C N$, where C is the number of channels per COP and N is the total number of compute nodes in the system. Also stated previously was the fact that the worst-case time required for a one-to-all broadcast on a COP system is:

$$t_{1\text{-}ALL} = [t_{CN\text{-}COP} + (2L-1)C * t_{OP} + (2L-2)t_{COP\text{-}COP} + t_{COP\text{-}CN}]P_{CLK}.$$

The first section of TABLE III shows the times required for a one-to-all broadcast on 64 node COP systems with different width COP network trees.

Looking at just the times in this section of TABLE III leads to the conclusion that an oct tree with 8 nodes per COP is the best choice by about a factor of two. However, looking at hardware complexity required for this topology shows the tradeoffs.

The number of COPs required for a particular topology can be expressed as:

$$N_{COPs} = \sum_{i=0}^{L-1} C^i$$

and the number of links between compute nodes and COPs and between COPs can be expressed as:

$$N_{Links} = \sum_{i=1}^{L} C^i$$

The Number of COPs column in TABLE III shows that an oct-tree based COP system with 64 compute nodes requires nine times as many COPS as the same system with a 64-ary tree topology. TABLE III furthermore shows that the oct-tree based topology requires 12.5% more links than a 64-ary tree topology.

The second section of TABLE III shows that for a 4096 node system the oct-tree topology is also faster than the 64-ary tree topology by about a factor of two. Again however, the oct-tree based topology requires nine times as many COPs and 12.5% more links. It seems doubtful that the factor of two increase in performance predicted for an oct-tree based topology justifies the cost of this large number of additional COPS and links. Also, increasing the number of components by a factor of nine significantly increases the chances of a component failure.

Another deciding factor not shown in TABLE III is the increased software complexity required to initialize all the additional COPS and to direct commands to the appropriate COPs. My programming experience indicates that the extra software overhead

TABLE III

COP TREE-WIDTH TRADEOFFS

Nodes	Channels per COP	Number of COP Levels	Total COPS	Links	Clocks 1-All	Time μs
64	64	1	1	64	220	2.2
64	8	2	9	72	124	1.2
64	4	3	21	84	136	1.4
64	2	6	63	126	214	2.1
4096	64	2	65	4160	628	6.3
4096	16	3	273	4368	316	3.2
4096	8	4	585	4680	268	2.7
4096	4	6	1365	5460	280	2.8
4096	2	12	4097	8191	430	4.3
32	32	1	1	32	124	1.2
1024	32	2	33	1056	330	3.3
32768	32	3	1057	33824	556	5.6

required by an oct-tree topology COP system could easily eliminate much of the gain predicted by TABLE III.

In this initial research I opted for the 64-ary tree topology to reduce hardware costs, make programming easier, service a sizable number of compute nodes with a single COP board, and accommodate up to 4096 nodes with only a two level system. However, the COP system topology tradeoffs represent a fairly large design space. To give a few more points in this space, the third section of TABLE III shows the values for different sized systems using a 32-ary tree topology. This topology might prove appropriate for a workstation cluster system, where it seems unlikely that a 4096 node capability would be needed. The point here is that further research will be needed, especially in the software aspects, to center a final design within the available design space. However, in the next section I show that the initial 64-ary tree COP system outperforms other current systems.

Comparison of Other Systems with the COP System

"Big Iron" Multicomputer Global Operation Performance Comparison

In this section I derive expressions for the time required to perform these operations on example "Big Iron" multicomputers, cluster multicomputers, and distributed shared memory multiprocessors, and then compare the performance of the COP system where possible with the published performance of current, representative machines in each category

As discussed earlier, assuming no contention for network channels, the time required to send and receive a single message on a message-passing-only multicomputer is $t_m = t_L + M_B * R_T$. The latency time, t_L, includes the time required to packetize the data to be sent, the time to add a header which specifies the destination for the message, the time to establish a path through the network, and the time to depacketize and process the message at the receiving node. The total source to destination hardware transmission time is $M_B * R_T$, where M_B is the number of bytes in the message and R_T is just the reciprocal of the network message bandwidth. Performing any operation on a message-passing-only system will require some number of these message times plus any required computation time.

In a 2-D mesh-connected, message-passing machine, the number of steps or phases required to broadcast a message from one compute node to all the other compute nodes is the diameter of the network, D. For a square mesh, $D = 2\sqrt{N} - 2$, where N is the number of compute nodes. For a rectangular mesh, $D = h + w - 2$, where h is the number of compute nodes along one edge and w is the number of nodes along an orthogonal edge. Since each phase requires a time of $t_L + M_B * R_T$, the total one-to-all broadcast time is: $t_{1\text{-}ALL} = D(t_L + M_B * R_T)$.

Using the Intel Paragon numbers of t_L = 25 microseconds, and R_T = 1/200

Mbytes/sec cited earlier from Rosing [Rosin94a], this equation gives a time of 350 microseconds to broadcast a 64-bit value to the other 63 compute nodes in a 64 node mesh. For a 16x32 rectangular mesh, the expression predicts a time of 1200 microseconds or 1.2 ms to broadcast an 8-byte value. This number agrees closely with that recently reported by van de Geijn [Geijn94a] for a 16x32 Paragon using the new InterCom Collective Communications Library. For future comparison, a 4096 node square mesh machine would require about 3150 microseconds for this operation. van de Geijn also reported that the 16x32 Paragon required 12,000 microseconds to broadcast a 64 Kbyte vector to all 512 nodes. For a preliminary comparison, an earlier section mentioned that a two-level COP system requires only about 1600 microseconds to broadcast a 64 Kbyte vector to as many as 4096 compute nodes.

For global reductions with small vectors, Barnett [Barne93a] has shown that a fanin/fanout algorithm is the most efficient on 2-D mesh multicomputers. As discussed in Chapter III, this algorithm uses a minimum spanning binary fanin tree to produce the global sum at one node and then uses the same type tree to broadcast the result to all the other compute nodes. The number of steps in each tree is equal to D, the diameter of the network. Assuming that the time to do the actual addition is very small as compared to the message latencies, then the basic timing for a global sum operation on a message-passing mesh is: $t_{SUM} = 2D(t_L + m_B * R_T)$. This is just twice the time calculated earlier for a broadcast, so the predicted time for a global sum is 700 microseconds with 64 nodes, 2300 microseconds with a 512 node mesh and 6300 microseconds with a 4096 node mesh.

For synchronization barriers implemented with Fanin-Fanout trees on a 2-D square mesh, message passing machines the expressions and times are the same as those given in the preceding section for a global sum operation. Specifically, the times are 700 microseconds for a 64 node machine, 2400 microseconds for a 512 node machine, and

6300 microseconds for a 4096 node machine.

For all-to-all broadcast on a message passing machine, Johnsson [Johnsa] has shown that the lower bound on the required time is (N-1) x M x t_C, where M is the number of words and t_C is the communication time. Expressing this in terms of T_L, M_B, and R_T gives $T_{A2A} = (N-1)(t_L + M_B * R_T)$.

This expression predicts an all-to-all broadcast time of about 1600 microseconds with a mesh of 64 compute nodes and a time of about 102,000 microseconds for a mesh with 4096 compute nodes.

In order for multiple compute nodes to read successively modified values of a shared variable which is stored in the memory of one of the other compute nodes in a multicomputer, each node sends a request message to the node holding the variable. That node has to return a message containing the variable.

If we generously assume that the time to access memory on the receiving node is negligible as compared to the message passing overhead, then the total time required to send and receive the request message, and send and receive the reply message is just $2(t_L + M_B * R_T)$.

If the system has N processors, then as many as N-1 processors can be sending acquire messages to the node which has the shared variable and that node must send values back to all N-1 processors. If a compute node has a single port connection to its router and it can send one message and receive one message at a time, requests and responses can be overlapped, so the total time to service the N-1 requests is about $(N-1)(t_L + M_B * R_T)$. With the previously stated Paragon parameters, this gives a time of about 1600 microseconds to service 63 requests or 102,000 microseconds to service 4095 requests.

TABLE IV summarizes the expressions for the times required by various global operations on a 2-D square mesh multicomputer and shows the timing performance these expressions predict for a machine with the latency and bandwidth parameters of the Intel Paragon. For comparison, TABLE IV also includes a copy of the COP timing values from TABLE II.

