To the Graduate Council:

I am submitting herewith a thesis written by Steven L. Kilgore entitled "A Distributed Hypercube Simulator on UNIX Workstations in a TCP/IP Based Network". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Tom Dunigan
Dr. Tom Dunigan
Major Professor

We have read this thesis
and recommend its acceptance:

D J Straight

J Moore

Jeffrey D Case

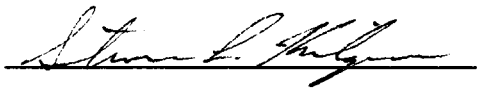
Accepted for the Council:

Carolyn Winkel
Vice Provost and
Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date 12/1/89

A DISTRIBUTED HYPERCUBE SIMULATOR ON UNIX WORKSTATIONS IN A TCP/IP BASED NETWORK

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Steven L. Kilgore
December 1989

ACKNOWLEDGMENTS

I would like to thank the following people and organizations, without whom this work would not have been possible.

International Business Machines, Inc. for providing the equipment on which this research was conducted.

Proteon, Inc. for the loan of the token ring wiring centers for use in this project.

The members of my thesis committee, T. H. Dunigan, J. D. Case, J. H. Poore, and D. W. Straight for their help and encouragement.

Vance Bass of the Knoxville IBM office for assistance with technical questions throughout the course of this research.

The staff of the University of Tennessee Computer Science Laboratories and the UT Computing Center, especially Chris Jepeway, Ken Key, Keith Moore, and Sam Nicholson, for technical assistance and encouragement.

Jim Summers of the CS Lab staff for repeatedly reinstalling AIX as needed.

Arthur C. Guinness for inspiration.

ABSTRACT

This paper describes the development and implementation of a simulator for the hypercube model of distributed memory parallel processors on a TCP/IP based network of UNIX[1] workstations. One of the primary advantages of this simulator is that multiple processors do work in parallel and actual real-time speedups are therefore possible. Benchmarks are provided for various network and operating system configurations along with an analysis of the performance and a comparison with the performance of the same programs on an actual hypercube, the Intel iPSC2.

1 UNIX is a trademark of AT&T Bell Laboratories

TABLE OF CONTENTS

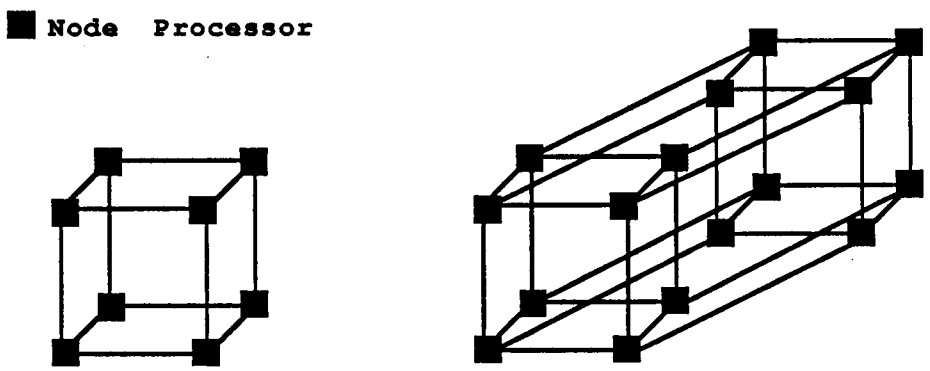
LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION TO THE PROBLEM

1.1 The Hypercube Model of Parallel Computation

A hypercube is a distributed memory parallel processor in which the individual processing units are interconnected by communication channels in a cube-like configuration. In an n-dimensional hypercube each node has a direct link to its n nearest neighbors. Thus in a two-dimensional hypercube the processors are logically connected in a square, in three dimensions the logical configuration is a geometric cube, and in four or more dimensions a "hypercube". In addition to the nodes which comprise the hypercube itself there is also a front-end or host processor which serves as the user interface and controls the other processors.



3-dimensional Hypercube **4-Dimensional Hypercube**

A typical hypercube application consists of at least two distinct programs, one of which is executed on the front-end host while the others run on one or more of the node processors. The host program handles such tasks as distribution of needed data to the nodes, collection and recombination of results, and I/O. The node programs are often, though not necessarily, identical copies of the same

code which are applied by the individual processors to different data.

Information is transferred among the various processors through message passing which must be explicitly coded as part of the application. Such communication is accomplished by means of a library of routines which are provided as extensions to standard programming languages such as C or FORTRAN. Other routines may also be available for use in explicit process synchronization.

1.2 Motivation for Simulation

One obvious reason for the development of a hypercube simulator is the limited accessibility of actual hypercube machines which are still relatively expensive and therefore not readily available to many potential users.

Even in an environment where a physical hypercube is available it is often much more efficient to use a simulator for code development. This is especially true since, in most cases, the hypercube architecture does not lend itself well to timesharing. While some hypercubes do allow partitioning of the available node processors into subcubes where different applications may execute concurrently, many only support a single user at a time. Thus the use of a simulator for development reduces contention for access to the real hypercube.

In addition, there are several points in favor of using a simulator over a real hypercube. For instance, the hypercube architecture tends to be highly specialized whereas the independent workstations which comprise the simulator are fully functional general purpose computing systems. Hardware implementations of hypercubes are also still relatively new and rapidly evolving. Thus the significant investment

required to purchase a physical hypercube may become obsolete long before a comparable investment in general purpose workstations and simulation software.

Another point in favor of simulation is the availability of equipment on which to conduct the simulation. The vast majority of currently available computing resources are still in the form of single processor machines, many of which would be of potential use in hypercube simulation. Perhaps the most widespread example of this class of machine is the engineering workstation, a great number of which are now available in many environments. Considered as a group, the workstations represent a vast amount of potential computing power, much of which is lost as individual machines often sit idle.

In most such cases where a significant number of workstations are present, they are linked by a high speed network which makes it possible to transfer large quantities of data efficiently among the various machines. Such a network can be used to provide the function of the internal communication channels of a hypercube thus making it possible to use several workstations to simulate the nodes of a hypercube.