TABLE IV

"BIG IRON" MULTICOMPUTER TIMING EXPRESSIONS AND VALUES COMPARED WITH COP VALUES

Processor Operation	Expression for Time Required	64 Processor Paragon type machine μS	COP + 64 Processor Paragon type machine μS	4096 Processor Paragon Type machine μS	COP + 4096 Processor Paragon type machine μS
R-M-W	$N(t_L + m_B R_T)$	1,600	2.2	102,000	1400
One-to-All	$(2\sqrt{N} - 2)(t_L + M_B R_T)$	350	2.2	3,150	6.3
Global OP:64 bits	$2(2\sqrt{N} - 2)(t_L + M_B R_T)$	700	2.2	6,300	6.3
Barrier	$2(2\sqrt{N} - 2)(t_L + M_B R_T)$	700	2.2	6,300	6.3
All-to-All:64 bits	$(N - 1)(t_L + M_B R_T)$	1,600	13	102,000	1,650

TABLE IV clearly shows that the times for these global operations on a COP system are consistently 2-3 orders of magnitude less than they are for a 2-D mesh multicomputer without a COP system. Although the numbers in TABLE IV are for 8-byte quantities, the COP system also maintains a performance advantage for longer vectors. As stated earlier, for example, van de Geijn [Geijn94a] reported that a 16x32 node Paragon using the new InterCom Collective Communications Library required 1300 microseconds to broadcast an 8-byte value to 512 nodes and about 12,000 microseconds to broadcast a 64 Kbyte vector to all 512 nodes. In an earlier section I showed that a two-level COP system requires only 6.3 microseconds to broadcast an 8-byte value to as many as 4096 nodes and only about 1600 microseconds to broadcast a 64 Kbyte vector to as many as 4096 nodes. In a later section of the chapter I make some projections about the reduction in overall program execution time provided by adding a COP system to this kind of machine.

Cluster Multicomputer Global Operation Performance Comparison

A section in Chapter II described several attempts currently under way to build scalable multicomputers with clusters of workstations using mesh router or crossbar switch networks. A major difficulty with these systems is the long message latency. As cited in Chapter III, Steenkiste [Steen94a] reports that the measured time for a single 16-byte message on Carnegie-Mellon's Nectar cluster system is 97 to 234 microseconds, depending on the protocol used. As also cited in Chapter III, Cohen [Cohen94a] reports that the message latency between user memory on one node and user memory on another node in the USC/ISI ATOMIC cluster system is about 1500 microseconds.

Due to the mesh or similar topology of these systems, global operations are usually implemented with some type of minimum spanning tree. The long message latency times cascade along the branches of the trees and make these operations very expensive. The long message latency time on these systems is caused by the high-level "protocol stack" used to send and receive messages. These time consuming protocols are required to maintain protection, implement error detection, and maintain software compatibility between machines.

Since the COP system enforces User/Supervisor and process level protection in hardware, user programs can safely be given direct access to the COP port, which is mapped into user memory or port address space. The COP system can then provide even greater speedup for global operations on this type system than on a "Big Iron" multicomputer such as that described in the preceding sections. TABLE IV shows that even if the performance of a cluster system's main interconnection network catches up with that of the "Big Iron" machines, the COP system still provides substantial performance gain for global operations.

Distributed Shared Memory System Global Operation Comparison

According to Koeninger [Koeni94a], the latest and greatest of the distributed shared memory machines is the CRAY T3D. This machine is connected as a 3-D torus with two processing elements at per node. Each processing element has up to 64 Mbytes of DRAM which is part of the physically distributed, logically shared global memory. Memory transfers are done with message packets containing "payloads" of one to four 64-bit words. The global memory is managed by special, dedicated hardware, so a processing element can read from or write to a memory location physically located at another processor without directly involving that processor.

When a processor initiates a memory operation, the memory management hardware determines if the addressed location is in the local memory. If not, a short message is sent to the remote memory containing that address. For a write operation, the hardware at the remote processor simply writes the data word to the specified address. For a read, the hardware at the remote processor generates a short message which returns the data word to the requesting processor.

I was not able to find published data specifically showing the performance of the T3D for global operations, so for comparison purposes I will use published values of message latency and bandwidth to estimate some values for these operations. Measurements reported by Numrich [Numri94a] show that the average latency (software + hardware) for a single 64-bit memory write message to an adjacent node on the T3D is 2.7

microseconds and the "payload" bandwidth is 126 Mbytes/sec.

Performing a global sum or other global operation on this type machine involves several phases. Each processor must gain exclusive access to the partial result, make its contribution, determine when all have contributed, and read the final result. A two lock system is commonly used to synchronize these phases. Each processor spins on the first lock until it gains access, locks the lock, makes its contribution, increments a count, unlocks the lock. When all processors have contributed, execution proceeds to the second lock. Here each processor spins until it gains access, locks the lock, reads the final result, unlocks the lock, and continues.

Given an atomic instruction to read and lock the lock, the entry part of this sequence requires four discrete memory operations plus two add operations. Likewise, the exit part of the described sequence requires four memory operations. As I describe in the next chapter, the T3D has a very fast hardware-based barrier mechanism which can be used to notify all processors when all have contributed. This removes the need for the second lock, but processors must still sequentially read the final result, so the total number of memory operations for each processor is five.

Binary software combining trees are often used to reduce the lineup at a single pair of locks. Assuming binary software combining trees are used for both the entry and exit parts, and generously assuming that these contribute no additional overhead time, the total time can be represented as:

$$t_G = 2 * \log N * (5(t_L + M_B * R_T) + t_{ADD}).$$

With a maximum size T3D system of 2048 processors, the t_L and M_B values quoted above, and t_{ADD} negligible as compared to latency time, this expression predicts a time of about 297 microseconds to produce a global floating point sum. As shown in TABLE II, a COP system requires about 14 microseconds to perform a global floating point sum over

4096 processors.

The point here is that in spite of the relatively low latency and high bandwidth of the T3D network, the COP system still provides substantial performance improvement for global operations such as reduction which otherwise require cascaded latency times.

Effect of Some Global Operations on Overall Execution Time

Introduction

In the preceding sections I have shown that the COP system provides substantial speedup for common global operations on a variety of multicomputers and multiprocessors. However, it does little good to make a particular operation 100 times faster, if that operation only represents 1% of the total execution time for a program. Therefore, I will now give some examples of how the speedup provided by a COP reduces the overall execution time of some common scientific programs.

The examples I have chosen represent very general kinds of programs or important "computational kernels" which are used in a wide range of programs. In addition to the applications described in Chapter III, the Jacobi algorithm discussed first is used in solving elliptic partial differential equations and as a preconditioner for other methods such as finite difference. The molecular dynamics program discussed next is an example of the very general N-body problem. The third example is the LINPACK Benchmark which uses LU factorization to solve dense systems of linear equations which are found in many physical problems. The final example is a preconditioned conjugate gradient algorithm, which is another important method for solving systems of linear equations.

A Block-Factored Jacobi Program

To illustrate the overall performance gain provided by a more efficient barrier mechanism, I will assume a slightly modified version of the message-passing Jacobi program shown in Figure 8. For this example assume a message-passing multicomputer with 64 compute nodes arranged in an 8 x 8 mesh topology. Further assume that, instead of assigning rows of grid points to each compute node as was done in the program in Figure 8, a 32 x 32 block of grid points is assigned to each compute node. Block assignment gives greater flexibility in setting grid point aspect ratio. The overall grid for this example contains 65,536 points in a 256 x 256 matrix.

As explained in Chapter III and shown in Figure 8, the three major phases of a message-passing Jacobi program are: 1. Communicate last point values to nearest neighbors, 2. Compute new point values, 3. Aggregate and broadcast the global termination Boolean value. Note that in this program, the third phase also implements a barrier which prevents any compute node from starting a new update loop until all compute nodes have made their contribution to the global Boolean value.

For the communication phase in this example, each compute node has to exchange values with its four nearest neighbors. The time for these exchanges will be $4(t_L + M_B * R_T)$. Assuming each message requires 32 8-byte floating point values and using the previously stated Intel Paragon numbers for T_L, M_B, and R_T, the total time for this phase is 4(25 microseconds + 256 bytes * 0.005 microseconds/byte) = 105 microseconds.

During the Compute phase each compute node will perform four floating point operations (FLOPs) on each of its 1024 points. Assuming that the compute node can perform 100 MFLOPs per second, the total time for this phase is 4 FLOPS/point * 1024 points * 1/100,000,000 FLOPs/second = 41 microseconds

As discussed previously and shown in TABLE IV, the time required for the Aggregate and Broadcast phase on a square mesh is $2(2\sqrt{N} - 2)(t_L + M_B * R_T)$. Again assuming Intel Paragon values, this gives a time of 700 microseconds.

Figure 27 shows how these time relate to the overall execution time. The sum of the Communicate and Compute times will be multiplied by the number of iterations between convergence checks. This time will be added to Aggregate and Broadcast time, and the result multiplied by the number of loops required for convergence. Since the total time is a linear function of the number of loops, looking at just the time ratio for one loop should be enough to see the overall relationships. Using the previously calculated values and assuming 16 iterations between convergence checks, the time for one loop is 16(105 microseconds + 41 microseconds) + 700 microseconds = 3036 microseconds. With these values, about 23% of the loop time is spent in the Aggregate and Broadcast phase. Even if the problem size is increased so that each compute node has more points or the number of iterations between convergence checks is increased, the percentage of time spent in this phase is substantial.

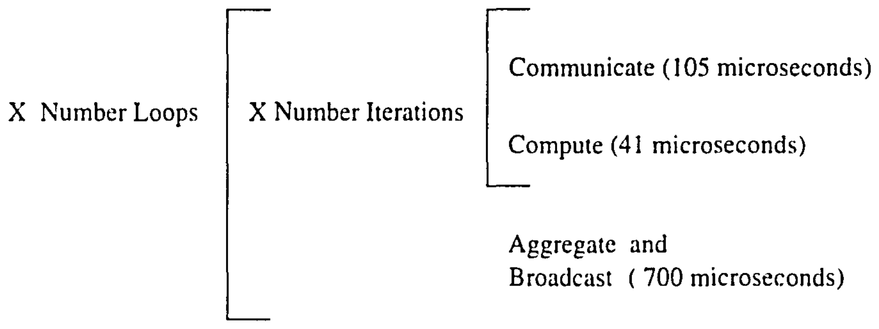

X Number Loops
X Number Iterations
Communicate (105 microseconds)
Compute (41 microseconds)
Aggregate and
Broadcast (700 microseconds)

Figure 27. Execution time relationships for Jacobi program

As shown in TABLE II, a single level COP system can perform the Aggregate and Broadcast phase for 64 compute nodes in about 2.2 microseconds. Inserting this value in place of the 700 microseconds in the above expression reduces the time required for each loop to about 2338 microseconds, which is a 23% reduction. Due to the linear relationship, this 23% reduction in execution time should extend to the overall program.

Incidentally, note that the communicate phase of this program provides an example of the type of communication for which the COP system is not generally as efficient as the main interconnection network. In this phase each compute node exchanges four 256-byte messages with its four nearest neighbors. Since there is no contention for these messages on the compute node mesh network, all of the nodes can be sending and receiving these messages at the same time. As shown above, the total time is just the time for four messages, or for the values given, 105 microseconds.

With a COP system, the timing for this operation is the same as the time for an all-to-all broadcast. As shown in TABLE II, the time required for an 8-byte all-to-all broadcast to 64 nodes is 13 microseconds. If the channel lock feature is not invoked, the total time required is just (13 microseconds/8 bytes) x 256 bytes = 416 microseconds. This example shows well the advantage of the point-to-point main network for this type communication.

A Molecular Dynamics Program

To illustrate the role played by the global sum operation in important scientific programs, I will use a molecular dynamics program described by Clark [Clark92a]. According to Clark the purpose of this type program is to calculate the motion of each atom in a group, as determined by the forces exerted by all the other atoms in the group. Again according to Clark, these programs repeatedly compute the total force on each atom and then use Newton's laws of motion to determine the new position and velocity for each atom. The six major parts of the program are Non-Bonded force calculation, Pairlist generation, Shake, Bonded Force calculation, Global Sum calculation, and Load Balancing.

The first six columns in TABLE V show the data collected by Clark when he ran this program for 500 time steps on an Intel iPSC/860 with 1 to 32 compute nodes. Note that as the number of compute nodes used is increased, the times for the Non-Bonded

Force(NBForce) calculation and Pairlist generation(PList) decrease, the Shake and Bonded-Force(BForce) calculation times remain the same, and the Global Sum(GSum) and Load Balancing(LBal) times increase. As the number of nodes is increased, the total execution time asymptotically approaches the sum of the constant and increasing times. To further explore this effect, I extended Clark's graphs to produce the values for 64 nodes and for 128 nodes shown in TABLE V. As expected, increasing the number of compute nodes from 4 to 8 produced a 40% decrease in execution time, but increasing the number of compute nodes from 64 to 128 produced only about a 10% decrease in execution time. To see how adding a COP affects these numbers, let's first look at the Global Sum part of the program.

TABLE V

TIMING IN MINUTES FOR MOLECULAR DYNAMICS PROGRAM

							COP+		COP+		COP+
Nodes	1	2	4	8	16	32	32	64	64	128	128
NBForce	290	170	75	36	19	10	10	5	5	3	3
PList	190	100	50	28	18	10	10	5	5	3	3
Shake	11	11	11	11	11	11	8.5	11	8.5	11	8.5
BForce	5	5	5	5	55	5	5	5	5	5	5
GSum	-	0.8	1.5	2.0	2.1	2.5	0	2.5	0	2.8	0
LBal	0	0.2	0.3	0.4	0.5	1.2	1.2	1.8	1.8	2.5	2.5
Total	500	287	143	82	56	39.7	34.7	30.3	25.3	27.3	22

This program used a total of 6968 molecules, so for the case of 32 compute nodes, each compute node does the computations for about 220. Assuming that each force is represented by three double-precision floating point components, the total time for a COP system to accumulate the 220 forces from each compute node and broadcast the result for 500 time steps is 500 steps x 220 forces/step x 3 components/force x 4.7 microseconds/global sum = 1.6 seconds. This is nearly a factor of 100 reduction from the 2.5

minutes which Clark measured, and is essentially zero as compared to the other times in the 32 node column of TABLE V.

Again according to Clark, the Shake part of the program uses a Jacobi algorithm. Assuming that a COP can improve this part of the program by 23% as in the preceding Jacobi example, the time required for the Shake part would then be about 8.5 minutes instead of 11 minutes.

Load balancing usually involves some global operation that the COP might benefit, but the Clark's paper did not contain enough detail in this area to determine if this was indeed the case.

As shown in TABLE V, the time for the program to execute on a 32 node system with a COP is about 34.7 minutes, which is about a 12.6% improvement over the time for the same number of nodes without a COP. Although this represents some improvement, the real benefit of the COP becomes more obvious as the number of computes nodes is increased. For example, just doubling the number of compute nodes from 64 to 128 produces an improvement in execution time of about 9.9%. Doubling the number of nodes from 64 to 128 and adding a COP gives a 27.4% reduction in execution time. In other words, the combination gives about three times as much improvement as simply doubling the number of compute nodes.

Since the COP system decreases the fixed, sequential parts of the program, it not only decreases overall execution time, but also, as predicted by the previously cited work of Driscoll and Daasch, it increases the number of compute nodes which can be beneficially used on the program.

As a further point of reference, the last two columns in TABLE V show that just adding a COP to a 128 node system reduces the execution time from 27.3 minutes to 22 minutes, which is about 20%. The question that may occur here is whether a 20% gain is worth the effort of adding a COP. Although the program described here runs for minutes, a larger number of atoms would make the program run for days or months. If adding a COP system reduces the execution time for this kind of program by 20% from 10 days to 8 days, it not only makes the system available two days earlier, but it also decreases the chances of a machine failure during the program run.

Linear Algebra Programs

As an example of the importance of broadcast or multicast I will use an implementation of the LINPACK benchmark described by van de Geijn [Geijn91a]. This program uses LU factorization to solve a system of linear equations. Two important parts of the LU factorization are broadcast of the column panel and broadcast of the row panel. van de Geijn reports that when this program was run on a 128 node Intel Delta (predecessor of the Paragon) machine, almost a third, or 14 seconds of the 45 second, total execution time was spent in these operations. The Delta used was an 8 x 16 mesh, so the diameter, D, was 22. The time to broadcast a single 8 byte value with a minimum spanning tree then is $D(t_L + M_B * R_T)$. Generously assuming that the Delta values for T_L and R_T are comparable to those of the Paragon, the time to broadcast a single 8 byte value is about 550 microseconds. A two level COP system can broadcast an 8 byte value to 128 compute nodes in about 4.4 microseconds. This is less than 1% of the time required using the main message passing network, so the overall performance gain is about 30%.

Preconditioned Conjugate Gradient Program

As mentioned earlier, the preconditioned conjugate algorithm is another method of solving large sparse systems of linear equations. Kumar [Kumar94a] shows that the time for one iteration of this algorithm on a mesh is:

$$T_P = t' \cdot \frac{n}{p} + 3t_s \log p + 6t_h\sqrt{p} + 4t_s + 4t_w\sqrt{n/p}.$$

In this expression t'_c represents computation time, n is the number of equations, p is the number of processors, t_s is the message latency time, t_h is the per node message hop time, and t_w is the time cost to transmit a word.