Whereas simulation on a uniprocessor is useful for code development and programmer training, the single central processing unit must still do all the work and no real advantages are obtained in terms of execution speed. However, when several of these uniprocessor systems are used to simulate a hypercube all the other advantages of simulation are retained and true parallel processing may now be achieved.

Clearly, the use of a group of networked workstations to simulate a single parallel processing system is of great potential advantage in terms of utilization of the individual

machines, in the usefulness of the simulator, and in the real-time speedups obtained by truly doing the work in parallel.

2 STATEMENT OF THE PROBLEM

The purpose of the research is to develop software which will employ a collection of networked UNIX workstations to simulate a hypercube parallel processor, and to investigate the performance of various configurations of this simulator in comparison with each other and with an actual hypercube, the Intel iPSC2.

3 RESEARCH QUESTIONS

1. What sort of speedups are obtained by the simulator on computationally intensive applications? How does this compare with the performance of a real hypercube?

2. What effect does communications overhead have on the performance of the simulator? How does the performance of the simulator compare with that of a real hypercube on applications which involve varying amounts of inter-node communication?

3. What effect do the underlying network technology and configuration have on the performance of the simulator?

4. What effect does the underlying operating system (IBM ACIS/4.3 and AIX) have on the performance of the simulator?

5. What aspects of each operating system (IBM ACIS/4.3 and AIX) facilitated and/or hampered development of the simulator on that platform?

4 IMPLEMENTATION

4.1 The Distributed Hypercube Simulator (*dcube*)

The *dcube* distributed hypercube simulator is based on previous hypercube simulators developed by T. H. Dunigan of the Oak Ridge National Laboratory, Oak Ridge, Tennessee. The development and testing of this simulator have been supported by a research grant to the University of Tennessee Computer Science Department by IBM.

4.2 Development Environment

The majority of the available workstations today use some form of UNIX-based operating system. It is also comparatively easy to port code developed for one dialect of UNIX to any of a wide variety of related systems. In addition, many true hypercubes have employed some type of UNIX or a UNIX-like operating system for their front-end host processors. A large amount of previous work in hypercube simulation has also been done on some form of UNIX platform. Thus some form of UNIX operating system would present an attractive platform on which to develop the distributed hypercube simulator.

Support for the TCP/IP suite of network protocols is generally either included as part of most UNIX-related operating systems or is available as an option. In addition, TCP/IP is the primary communications protocol and de facto standard employed in most major networks such as the Internet. Thus TCP/IP was chosen as the network protocol on which to implement the initial versions of the distributed hypercube simulator.

4.3 Hardware Configuration

The work described in this paper was conducted on 18 RT PCs provided for this project by IBM. The machines are model 115 RT systems, each with 4 Megabytes (Mb) of memory. Each machine also had a single internal 70 Mb hard disk. Ethernet and token ring adapters were also provided by IBM and token ring wiring centers were provided by Proteon, Inc.

Installation of each operating system left very little room on the hard disks to be used for development and storage of source code and data files. In the later stages of the project it became necessary to remove the UNIX manual pages from the primary development machine in order to obtain more disk space. Results of benchmarks had to be stored on one of the Computer Science Department's fileserver systems due to lack of sufficient space on any of the RTs.

4.4 Software Configuration

The operating system software for the project was also donated by IBM and included versions of both AIX and ACIS/4.3, later renamed AOS.

Severely out of date versions of AIX were originally provided. The first release of AIX shipped was version 1.01 dated 1985. Release 2.1 was eventually obtained, however, TCP/IP support under AIX only reached necessary levels under version 2.2, which was received several months after the research had begun and under which the benchmarks described herein were conducted. However, it has still not been possible to obtain a suitable user interface such as X-windows for AIX and therefore 4.3 continues to be used for development and monitoring of experiments.

In addition, it was not possible during the course of the experiments to obtain the latest release of 4.3 for the RT. The most recent implementation of 4.3/RT which was provided is now two versions behind the latest release and does not include NFS (Network File System) support. Recently published benchmarks from the IBM/USC Advanced Computing Support Center in Los Angeles (ACSC, 1988) indicate that the latest release of 4.3, now known as AOS Release 2.0, has been tuned extensively and performs much better than the release on which the benchmarks documented herein have necessarily been based.

4.5 User's Guide

The *dcube* simulator is designed to run with one workstation as the front-end processor and other workstations as the nodes of the hypercube. It is possible to have a single workstation act as multiple nodes in the simulated hypercube or to run node processes on the host workstation. Of course, whenever multiple processes of the simulator are run on a single workstation some degree of parallelism is sacrificed.

Working versions of the simulator have been developed and tested for the IBM PC/RT under both IBM/ACIS 4.3 (Berkeley UNIX 4.3 for the RT) and AIX (IBM's System V based UNIX operating system for the RT).

The simulator consists of a user interface and driver program, which handles such tasks as establishment of machine-to-machine communications, and a library of extensions to both C and FORTRAN which are incorporated by the user into their application program. The programs are compiled and linked with the simulator libraries through the use of the *dbld* shell script.

In order to run an application a user first invokes the main program, *dcube*, on the front-end host machine and then specifies the dimension of the desired simulated hypercube, the application to run, and the machine(s) on which to run it. The user interface provides a series of simple commands which are used to provide this information. The commands can also be placed in a file which may then be used as input to the simulator.

The *dcube* commands are modeled after those used to control an Intel hypercube from the host processor and the syntax used by both the Intel simulator (Intel, 1986) and previous simulators developed by Dunigan (Dunigan, 1987). Available commands are

c [logfile]
h file
l [-c dim] [-h host] [-n node] [-p pid] [file]
s
t [on|off]
u file
q

The *u* command may be used to specify an input file from which subsequent commands will be read in.

The *c* command is used to enable logging of the simulation to the specified file. The default filename is SIMLOG.

The *t* command is used to enable or disable tracing.

The *h* command is used to specify the filename of the host side of the application. At least one *h* command is required for a successful simulation and if more than one is issued the last filename specified will be used.