The first term in the expression represents total computation time, the next two terms represent the communication time required by inner product computation, and the last two terms represent the communication time required during matrix-vector multiplication. Kumar also points out that the second two terms are negligible and can be dropped if the algorithm is executed on a machine that has fast reduction capability. Since the COP provides fast reduction capability, this term can then be dropped when computing the communication time on any machine using a COP.

To give a rough idea of how much this decreases communication cost, I will assume n = 64, p = 64, t_s = 25 microseconds, t_h = 0.04 microseconds, and t_w = .005 microseconds/byte. Substituting these values in the expression above gives $T_P = t'_c + 552$ microseconds. Leaving out the second two terms gives $T_P = t'_c + 100$ microseconds. This represents performance gain of about 82% in the communication requirements for the program.

COP System Benefits Versus Cost

According the 1994 Digital Equipment Corporation catalog, a DEC 3000 Model 600 AXP workstation costs about $21,000, so a 64-node cluster of these workstations would cost about $1,344,000. A rough estimate is that a complete COP system for a 64-node workstation cluster can be built for about $30,000, which is only about 2.2% of the cost of the workstations. The program performance analyses in the preceding sections showed that the COP system provides an average decrease in execution time of about 25%. A 25% improvement in performance for a 2.2% increase in cost seems to be a good ratio.

As another example, a 64-node Intel Paragon multicomputer currently costs about $2,000,000. The $30,000 cost of a COP system for this size machine represents only about 1.5% of the basic system cost. Assuming as before that a COP system provides an average performance gain of about 25%, the benefit-cost ratio of 25% to 1.5% is even better than that projected for a 64-node workstation cluster system.

For still another example, Koeninger [Koeni94a] notes that a 256 processor Cray T3D costs about $9,000,000. The five COP boards required to service a system of this size cost about $150,000, which represents about 1.7% of the cost of the T3D system. Although detailed performance data for the T3D is not yet available for comparison, the relatively small cost of the COP system should maintain a good benefit to cost ratio on this system also.

Chapter Summary

This chapter has shown that the COP system can speed up a variety of global operations by two to three orders of magnitude. The examples in the last section show that the COP provides an average decrease in overall execution time of about 25% for some common scientific programs and computational kernels which utilize COP supported global operations. Furthermore, the COP system provides this 25% improvement for an additional cost of only about 2%.

As pointed out in Chapter IV, the COP system topology does not depend on the topology of the under-lying machine. This means that the COP system can provide this performance gain on "Big Iron" multicomputers, cluster multicomputers, or multiprocessors. In the next chapter, I summarize some of the many other attempts to improve the efficiency of global operations on these machines and where possible compare the performance of these others with that of the COP system.

CHAPTER VI

OTHER WORK ON HARDWARE SUPPORT FOR GLOBAL OPERATIONS

Introduction

The expressions in TABLE IV show that the time required for several common global operations on message-passing distributed memory multicomputers is directly proportional to $D(t_L + M_B * R_T)$. In the last chapter I showed how the COP system reduces D and t_L, and thereby substantially reduces the execution times for global operations. In this chapter I describe and compare other proposed or practiced methods for decreasing these times and for decreasing shared variable access time. To provide some order in the discussion, I divide the work into two main categories as follows.

1. Work aimed at decreasing and/or hiding latency time, increasing network bandwidth, and decreasing remote memory access time.
2. Specialized hardware support for synchronization and/or other global operations.

Attempts to Decrease Message Passing and Memory Access Overhead

1. Work aimed at "Big Iron" multicomputers

Many diverse methods of improving message passing efficiency have been tried for machines in this category. I will concentrate on those that very specifically relate to the COP system.

Hsu and Banerjee [Hsu90a], show the advantages of using a communication coprocessor at each compute node to relieve the main processor of the burden of packetizing, sending, receiving, and depacketizing messages. Perhaps influenced by this work, Intel

Paragon systems use a dedicated i860 as a communication coprocessor at each compute node.

A communication coprocessor allows the main processor to continue computing while a message is being sent if the algorithm allows. This overlap of computation and communication improves the overall efficiency, but the absolute message latency is still present. Therefore, the cascaded latency times of global operations are still a problem.

Including the communication circuitry on the same chip as the processor provides tighter and theoretically more efficient coupling between the two. The custom processor chip used in the nCUBE 2S [Zorpe92a], for example, contains a floating point computation unit, a memory management interface, a message-routing unit, and 14 pairs of direct-memory-access (DMA) channels. One of the 14 DMA channels is used for external input/output. The other 13 DMA channels are used to connect a processor to its nearest neighbors in a hypercube topology which can contain up to 8192 nodes. In this system, a processor sends a message to a neighboring processor by writing the message into the receiving processor's memory on a DMA basis. However, long range communication requires a "bucket brigade" type transfer from one processor's memory to the next along the way. The latency of each transfer adds to the total message delay for both point-to-point and global "tree" communication. Contributing to the latency is the fact that currently, the only access to the network on the nCUBE 2S is through the operating system.

Another example of a processor chip that contains both a computation agent and a communication agent is the Intel iWarp "component" [Borka88a] used in Intel iWarp systems. The communication agent has four independent input ports and four independent output ports. The independence of these ports allows iWarp components to be connected in ring, mesh, or torus topologies for general purpose computing or to be connected in special array configurations suitable for systolic graphics or signal processing algorithms. The wormhole routers in the chip allow messages to pass through a communication agent without disturbing the computation agent in that node. For general message passing, however, there is no actual communication coprocessor, so the main computation element still has the overhead of packetizing and depacketizing messages.

The main contribution to my thoughts by iWarp was provided by its systolic communication mode. In this mode data is rapidly transferred from the computation element in one cell to the computation element in another without going through the memory of either. For this special, direct type of communication the receiving cell explicitly "knows what to do with" the incoming data, so a complex message format is not needed. The dedicated communication links in the COP system provide a similar type of direct communication.

Another machine with processors that integrate computation and communication is the Jellybean Machine or J-Machine [Dally92a], developed as a collaborative project between MIT and Intel Corporation. The Message-Driven Processors(MDPs) in a J-Machine implement a fine-grained, data-flow model of computation. J-machine messages contain a pointer to the computation handler for the message, one data operand, and a pointer to another data operand if needed. When a message arrives at a processor, it is put in a FIFO queue. The specified handler is invoked and the computation performed when the processor reads the message from the queue. Thus, computation is message driven. The advantage of this approach is that the processor does not try to execute an instruction until it has both the instruction and the data for that instruction. Therefore, processors spend their time executing instructions for which they have the data, rather than executing instructions that each require a wait for the required data.

Each MDP in a J-Machine contains an ALU, and Address Arithmetic Unit, a memory controller with ECC, a network interface, and a router. The six ports on the router are used to connect the MDPs in a 3-D mesh topology with up to 64K nodes. According to Dally, the J-machine requires about 3 microseconds for a message send-receive operation, which is quite good, especially since the machine only has a 16.5 Mhz clock. However, the mesh topology of the J-machine requires trees of some sort for global operations and thus suffers the cascaded message latency problem for these operations. Also, since the J-Machine uses small, custom processors and a message-driven programming paradigm, I feel that the mountains of existing software would need major modifications or complete rewriting to efficiently execute on this kind of machine.

The active messages scheme developed by von Eicken [Eicke92a] is a pure software approach to reducing message latency. In some ways this approach is very similar to the message driven approach of the J-machine, but in other ways it is quite different. An active message contains both data and a pointer to the handler process which will read the message on the receiving node and integrate the data into an ongoing computation. The difference here is that the handler process simply reads the message off the network and puts the data in the correct place; it does not perform the actual computation as does the handler in a J-Machine.

von Eicken [Eicke92a] reports that the active message approach reduced the message send-receive latency for the Thinking Machines CM-5 from 86 microseconds to 23 microseconds. However, the active message approach assumes SPMD only program execution, and therefore does not provide any of the protection mechanisms required for MPMD program execution.

Another software based attempt to reduce the message-passing latency is work of Rosing [Rosin93a] at ICASE. Rosing and his colleagues replaced the NX/2 operating system in an Intel iPSC/860 with an operating system which allows user programs to directly access the communication hardware, rather go through the usual operating system calls. According to Rosing this reduces the minimum message latency from 70 microseconds to about 25 microseconds. This approach does reduce the latency, but it also places the responsibility for message formatting, etc. on the application programmer.