The *l* command can be used to establish the dimension of the simulated hypercube with the *-c* option or to specify the host name of a remote machine to be part of the simulation, which node in the hypercube that machine is to become, and the filename of the node application program. Clearly, different node applications can be specified to run on different remote hosts if the application requires such capabilities. It is also possible to run multiple node processes on a single remote machine simply by using the *l* command with the same hostname and different node numbers.

The *s* command is used to start the simulation run.

The *q* command is used to quit and exit the dcube simulator.

For further information on the syntax of the library functions and the use of the simulator itself refer to the manual page for the simulator in Appendix A.

4.6 Underlying Mechanisms

When the simulator is instructed to begin execution of the user's application a number of steps actually take place. The main program first uses a *popen()* call to execute a remote command shell (through *rsh* on 4.3 or *remsh* on AIX machines) on each of the remote node machines to run the program *rmtdcube*. This remote side of the simulator then attempts to open a TCP/IP communication channel back to the front-end machine using *connect()*. Once the connection has been established the remote process will then use *exec()* to begin execution of the remote portion of the application and *dcube* will *exec()* the host side application program. In this way the communication channels which have been opened between *dcube* and *rmtdcube* are inherited as simple channel descriptors by the different application programs. If one or

more of the node processes are actually executed on the same machine as the host process, rmtdcube is activated using *fork()* instead of *popen()* as above and communications are established as before. In either case the nature of the connection is transparent to the user and their application.

Obviously, this implementation requires that the user who runs *dcube* on the host system must have permission to execute a remote command shell on each machine on which a node process is to execute. Such permissions may be established using the standard UNIX *.rhosts* file in the user's login directory or by the system administrator in the file */etc/hosts.equiv* on each remote machine. An executable copy of *rmtdcube* must also exist on each remote machine in a location which is included in the user's path, and, in the absence of some sort of network file system a copy of the portion of the user's application program which is to be executed on the remote system must reside in the user's login directory on each machine. A network filesystem would remove the need to copy the node application program to each system since each machine could then load and execute the program from a single location on one machine.

On each of the remote machines which is a part of the simulation only one TCP/IP connection is established per node process and that connection goes to the front-end machine. All messages, including those from one node process to another, are passed via the front-end host. Since a single machine is therefore exposed to every message that is passed, a thorough trace of the execution of the entire application can be provided on the front-end host. The availability of such a logging capacity is of great potential benefit in the process of debugging a hypercube application.

On the other hand, since each message must be processed and acted upon by the host, there is potential for a

bottleneck. This implementation also requires that a large number of processes run on the host machine when the simulator is initialized and that the host machine be capable of handling a high number of simultaneously open I/O channels. Both of these factors may place a limit on the size of the hypercubes which can be simulated readily, especially since each time the dimension of the hypercube is increased by one the number of required subprocesses and open channels on the host doubles.

5 RESULTS

A series of benchmarks were applied to the simulator in order to test both computation and communication performance in a number of different configurations. Various combinations of the 4.3 and AIX operating systems over both Ethernet and token ring networks were tested and the results are summarized in the following sections. In all benchmarks described as "mixed operating systems" an equal number of AIX and 4.3 machines were used in the tests. In all cases described as "mixed network technology" half of the node machines were connected by an Ethernet, half of the nodes were joined by a token ring, and the two networks were bridged by an additional RT which acted as the front-end host. Thus, since all messages from node to node must already pass through the host, no additional hops were required to cross the bridge from a node machine on the Ethernet to one on the token ring.

Results summarized below in each case indicate an average of at least five benchmark runs under the specified circumstances. 0-dimensional benchmarks indicate tests run with a front-end host and a single remote node machine.

The segment of Ethernet which connected the nodes of the simulator during benchmarks was isolated from the rest of the campus network by a LANbridge. During the process of all Ethernet-based benchmarks an additional program was run on a system on the other side of the bridge which periodically polled the bridge and recorded reported traffic seen on each side of the bridge. These values were then used to determine the level of network traffic generated by the hypercube simulator during the benchmark runs. No equivalent facility was available to the investigation to analyze packet traffic on the token ring network.

5.1 Simulator Performance On A Computationally Intensive Task

The first performance benchmark applied to the distributed hypercube simulator approximates the value of *pi* based on inscription of a specified number of rectangles in a unit circle. The host program serves as user interface to inquire the number of rectangles to be used, and transmits that information to each of the node processes. Computations are then carried out for an equal number of rectangles by each of the node processors in the cube. Results are returned to the host process which combines the results from the individual nodes and reports results and timing information to the user. This code requires a significant amount of computation and a relatively low amount of communication. For a complete listing of the source code used for this benchmark, please refer to Appendix B.

The results obtained by various configurations of the simulator and by the Intel iPSC2 are summarized in Tables 1 through 4. The number of rectangles specified in each case is 1 million and values reported are in milliseconds.

Table 1. Rectangle Rule Benchmarks on the Intel iPSC2 Hypercube

Cube Dimension	Milliseconds
0-dimensional	42746.0
1-dimensional	21368.0
2-dimensional	10695.0
3-dimensional	5344.0

Table 2. Rectangle Rule Benchmarks of *dcube* With Operating System IBM ACIS/4.3

	Ethernet	Token Ring	Mixed
0-dimensional	50298.2	50196.4	N/A
1-dimensional	25258.4	25203.8	25270.4
2-dimensional	12731.3	12639.0	12694.1
3-dimensional	6500.3	6485.6	6528.4

Table 3. Rectangle Rule Benchmarks of *dcube* With Operating System AIX

	Ethernet	Token Ring	Mixed
0-dimensional	43200.0	43600.0	N/A
1-dimensional	22000.0	21600.0	21600.0
2-dimensional	11100.0	10600.0	11200.0
3-dimensional	5400.0	5600.0	5600.0

Table 4. Rectangle Rule Benchmarks of *dcube* With Mixed Operating Systems

	Ethernet	Token Ring	Mixed
1-dimensional	25162.8		
2-dimensional	12669.4	Not Possible	
3-dimensional	6452.4		