Also, since this direct access circumvents the usual protection mechanisms, it can only be used for SPMD program execution.

Two important points emerge from the preceding discussions. The first is that even with the message latency drastically reduced as in the J-Machine, active messages, or Rosing's work, the cascaded latencies of the tree algorithms required for global operations on many of these machines still contribute substantially to overall execution time. Second, if user programs are to be given direct network access in order to reduce latency time, some mechanism must be implemented to provide protection so that the overall machine can be used for MPMD program execution as well as for SPMD program execution. The COP system both reduces the time cost of global operations and for these operations; it also provides the protection mechanisms needed for MPMD program execution.

2. Work related to cluster multicomputers

As described in Chapter II, several of the latest research and commercial cluster multicomputers use interconnection networks with crossbar switches or wormhole routers instead of shared medium networks such as ethernet or FDDI. These newer networks improve system scalability and provide greater bandwidth.

One cluster multicomputer that uses a unique network interface to reduce message latency is the Princeton Shrimp system described by Li [Blumr93a]. Shrimp is actually an example of distributed shared virtual memory system, but since it uses Intel Pentium workstations as compute nodes, I will discuss it in this section.

Message passing in the Shrimp system is separated into two phases. During the mapping phase, an operating system call is used to set up a communication path between two nodes. This is done by mapping the same page of virtual memory to the sending and receiving nodes. The second phase is the actual communication. When a compute node writes a word to a location in the shared virtual page, a special network interface chip snoops the transaction on the compute node memory bus and generates a message to the same virtual address in the remote node. The network interface essentially functions as a snooping communication coprocessor. The point here is that only the mapping phase requires operating system involvement to enforce protection. User programs can have direct access to the virtual memory mapped page assigned to them, and thus send messages with low latency. Li has reported a latency time of about 2 microseconds for the communication phase of a single point-to-point message. If a particular communication path is used over and over, the mapping phase is only needed for the first use.

As reported by Li, however, a weak point of Shrimp is broadcast and other global operations. The Shrimp system uses an Intel Paragon router backplane to implement its interconnection network. Since this backplane has a 2-D mesh topology, spanning trees are needed to implement broadcast and other global operations. Not only do message latencies cascade along the branches of these trees, but also multiple operating system calls are required to set up the mapping for the trees before the global operation and to restore the previous communication links after the operation. I feel that addition of a COP system to Shrimp could greatly improve its performance on global operations.

I found no work proposing a system such as the COP for cluster multicomputers.

3. Work related to distributed shared memory machines

The discussion in Chapter III showed that a major problem on distributed shared memory systems is the memory "hot spots" caused by, for example, contention for a lock variable.

One of the relatively early attempts to alleviate this problem was the combining network in the NYU Ultracomputer [Gottl83a]. The Ultracomputer uses an Omega type, multistage interconnection network to interface processors with the memory modules. The network is enhanced so that it can combine access requests directed at the same memory location. Furthermore, if several processors need to receive, for example, successive values of an array index variable, the network circuitry will fetch the current value of the variable from memory, add a specified value for each request, return successive values to each of the requesting processors, and write the final value of the variable in memory. As shown by Almasi and Gottlieb [Almas94a], the fetch-and-add primitive can also be used to implement barriers. The importance of this fetch-and-add mechanism is that for simultaneous accesses, it provides "answers" to all the processors in about the same time as required for a single request. This means that the entire network is not tied up while each request is serviced sequentially. However, if the requests do not all arrive at the same time, they must still be serviced serially so the combining capability is in that case not beneficial.

A somewhat similar machine, the IBM RP3 [Pfist85a], discussed briefly in Chapter II has one high speed omega network for general memory access and another omega network with combining capability as previously described for the NYU Ultracomputer omega network. A major problem with both of these is cost and complexity of the network(s) when extended to a large number of processors. The COP system provides fast fetch-and-add capability with no spinning or traffic on the main interconnection network and provides other capabilities as well.

The Stanford DASH [Lenos93a] distributed shared memory system discussed in Chapter II has both fetch-and-increment capability, and queue-based locks to assist with synchronization. Lenoski [Lenos93a] reports that using fetch-and-increment decreases execution time for a single barrier to about 30 microseconds for a 44 processor machine as compared to 375 microseconds for a pure linear barrier on the same machine.

Queue-based locks on the DASH are handled as part of the directory logic. The directory keeps track of which processors are spinning on a particular lock variable and, when the lock becomes available, it is randomly granted to one of the waiting processors.

The COP system can easily implement both fetch-and-increment and queue-based locks if desired, but the COP system directly provides fast barriers which is one of the main uses for these.

The Cray T3D distributed shared memory system, also discussed in Chapter II, has several mechanisms to reduce remote memory access times. These include a one-word prefetch queue and a remote processor store which queues up to four writes directed to a remote location. The T3D also has a block transfer engine which efficiently manages transfers of large blocks. These mechanisms are helpful for many operations, but as pointed out by the performance estimates in the last chapter, they are not particularly helpful for several of the global operations performed by the COP. For synchronization, however, the T3D does have a very fast hardware-based barrier mechanism which I discuss later.

Another way of coping with the latency time of remote memory accesses or synchronization is to swap to another lightweight process or thread of execution while waiting. Examples of systems which use this approach are the Tera machine [Alver90a] proposed by Tera Computer Company, and MIT's Alewife [Agarw90a] system which I mentioned in a discussion of cache directories in Chapter II. In the Alewife system, the modified SPARC processor at each node maintains four program contexts in hardware and can very quickly swap contexts. When a cache miss or synchronization wait occurs, a software trap forces a context swap to another thread if one is available. This scheme assumes that overall program algorithms contain enough parallelism to support multiple threads and that a compiler which can generate the required threads is available. For a program such as the Jacobi program described in Chapter III, I am not sure what you have the processors do while waiting for all to complete an iteration as required by the algorithm. With no other threads to swap to, the processors must just spin until all have checked in at the barrier. In this case, the speedup provided by the COP system still reduces overall execution time as projected for other systems.

The Alewife system does have a full/empty bit associated with each memory location to provide fine-grained synchronization as was done in the Delecor HEP. A read of an empty location or a write to a full location can generate a trap and cause a context switch to keep the processor busy while waiting for synchronization. This reduces spinning on memory locations, but again requires available threads to actually improve processor utilization.

The main point from the research reported in this section is that most if not all of these distributed shared memory systems could benefit significantly from the addition of a COP system for access to shared variables and other global operations. In the next section I discuss and compare other proposed and existing hardware support for these operations.

Specialized Hardware for Synchronization and Other Global Operations

Machines with hardware support for barriers

The first topic I researched in this area was dedicated hardware to directly implement barriers in shared memory machines. The major works I initially found in this category were [Gupta89a], [Beckm90a], [Hwang91a], [OKeef90a], and [Ghose91a]. All of these approaches are essentially based on a large AND gate with a processor connected to each of its inputs. When each processor reaches the barrier, it asserts a signal connected to its input on the AND gate. When all of the processors have reached the barrier, the signal from the output of the AND gate is broadcast back to all the processors to signal the barrier exit phase. Some of these approaches insert flip-flops in series with the AND gate inputs and/or output to better control signal timing. The design proposed by Ghose [Ghose91a] uses an input register with maskable bits so that a barrier can be implemented for a specified subset of processors.

The Thinking Machines CM-5 [Leise92a], also has a subnetwork of this type for barrier synchronization. Likewise, the Cray T3D [Kessl93a] has a separate, binary tree network and an array of AND gates to implement barrier synchronization.

Using simple dedicated hardware as done in these systems can implement a barrier in a very short time. According to Kwan [Kwan93a], for example, the CM-5 can implement a barrier over 256 processors in about 5 microseconds, and according to Koeninger [Koeni94a], the CRAY T3D can implement a barrier over 2048 processors in 330 nanoseconds. However, these specialized barrier networks tend to be very limited in their capabilities. In a recent personal conversation, Kent Koeninger, Software Program Manager for the Cray T3D, told me that while the barrier network works well if used for the whole machine, it has two major problems for smaller partitions. First, the binary tree topology of the barrier network does not map easily into the cubes of processors usually assigned to different users. Second, the barrier network does not have the protection mechanisms required for MPMD program execution. Kent further said that due to the limited applicability of this barrier-only network, it will probably not be included in future systems.

The COP system implements other important operations as well as barriers, provides MPMD required protection, and allows processor subset participation in barriers so its cost is more easily justified. However, the generalized barrier mechanism which provides these capabilities in the COP is in some cases slower than the specialized "big AND" approach. For comparison, a two level COP system can implement a barrier over 256 compute nodes in 4.5 microseconds. This is faster than the 5 microseconds quoted above for the CM-5, but obviously much slower than the 330 nanoseconds quoted above for the T3D. It seems dubious that it is worth the cost of adding the extra hardware and signal lines to improve just the barrier efficiency by this amount. In the next section I discuss systems which include specialized hardware for other global operations as well.

Machines with hardware support for other global operations

The Sequent Balance(R) is a bus-based, shared memory system such as that shown in Figure 2. Each processor in a Balance has an associated Serial Link and Interrupt Controller(SLIC) IC. The SLIC chips communicate with each other over a secondary network which consists of a bit-serial bus. A priority resolution scheme arbitrates multiple attempts to access the serial bus.

According to Beck [Beck87a] the main functions of the SLIC chips are to distribute interrupt service requests among the processors for load balancing, to provide interprocessor synchronization, and to assist in system configuration and control.

When an interrupt is injected into the system, SLIC chips arbitrate to determine which one will accept the interrupt. Once a SLIC chip has accepted an interrupt, it will no longer contend for any other interrupts until it finishes servicing the first. Interrupt service requests are thus distributed among the processors.

To provide low level synchronization each SLIC has 64 single-bit "gates" which, along with test-and-set primitives, can be used to implement locks. Gates are duplicated in all the SLICs, so a processor spins on the local copy of a gate rather than over the SLIC bus or over the main memory bus. This reduces synchronization traffic on the main bus, but requires periodic broadcasts to keep the gates current.

The idea of using a low-cost secondary network and SLIC chips to move synchronization traffic and other traffic off the main memory bus seems good, but contention for the bit-serial bus would rapidly create a bottleneck if the number of processors is increased beyond the Balance's maximum of 30. The COP system uses individual point-to-point communication links to avoid this problem. Also, the COP system provides many more operations and reduces duplication by allowing one COP to service a group of compute nodes.

Another system that contributed to my thinking was the PAX machine. According to Hoshino [Hoshi89a], the PAX-128 is an 8x16 torus of Processing Units built with 8-bit commodity microprocessors. The system uses a general purpose minicomputer as a host for user interface program development. The system also contains a Control Unit microcomputer which transfers programs and data between the host and the processing units, assists in synchronization, and does global broadcasts. The Control Unit is connected to all the Processing Units by a parallel bus and Processing Units are connected to each other by parallel buses. Communication between the Control Unit and Processing Units and communication between Processing Units is done via shared memory address spaces. To send program code or data to a Processing Unit, for example, the Control Unit selects the target Processing Unit and writes the code and data in a section of the Processing Unit's memory on a DMA basis. Long range communication between Processing Units is done with a "bucket brigade" approach where the "message" is passed along from one Processing Unit's memory to the next until it reaches the destination Processing Unit.

In the PAX-128 the Control Unit and the Extended Control Unit bus supply a mechanism for global broadcast to all or to a subset of the Processing Units. For a broadcast to all the Processing Units, the Control Unit halts all the processors, writes the broadcast data at the same address in each Processing Unit's memory, then restarts the Processing Units. The Control Unit also provides a mechanism for global synchronization. When a Processing Unit reaches a barrier, it writes a code into its status register and halts. The Control Unit continuously monitors all the status registers and when all the Processing Units have checked in, the Control Unit sends out a signal which restarts all the Processing Units. This mechanism is very similar to the "big AND" mechanism described previously.

Although the PAX-128 used 8-bit Processing Units and a relatively slow 2 MHz processor clock frequency, its hardware barrier mechanism allowed it to service a barrier in about 25 microseconds and its control unit bus allowed it to broadcast N bytes of data to all the Processing Units in 4N + 148 microseconds. I felt that these numbers strongly supported my idea of adding a secondary network which is optimized for global communication and some centralized hardware to perform global operations.

Another system with specialized hardware for global operations is the Thinking Machines CM-5 [91a]. A CM-5 machine has two types of processors, compute processors and control processors. The control processors in a CM-5 are SPARC workstation type computers which run a Unix-like operating system. The control processors are used to partition the system, distribute programs to compute processors, run diagnostics, and manage I/O operations.

The compute processors and the control processors in a CM-5 are connected by a data network, a control network, and a diagnostic network. The data network connects the compute and control nodes in a "fat tree" topology as described in Chapter II. The control network and the diagnostic network connect compute processors and control processors in binary tree topologies with the processors at the leaves. Communication on all three networks is by message passing. The data network supports only point-to-point communications. The control network supports both point-to-point and broadcast communications.

Each compute processor and each control processor has a network interface unit which allows it to communicate over the three networks. The control network section of each network interface unit contains three broadcast functional units, a combine type functional unit, and three global operation type functional units. The three broadcast units allow user type, supervisor type, or interrupt type messages to be broadcast to all processors. The combine unit is used for global operations which involve integer addition, bitwise logical OR, bitwise logical EXOR, MAX, and MIN. The three global operation functional units are mostly used to implement barriers as described previously.

Superficially, the COP system may seem somewhat similar to the CM-5 control network, but the COP is really quite different and has many prominent advantages over the CM-5.

First, the COP system is much more versatile. A user partition in a COP system can contain any number of compute nodes and the COP protection mechanisms allow both SPMD and MPMD program execution. The minimum user partition in a CM-5 system is 32 nodes and only SPMD program execution is allowed within a partition. Also, the COP mask system allows all or any subset of the compute nodes in a partition to participate in, for example, a barrier or broadcast operation. In a CM-5 system all the nodes in a partition must participate in a barrier or broadcast operation. Furthermore, the COP network is asynchronous, so the COP system can be easily used with workstation cluster multicomputers as well as with "Big Iron" machines. The CM-5 networks are synchronous and are therefore not applicable to cluster systems because these may include workstations with very different clock frequencies. The COP system provides a very efficient mechanism for maintaining and accessing globally shared write-able variables; the CM-5 does not.

Also, the COP system is faster than the CM-5 system for global operations. As shown in Table II, a single level COP system can broadcast an 8-byte word, perform a global integer sum, or execute a barrier in 2.2 microseconds. According to Kwan[Kwan93a], a CM-5 requires 12 microseconds, 6 microseconds, and 5 microseconds respectively for these operations on the same size partition. Furthermore, as stated in Chapter VI, a COP system can broadcast a 64 Kbyte vector to 64 compute nodes in about 1600 microseconds. The performance parameters reported in Kwan indicate that this same broadcast would take about 80,000 microseconds or 80 milliseconds on a 64 node CM-5 machine.

Finally, the centralized COPs do not require the duplication of low level functional units at each compute node as is done in the CM-5.

Chapter Summary

Although several of the references cited here contributed to my thinking as noted, no other single system duplicates or more efficiently implements the versatile functionality of the COP system.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Conclusions

My hypothesis was that global program operations on a multicomputer or multiprocessor could be significantly speeded up by the addition of a wide-tree secondary network and one or more centralized processors optimized for these operations. The analyses in Chapters V and VI conclusively demonstrate that the COP system's combination of speed and versatility exceeds that of any other system for many common global operations. Furthermore, the program examples in Chapter V show that speeding up these operations decreases overall program execution time by an average of about 25% at an additional system cost of only about 2%. Specifically, the COP system provides the following advantages.

The very high bandwidth bit-serial interconnect makes the COP system equally applicable to "Big Iron" multicomputers, workstation cluster multicomputers, and distributed shared memory machines. The independent links between compute nodes and COPs provide a high degree of fault tolerance, because, if one link fails, the rest of the COP system can continue to operate. The mask based partitioning and protection approach used in the COP makes it independent of the underlying network topology. Also, as shown in TABLE II, the wide-tree topology reduces the number of interconnect links and the number of COPs required as compared to a narrower tree topology. The wide topology also eases programming because mask word widths match data path widths of current machines.

The COP system can easily be added existing systems as an upgrade. The compute node to COP-Network interface circuitry can be built on a small board for insertion in a system bus slot, insertion in a SIMM slot, or piggy-backing on a compute node board. For new system designs, the small amount of circuitry required for this interface can easily be included directly in the board design.

The entire COP system can be implemented with existing technology. Except for the Spanceivers, the COP system uses standard components such such as buffers, latches, multiplexers, SRAM, ALUs, and FPUs. The controllers can be implemented with high speed programmable logic devices to avoid producing a custom ASIC. A rough estimate is that a complete 64-node, single level COP system could be built for about $30,000. However, if less than the full 64 channels are needed, the cost can be significantly decreased by inserting Spanceivers only for the channels needed. As described in Chapter IV, the COP controller can be jumpered to only cycle through the number of interfaces actuallly used.