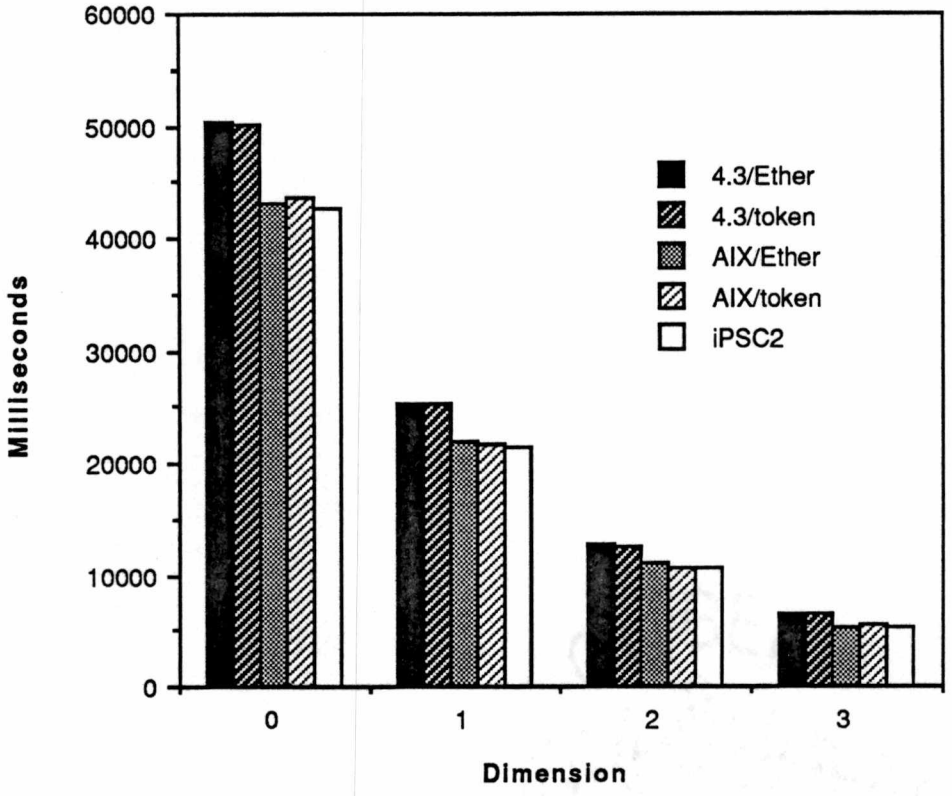


Figure I. Results of Rectangle Rule Benchmarks

As the summary of this data in Figure I clearly indicates, the distributed simulator performs extremely well in such a computationally intensive task, in its fastest configurations actually rivalling the performance of the iPSC2. From the data presented it is also possible to determine to what extent the parallel capacity of the system in question has actually been utilized, by calculating the parallel speedup obtained by benchmarks using successively larger numbers of processors.

In parallel computing the term "speedup" is generally defined such that "The speedup achieved by a parallel algorithm running on *p* processors is the ratio between the

time taken by that parallel computer executing the fastest serial algorithm and the time taken by the same parallel computer executing the parallel algorithm using *p* processors." (Quinn, 1987).

Alternately, we may define the speedup to be "the time taken by the parallel algorithm on one processor divided by the time taken by the parallel algorithm using *p* processors." Such a definition tends to exaggerate the total speedup obtained by the algorithm since it hides any overhead incurred by operations required to utilize parallelism (Quinn, 1987). A good example of such additional overhead is the extra work required to initially distribute data to the node processors, and to recombine their results when the algorithm completes. Nevertheless, since the primary interest is in comparing the performance of two parallel systems, the *dcube* simulator and the Intel iPSC2, running the same algorithm, such a measurement is meaningful in this case. Thus the speedup data displayed in Figure II are based on this second definition.

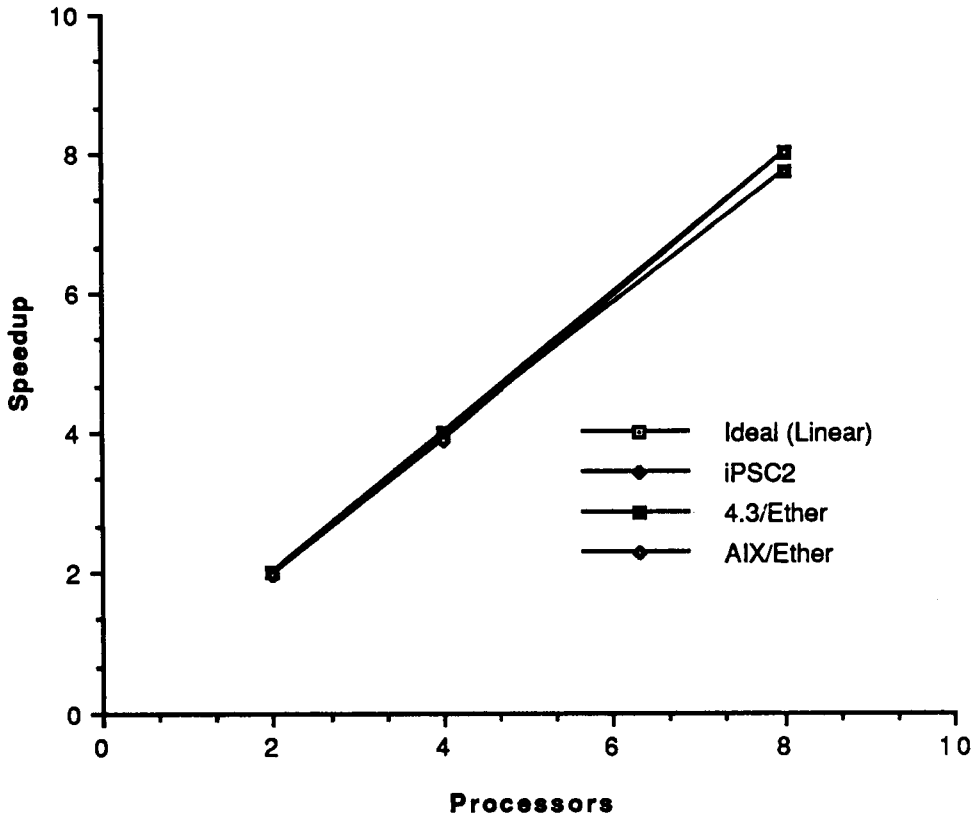


Figure II. Speedups Obtained in Rectangle Rule Benchmarks

The speedups displayed in Figure II approach very closely the ideal of linear speedup, in which doubling the number of processors devoted to accomplishing a given task will result in completion of the task in half as much time. Linear speedups are the goal to which parallel algorithms and processors generally aspire, thus we cannot expect significantly better performance than that displayed by the simulator in this set of benchmarks.

5.2 Effect of Communications Overhead

The second benchmark applies the Cholesky method to factorize a square matrix of specified dimension. The user is also allowed to specify whether the matrix should be considered to wrap around or reflect back at the edges, the blocksize into which the matrix is to be divided for the parallel factorization to take place, whether natural or gray code ordering of nodes should be used, and the method of communication to be used by the simulated hypercube. Valid choices for the method of communication are the standard hypercube bcube method with a minimum spanning tree, broadcast, or embedded ring communications. All of these choices are gathered by the host process and distributed to the nodes. Computations are then carried out at the nodes and results are reported back to the host process where they are recombined and reported to the user. A significant amount of communication among the various nodes is required in the process of the factorization and thus this benchmark is relatively intensive both in terms of computation and communication. A complete listing of the source code for this benchmark is included in Appendix B.

In each of the benchmark runs that produced the following results (Tables 5, 6, 7, and 8) the specified matrix dimension was 100, and wrap mapping, gray code ordering, maximum blocksize, and bcube communications were selected. All results are in milliseconds.

Table 5. Cholesky Matrix Factorization Benchmarks on the Intel iPSC2 Hypercube.

Cube Dimension	Milliseconds
0-dimensional	3080.0
1-dimensional	2350.0
2-dimensional	1800.0
3-dimensional	1390.0

Table 6. Cholesky Matrix Factorization Benchmarks of *dcube* With Operating System IBM ACIS/4.3.

	Ethernet	Token Ring	Mixed
0-dimensional	56178.0	56175.4	N/A
1-dimensional	55490.2	55553.4	55496.6
2-dimensional	54855.2	55168.2	55090.8
3-dimensional	55315.2	57019.8	55466.6

Table 7. Cholesky Matrix Factorization Benchmarks of *dcube* With Operating System AIX.

	Ethernet	Token Ring	Mixed
0-dimensional	60000.0	59800.0	N/A
1-dimensional	59000.0	58800.0	59000.0
2-dimensional	58400.0	58200.0	58400.0
3-dimensional	59000.0	58800.0	58400.0

Table 8. Cholesky Matrix Factorization Benchmarks of *dcube* With Mixed Operating Systems.

	Ethernet	Token Ring	Mixed
1-dimensional	55461.0		
2-dimensional	54904.4	Not Possible	
3-dimensional	55346.0		

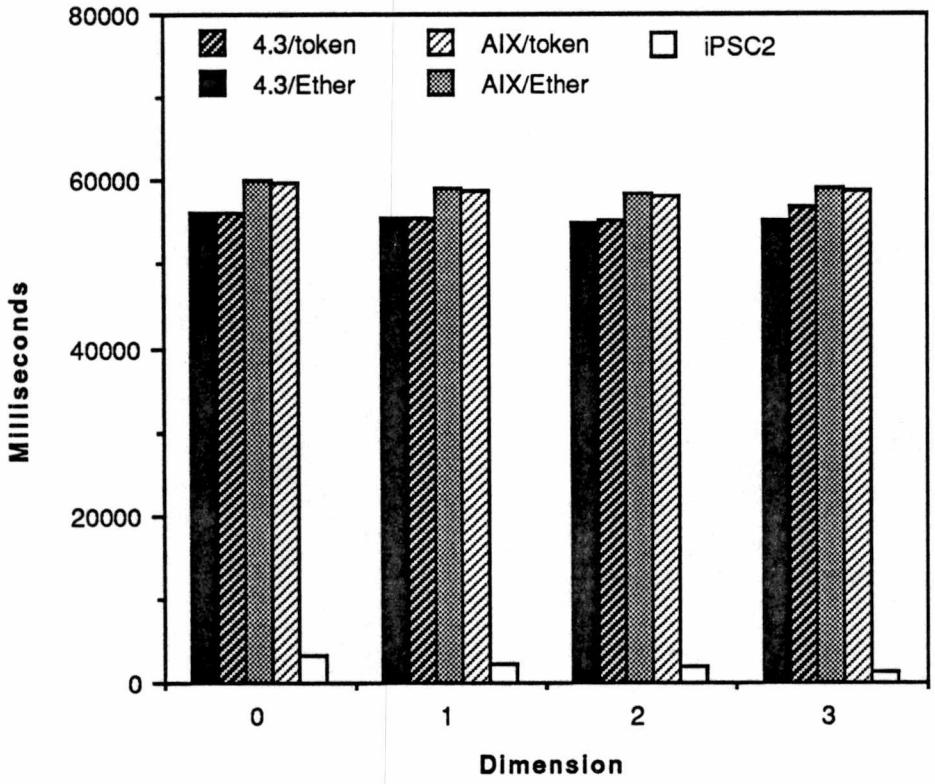


Figure III. Results of Cholesky Matrix Factorization Benchmarks

The data summarized in Figure III clearly indicate the considerable detrimental impact of communications overhead on the performance of the simulator. The Cholesky matrix factorization benchmarks provide a combination of high levels of both computation and communication. For a comparison of the volume of network traffic generated by each of the individual benchmarks running over an isolated segment of Ethernet, please refer to Appendix C.