Since the COP system does not change the basic programming paradigm, but simply provides an alternative, fast mechanism for implementing common global operations, the main software component needed to utilize it is a library of COP access routines. A smart compiler can then insert calls to COP access routines where appropriate. COP access routines could also be included in an MPI or PVM implementation. The User/Supervisor and process level protection mechanisms in the COP system make it compatible with MPMD program execution as well as with SPMD program execution.

The COP system provides many benefits, but it also has some limitations. As pointed out in Chapter V, for example, the COP system is less efficient than standard message passing for all-to-all personalized communication on a 2-D mesh multicomputer. However, operations for which the COP does excel suggest several other possible uses which I have not previously discussed here.

The fast broadcast capability of the COP system, for example, could be used to broadcast global clock values during distributed simulations. This strong capability could also perhaps be used to broadcast updates of shared objects as required by the ORCA [Tanen92a] language. Still another use of the COP broadcast speed might be to distribute load index values for the type of dynamic process level load balancing proposed by Xu and Hwang [Xu90a].

A totally different application of the COP system is performance monitoring. The COP network provides a very fast mechanism for exporting event counter data and other performance data to external recording equipment. Also, for distributed debugging, the COP system provides a way of exporting breakpoint data for external examination.

The point here is that the COP is a new and powerful system resource which will suggest other uses as we become more familiar with it. This point leads into the next section which describes my projected future work.

Future Work

The COP system described here presents many potential research topics. I am particularly interested in the application of the COP system to workstation cluster multicomputers because I feel that they will be the most widespread supercomputing platforms in the near future and they can potentially derive the most benefit from the COP system.

In reporting on a year-long effort at porting the NAS benchmarks and other scientific programs to PVM for execution on a cluster of workstations at NASA Ames Research Center, Castagnera [Casta94a] has pointed out the severe limitations imposed by the high network latency. As discussed in Chapter II, others are working on improving the general network performance for cluster systems, but no one seems to be working specifically on improving performance for the global operations addressed by the COP system.

Therefore, I feel that it is time to move from the "proof of concept" phase described in this document to the actual design and implementation phases for the COP system. Assuming suitable funding can be found, these phases might proceed as follows.

1. Since PVM seems to be very universally used, research the requirements for replacing PVM or underlying MPI calls with COP calls.

2. Research the requirements for interfacing with the available workstation buses.

3. Design, simulate, and build prototypes of the compute node to COP network interface and the actual COP. If the clock frequency for this prototype system is reduced to 50 MHz, a large portion of the circuitry can be implemented in FPGAs or CPLDs.

4. Measure and compare the performance of the system using the PVM programs available from NAS and/or locally generated programs.

5. Attempt to form a partnership with a company or other research group which is working on improving the overall network performance and try to integrate our efforts.

REFERENCES

94a. "Tandem Defines System-Area Cluster," *Electronic Engineering Times*, CMP Publications, Manhasset, NY, September 5, 1994.

Agarw90a.
Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *The 17th Annual International Symposium on Computer Architecture*, pp. 104-114, Seattle, Washington, May 28-31, 1990.

Ali94a.
M. Ali and J.M. Jagadeesh, "Express 3.2.5 for Workstations," *Computer*, pp. 73-75, IEEE Computer Society, August 1994.

Almas94a.
George S. Almasi and Allan J. Gottlieb, *Highly Parallel Computing, Second Edition,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

Alver90a.
Robert Alverson, David Callahan, Allan Porterfield, Daniel Cummings, Burton Smith, and Brian Koblenz, "The Tera Computer System," *1990 International Convereence on Supercomputing*, vol. 18, no. 3, pp. 1-6, 1990.

Amdah67a.
G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities"," *Proc. AFIPS Spring Joint Computer Conference, 30*, pp. 483-485, Atlantic City, N.J., April 1967.

Archi86a.
James Archibald and Jean–loup Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor," *Transactions on Computer Systems*, vol. 4(4), pp. 273-298, ACM, November 1986.

Barne93a.
M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, "Global Combine on Mesh Architectures with Wormhole Routing," *7th International Parallel Processing Symposium*, April 13-16, 1993.

Beck87a.
Bob Beck, Bob Kasten, and Shreekant Thakkar, "VLSI Assist for a Multiprocessor," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 10-20, The Computer Society of the IEEE, Palo Alto, CA, October 5-8, 1987.

Beckm90a.
Carl J. Beckman and Constantine D. Polychronopoulos, "Fast Barrier Synchronization Hardware," *Supercomputing '90 conference*, New York, NY, Nov. 1990.

Bell94a.
Gordon Bell, "Scalable, Parallel Computers: Alternatives, Issues, and Challenges," *International Journal of Parallel Programming*, vol. 22-1, pp. 3-46, Plenum Publishing Co., 1994.

Benne90a.
J. Bennett, J. Carter, and W Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-specific Coherence," *Proceedings 1990 Conference Principles and Practices of Parallel Programming*, pp. 168-176, ACM Press, New York, NY, 1990.

Berts89a.
Dimitri P. Bertsekas and John N. Tsitsiklis, *Parallel and Distributed Computation - Numerical Methods,* Prentice Hall, Englewood Cliffs, NJ, 1989.

Bisia90a.
Roberto Bisiani and Mosur Ravishankar, "PLUS: A Distributed Shared Memory System," *Proceedings 17th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, Seattle, WA., May 1990.

Blood88a.
William R. Blood, Jr., *MECL System Design Handbook,* Motorola Semiconductor Products Inc., Phoenix, AZ, 1988.

Blumr93a.
Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Technical Report*, Department of Computer Science, Princeton University, November 1993.

Borka88a.
Shekhar Borkar and Others, "iWarp: An Integrated Solution to High-Speed Parallel Computing," *Proceedings of the 1988 Supercomputing Conference*, pp. 330-339, Orlando, FL, Nov 14-18, 1988.

Butle92a.
R. Butler and E. Lusk, "Users' Guide to the p4 Programming System," *Technical Report ANNL-92/17*, Argonne National Laboratory, Argonne, Il., 1992.

Carri89a.
Nicholas Carriero and David Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32(4), pp. 444-458, April 1989.

Casta94a.
K. Castagnera and Others, "NAS Experiences with a Prototype Cluster of Workstations," *Proceedings, Supercomputing '94*, Washington, D.C., November 14-18, 1994.

Chitt90a.
S. Chittor and R. Enbody, "Hypercubes vs. 2D meshes," *Proc. SIAM Fourth Annual Conference on Parallel Processing for Scientific Computing*, pp. 313-318, 1990.

Clark92a.
Terry W. Clark, "Evaluating Parallel Languages for Molecular Dynamics Computations," *Proceedings Scalable High Performance Computing Conference SHPCC-92*, pp. 98-105, April 26-29, 1992.

CLSei93a.
C.L.Seitz and W. Su, "A Family of Routing and Communication Chips Based on the Mosaic," *Proceedings of Washington Symposium on Integrated Systems*, Seattle, WA, 1993.

Cohen94a.
Danny Cohen and Others, "ATOMIC: A High-Speed Local Communication Architecture," *Technical Report*, USC/Information Sciences Institute, Marina del Rey, CA, January 1994.

Corpoa.
Intel Corporation, *The IPSC/2 User's Guide.*

Dally90a.
W. J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers*, vol. 38, pp. 775-785, June 1990.

Dally92a.
William J. Dally and Others, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, pp. 23-39, April 1992.

Donga93a.
Jack J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley, "A Users' Guide to the BLACS," *Technical Report*, University of Tennessee, December 1993.

Drisc95a.
Michael A. Driscoll and W. Robert Daasch, "Accurate Predictions of Parallel Program Execution Time," *Journal of Parallel and Distributed Computing*, Early 1995.

Eicke92a.
Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Proceedings 19th Annual International Symposium on Computer Architecture*, pp. 256-266, ACM, May 1992.

Ewing93a.
R.E. Ewing and Others, "Distributed Computation of Wave Propagation Models Using PVM," *Proceedings Supercomputing 93*, ACM/IEEE, Portland, OR, November 1993.

Flynn72a.
Michael J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948-960, 1972.

Geijn91a.
Robert A. van de Geijn, "Massively Parallel LINPACK Benchmark on the Intel Touchstone DELTA and iPSC/860 Systems," Unpublished Report, University of Texas, pp. 1-9, 1991.

Geijn94a.
Robert van de Geijn and Others, "InterCom: Lean Mean Collective Communication Machine," *Intel On-Line*, September 1994.