In addition to sheer volume of data exchanged, a great deal of the communication involved in the Cholesky benchmarks

is node-to-node communication, which generates the most network traffic in the simulation due to the transmission of the message from node to node by way of the front-end host. Since each message from one node to another must actually take the form of two real messages in the simulator, we would expect the iPSC2 to perform somewhat better in any scenario which involves intense node-to-node communication. We must also recall, however, that this additional step of passing the message through the host processor also makes possible a valuable debugging facility.

In addition, the discrepancy in the performance displayed by the various configurations of the simulator and that obtained by the iPSC2 is too great to attribute entirely to the additional routing hop required in the simulator. If communications were equally expensive for the simulator and the real hypercube, and the simulator had to pass twice as many actual messages, it would be reasonable to expect the real hypercube to perform roughly twice as fast. Clearly the difference is much greater than a factor of two. Further communication-oriented benchmarks support the conclusion that communications overhead, even for a single message, is much greater through TCP/IP over either underlying network technology in the simulator than over a dedicated channel in the iPSC2 (pages 28-29).

The results obtained also indicate that the distributed simulator is not nearly as successful in achieving parallel speedup with such a mixture of communication and computation as with computation alone. In Figure IV we can see that even when run on the real hypercube this application does not yield speedups that approach the nearly ideal results obtained from the previous set of benchmarks. In comparison both with the ideal and with the performance of the iPSC2, the speedups of various configurations of the simulator vary

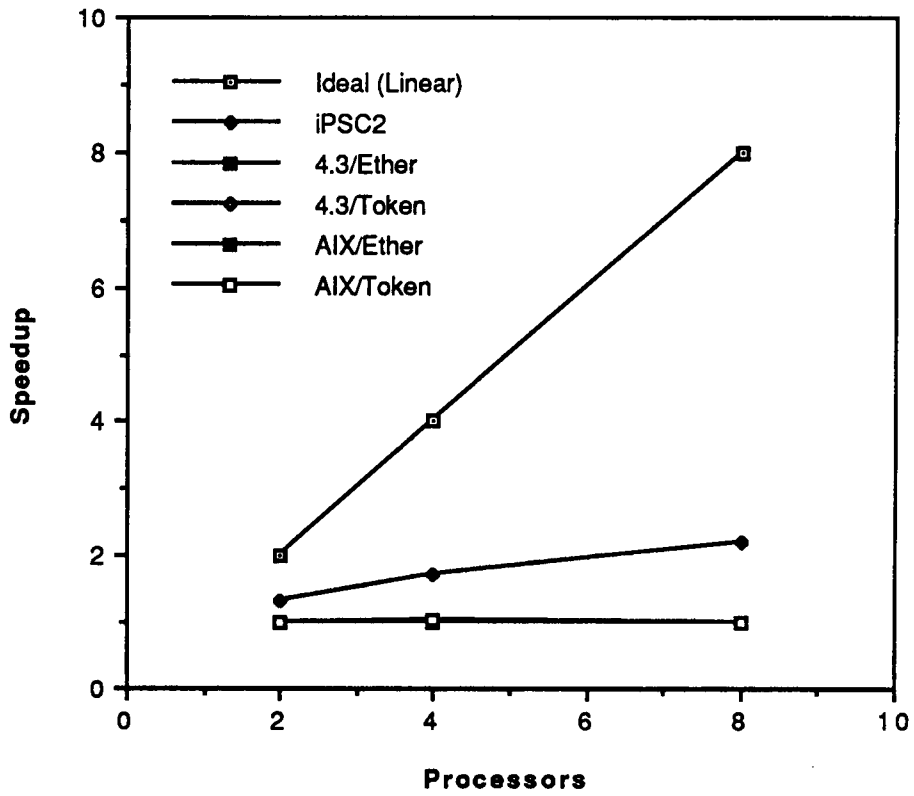


Figure IV. Speedups Obtained in Cholesky Matrix Factorization Benchmarks

so little among themselves as to appear virtually indistinguishable from each other in this graph.

A closer examination of the lower portion of Figure IV provides a better indication of the actual speedups obtained by the various configurations of the simulator. Such a close-up view is provided in Figure V.

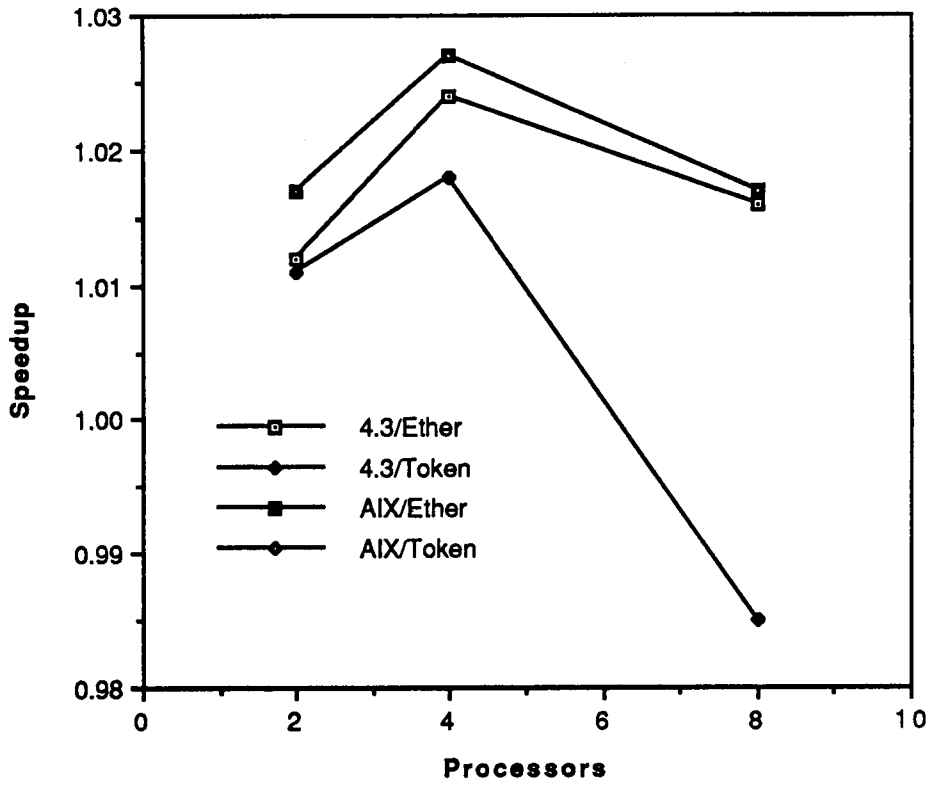


Figure V. Speedups Obtained in Cholesky Matrix Factorization Benchmarks (Simulator Only)