Ghose91a.
Kanad Ghose and Der-Chung Cheng, "Efficient Synchronization Schemes for Large-Scale Shared-Memory Multiprocessors," *Proceedings of the 1991 International Conference on Parallel Processing*, pp. I:153-170, August 12-16, 1991.

Gottl83a.
Alan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. C-32, no. 2, pp. 175-189, February 1983.

Grims93a.
Andrew S. Grimshaw, "Easy-to-Use Object-Oriented Parallel Programming with Mentat," *Computer*, pp. 39-51, IEEE Computer Society, May 1993.

Gupta89a.
Rajiv Gupta and Michael Epstein, "Achieving Low Cost Synchronization in a Multiprocessor System," *PARLE '89 Parallel Architectures and Languages Europe*, vol.

1, pp. 70-84, 1989.

Gusta88a.
John L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, May 1988.

Henne90a.
John L. Hennessy and David A. Patterson, *Computer Architecture: a Quantitative Approach,* Morgan Kaufmann Publishers, Inc., 1990.

Hoshi89a.
Tsutomu Hoshino, *PAX Computer: High-Speed Parallel Processing and Scientific Computing,* Addison-Wesley Publishing Company, 1989.

HPCC94a.
National Coordination Office for HPCC, *HPCC:Technology for the National Information Infrastructure(FY 1995 Blue Book),* Washington, DC, 1994.

Hsu90a.
Jiun-Ming Hsu and Prithviraj Banerjee, "A Message Passing Coprocessor for Distributed Memory Multicomputers," *Supercomputing '90 conference*, pp. 1-10, New York, NY, Nov. 1990.

Hwang91a.
Kai Hwang and Shisheng Shang, "Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessors," *1991 International Conference on Parallel Processing*, pp. I:171-175, 1991.

Johnsa.
S. Lennart Johnsson, "Communications in Network Architectures," *VLSI and Parallel Computation*, Morgan Kaufmann Publishers, Inc., San Mateo, CA.

Jorda78a.
Harry F. Jordan, "A Special Purpose Architecture for Finite Element Analysis," *Proceedings of the 1978 International Conference on Parallel Processing*, pp. 263-266, 1978.

Jul88a.
E. Jul, N. Levy, N. Hutchinson, and A. Black, "Fine Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134-135, Feb. 1988.

Kessl93a.
R.E. Kessler and J. L. Schwarzmeier, "CRAY T3D: A new Dimension for Cray Research," *Proceedings COMPCON Spring '93*, ACM Computer Society Press, San Francisco, CA, February 1993.

Koebe89a.
Charles Koebel and Piyush Mehrotra, "Supporting Shared Data Structures on Distributed Memory Architectures," CSD-TR 915, Computer Sciences, Purdue University, October 1989.

Koeni94a.
R. Kent Koeninger, Mark Furtney, and Martin Walker, "A Shared Memory MPP from Cray Research," *Digital Technical Journal*, pp. 8-21, Digital Equipment Corporation, Spring 1994.

Kumar94a.
Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing, Design and Analysis of Algorithms,* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

Kung91a.
H.T. Kung and Others, "Network-Based Multicomputers: An Emerging Parallel Architecture," *Proceedings Supercomputing 91 Conference*, ACM/IEEE, Alberquerque, NM, 1991.

Kwan93a.
Thomas T. Kwan, Brian K. Totty, and Daniel A. Reed, "Communication and Computation Performance of the CM-5," *Proceedings Supercomputing '93*, pp. 192-201, ACM/IEEE, November 1993.

Lee94a.
Yvonne L. Lee, "Digital Alpha Line Gets PCI Makeover," *Info World*, vol. 16, no. 45, p. 33, November 7, 1994.

Leise92a.
Charles E. Leiserson and Others, "The Network Architecture of the Connection Machine CM-5," *Proceedings - ACM Symposium on Parallel Algorithms and Architectures*, 1992.

Lenos93a.
Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy, "The DASH Prototype:Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, pp. 41-61, January 1993.

Leste93a.
Bruce P. Lester, *The Art of Parallel Programming*, pp. 120-147, Prentice-Hall, Inc, 1993.

Li91a.Jingke Li and Marina Chen, "Compiling Communication-Efficient Programs for Massively-Parallel Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 361-375, July 1991.

Li89a.Kai Li and R. Shaefer, "A Hypercube Shared Virtual Memory System," *Proceedings International Conference Parallel Processing*, pp. 125-132, Pennsylvania State University Press, University Park, PA, 1989.

Manch94a.
Robert J. Manchek, "Design and Implementation of PVM Version 3," *MS Thesis*, University of Tennessee, Knoxville, TN, May 1994.

Manji16a.
Narig Manjikian and Others, "High Performance Parallel Logic Simulations on a Network of Workstations," *Proceedings 1993 Workshop on Parallel and Distributed Simulation*, ACM/SCS/IEEE, San Diego, CA, May 116-19 1993.

Mori93a.
Shin-ichiro Mori and Others, "A Distributed Shared Memory Multiprocessor: ASURA -Memory and Cache Architectures," *Proceedings Supercomputing '93*, pp. 740-749, IEEE Computer Society Press, Portland, OR, November 1993.

Motor94a.
Motorola, *High Performance ECL Data Book*, 1994.

Nitzb91a.
Bill Nitzberg and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, pp. 52-60, IEEE Computer Society, August 1991.

Numri94a.
Robert W. Numrich, Paul L. Springer, and John C. Peterson, "Measurement of Communication Rates on the Cray T3D Interprocessor Network," *Proceedings*

International Conference, High-Performance Computing and Networking, vol. II, pp. 150-157, Springer-Verlag, Munich, Germany, April 1994.

OKeef90a.
Matthew T. O'Keefe and Henry G. Dietz, "Hardware Barrier Synchronization: Static Barrier MIMD (SBM)," *Proceedings of the 1990 International Conference on Parallel Processing*, pp. I:35-46, August 13-17 1990.

Pfist85a.
G.F. Pfister and Others, "The IBM Research Parallel Processor Prototype(RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

Physi91a.
U.S. Office of Science and Technology Committee on Physical, Mathematical, and Engineering Sciences, *Grand Challenges: High Performance Computing and Communications*, 1991.

Produ93a.
Motorola Semiconductor Products, Inc, *MC100SX1451 Autobahn Spanceiver Data Sheet*, 1993.

Rodeh93a.
Thomas L. Rodeheffer, "Experience with Autonet," *Computer Networks and ISDN Systems*, vol. 25 #6, pp. 623-629, Elsevier Science Publishers, 1993.

Rosin93a.
Matt Rosing, "Communications on Distributed Memory Multicomputers," *ICASE Research Quarterly*, vol. 2, no. 1, p. 4,5, March 1993.

Rosin94a.
Matt Rosing, *Personal Conversation*, Battelle Pacific Northwest Labs, March 1994.

Saave93a.
Rafael H. Saavedra, R. Stockton Gaines, and Michael J. Carleton, "Micro Benchmark Analysis of the KSR1," *Proceedings Supercomputing '93 Conference*, pp. 202-213, ACM/IEEE, Portland, OR, November 1993.

Smith94a.
Larry Smith and Others, "Optical Cabinet Links Being Studied," *Electronic Engineering Times*, CMP Publications, Manhasset, NY, February 28, 1994.

Steen94a.
Peter A. Steenkiste, "A Systematic Approach to Host Interface Design for High-Speed Networks," *Computer*, pp. 47-68, IEEE Computer Society, IEEE, March 1994.

Stumm90a.
Michael Stumm and Songnian Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer*, pp. 54-64, IEEE Computer Society, May 1990.

Tanen92a.
Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal, "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, pp. 10-19, August 1992.

Tennea.
University of Tennessee and NSF, *Draft for a Standard Message-Passing Interface*.

91a. Thinking Machines Corporation, *CM5: Technical Summary*, Cambridge, MA, October 1991.

Wilso93a.

Andrew Wilson, Jr., Richard P. LaRowe, Jr., and Marc J. Teller, "Hardware Assist for Distributed Shared Memory," *Proceedings 13th Annual International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Pittsburgh, PA, May 1993.

Xu90a.

Jian Xu and Kai Hwang, "Dynamic Load Balancing for Parallel Program Execution on a Message-Passing Multicomputer," *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pp. 402-406, 1990.

Yeung92a.

Donald Yeung and Anant Agarwal, "Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient," MIT/LCS/TM-479, Laboratory for Computer Science/Massachusetts Institute of Technology, October 1992.

Zorpe92a.

Glenn Zorpette, *The Power of Parallelism*, IEEE Spectrum, pp. 28-33, September, 1992.