Very little speedup is achieved as the number of processors is increased and eventually a point is reached past which the addition of more processors actually slows down the completion of the simulation. This phenomenon appears to be due to the amount of expensive node-to-node communication overhead that is added by the addition of more nodes. As more processors are added the size of each processor's portion of the matrix is reduced thus computation to communication ratio decreases and performance begins to degrade once the ratio becomes sufficiently small. In the

results above this behavior can be observed when the dimension of the simulated cube is raised from two to three.

5.3 Effect of Underlying Network Technology

The direct data transfer rate of both network technologies was measured using *ttcp* on the actual networks on which all benchmarks were performed. A random data pattern was transmitted across the network to a specific destination and the time required to complete the transmission was used to calculate the data transfer rate for the network in question. The token ring data transfer rate was measured under the 4.3 operating system at 134.24 Kb/sec as compared to the 4.3/Ethernet rate of 121.38 Kb/sec. The corresponding AIX data transfer rates were measured at 195.67 Kb/sec for token ring and 141.38 Kb/sec for Ethernet. Clearly the transfer rate is somewhat superior for token ring regardless of which operating system is in use.

For the most part these results are reflected in the various benchmarks of the configurations of the simulator. However, it is important to recall that the *ttcp* test measures point-to-point data transfer rate. In most actual applications on the hypercube simulator a given processor will need to communicate with more than a single other machine. Thus the actual simulator benchmarks provide a somewhat more relevant estimation of the true impact of the underlying network technology. The following sections discuss these effects on both of the previously discussed benchmarks as well as two additional benchmarks selected specifically to test network performance.

5.3.1 RECTANGLE RULE BENCHMARK

The underlying network technology of the different simulator configurations seemed to have relatively little impact on the results obtained from the rectangle rule benchmarks. This is not surprising in such a computationally intensive benchmark since the amount of communication, and thus the level of network traffic, are too small to produce a significant impact on the overall results. Please refer to Appendix C for the actual measurements of network activity observed during the benchmarks.

5.3.2 CHOLESKY BENCHMARK

Underlying network technology appears to make a slight difference in performance of the different Cholesky simulations. In simulations run under 4.3 those conducted over Ethernet performed slightly better than those which ran over token ring, in apparent contradiction to the indications of point-to-point data transfer rate as measured with *ttcp*. The opposite situation was true with the AIX benchmarks; token ring was slightly faster.

It is also the case that in the 4.3 benchmarks as the number of nodes increases so does the performance advantage of 4.3 running over the Ethernet as compared to 4.3 running over token ring; a situation which is also reflected by the results of the following communication intensive benchmarks.

5.3.3 RING MESSAGE PASSING BENCHMARK

The third benchmark was selected primarily to test the communications performance of the simulator in different network configurations. A message of user-specified size is

passed from node 0 around a ring embedded in the simulated hypercube for a specified number of loops and timing information is reported back to the front-end host. The host program prompts the user for size of message to pass, number of nodes to pass it through, and number of loops to perform. It then sends that information to node 0 which controls the ring and reports back when all iterations have been completed. All results are reported in milliseconds (Tables 9-12).

Table 9. Ring Message Passing Benchmarks on the Intel iPSC2 Hypercube.

Cube Dimension	Milliseconds
0-dimensional	134.5
1-dimensional	432.0
2-dimensional	864.5
3-dimensional	1728.33

Table 10. Ring Message Passing Benchmarks of *dcube* With Operating System IBM ACIS/4.3.

	Ethernet	Token Ring	Mixed
0-dimensional	12801.5	30316.2	N/A
1-dimensional	21901.4	57191.8	33439.4
2-dimensional	43796.2	105577.8	68475.2
3-dimensional	87635.2	216428.6	142179.2

Table 11. Ring Message Passing Benchmarks of *dcube* With Operating System AIX.

	Ethernet	Token Ring	Mixed
0-dimensional	10000.0	8600.0	N/A
1-dimensional	19600.0	17600.0	19400.0
2-dimensional	39400.0	33200.0	36600.0
3-dimensional	79600.0	70400.0	74600.0

Table 12. Ring Message Passing Benchmarks of *dcube* With Mixed Operating Systems.

	Ethernet	Token Ring	Mixed
1-dimensional	20139.4		
2-dimensional	40066.6	Not Possible	
3-dimensional	80220.4		

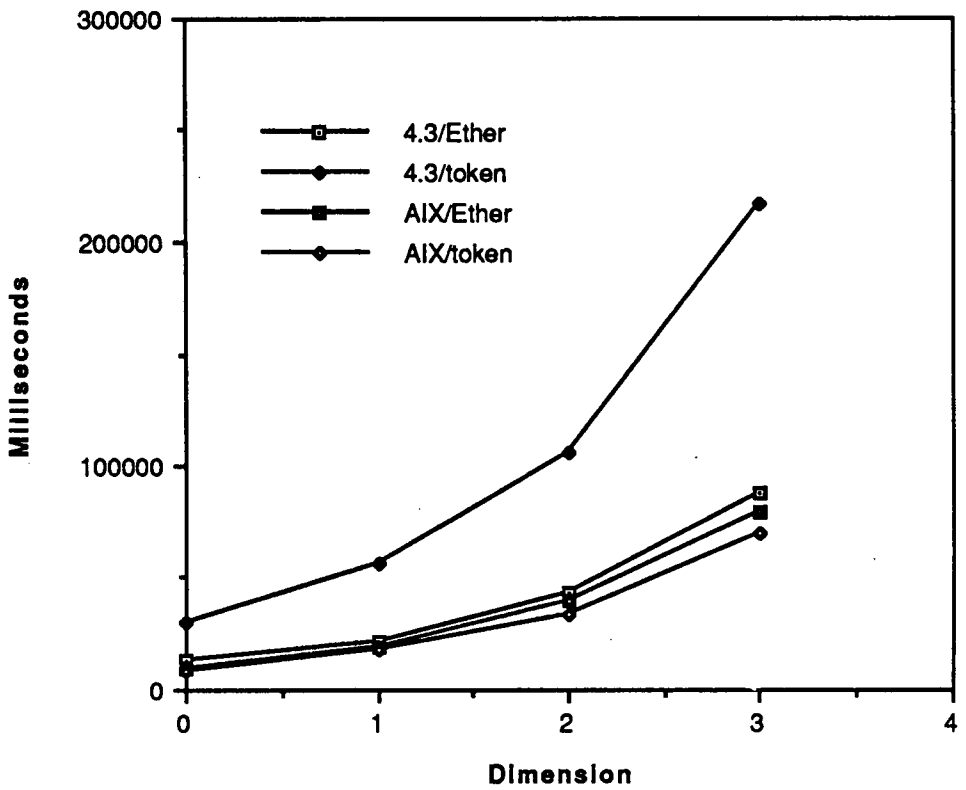


Figure VI. Results of Ring Message Passing Benchmarks

The ring message passing code (Figure VI) provides a communication intensive benchmark with an exceptionally small amount of computation involved. Furthermore, most of the communication is node-to-node and therefore generates the

highest possible amount of traffic in the simulator due to the fact that each message from one node to another must first pass through the host machine. Overall performance of the simulator in ring message passing surely suffers considerably as a result of this extra step in the path of each message.

Since this benchmark is very communication intensive the effects of underlying network technologies on the benchmark results is of particular interest and appears to have a highly significant impact, especially where 4.3 is concerned. In ring simulations conducted under AIX the benchmarks which were run over a token ring performed somewhat better than those which ran over Ethernet and mixtures of the two produced results between those yielded by the homogeneous configurations. The performance advantage exhibited by the token ring results under AIX can be attributed to the difference observed in token ring data transfer rate in the previously discussed *ttcp* measurements.

On the other hand, the performance of the token ring benchmarks under 4.3 was much slower than that of the same benchmarks run over Ethernet. Once again, as in the Cholesky benchmarks, the performance of 4.3 running over Ethernet and that of 4.3 running over token ring diverge as the number of nodes increases.

Perhaps the most likely explanation for this behavior lies in the fact that the early release of 4.3/RT under which the benchmarks were performed had in all probability not been tuned to work well with certain hardware, such as the token ring adapters in this case. The great majority of the messages generated in both the Ring and Cholesky benchmarks, and in the Echo benchmark discussed in the next section, are expensive node-to-node messages which must make the extra hop through the host machine. Thus a single system is required

to handle a high volume of network traffic routed to several different destinations, and the relatively poor performance of interaction between the operating system and network hardware is then exposed.

5.3.3.1 FORTRAN Ring Message Passing Code

This benchmark was performed only for the 4.3 configurations since 4.3 comes with a FORTRAN compiler, *f77*, whereas AIX does not. All parameters and details of the benchmark correspond to the preceding set of ring message passing benchmarks which were implemented in C (Table 13).

Table 13. FORTRAN Ring Message Passing Benchmarks of *dcube* with Operating System IBM ACIS/4.3

	Ethernet	Token Ring	Mixed
0-dimensional	13063.2	26178.0	N/A
1-dimensional	21892.8	44420.6	32090.6
2-dimensional	43754.8	107820.6	72566.6
3-dimensional	87589.2	216476.4	163247.2

All results obtained in the benchmarks of the FORTRAN versions of the ring message passing code agree very closely with the results obtained with the C version which are discussed in the preceding section. The difference in programming language seems to have only a negligible impact on the results.

5.3.4 ECHO MESSAGE BOUNCING BENCHMARK

The final benchmark performed was another test of communication performance wherein a message of user-specified size is sent from node 0 to each other node in the simulated hypercube and immediately echoed back. Node 0 records the time required for the round-trip of the message and reports

totals back to the front-end host. The host program prompts the user for size of message to pass and number of iterations to perform for averaging. It then sends that information to node 0 which conducts the echo tests and returns timing information back to the host. The host process then calculates the average amount of time required to bounce a message from node 0 to a node that is located n logical hops away from node 0 in the simulated hypercube, where n varies from 0 to the dimension of the simulated cube. These averages are then reported back to the user. Values returned are in seconds (Tables 14-17; Figure VII).

Table 14. Echo Message Bouncing Benchmarks on the Intel iPSC2 Hypercube.

Distance	Seconds
Hop of 0	0.00302
Hop of 1	0.00873
Hop of 2	0.00874
Hop of 3	0.00877

Table 15. Echo Message Bouncing Benchmarks of *dcube* with Operating System IBM ACIS/4.3.

	Ethernet	Token Ring	Mixed
Hop of 0	0.258988	0.528632	0.687992
Hop of 1	0.438312	0.910712	1.208208
Hop of 2	0.4386	0.947304	1.190628
Hop of 3	0.440052	1.031084	0.74276

Table 16. Echo Message Bouncing Benchmarks of *dcube* with Operating System AIX.

	Ethernet	Token Ring	Mixed
Hop of 0	0.192	0.2	0.188
Hop of 1	0.4	0.368	0.388
Hop of 2	0.396	0.376	0.32
Hop of 3	0.396	0.356	0.416

Table 17. Echo Message Bouncing Benchmarks of *dcube* with Mixed Operating Systems.

	Ethernet	Token Ring	Mixed
Hop of 0	0.259376		
Hop of 1	0.438164	Not Possible	
Hop of 2	0.439664		
Hop of 3	0.404564		

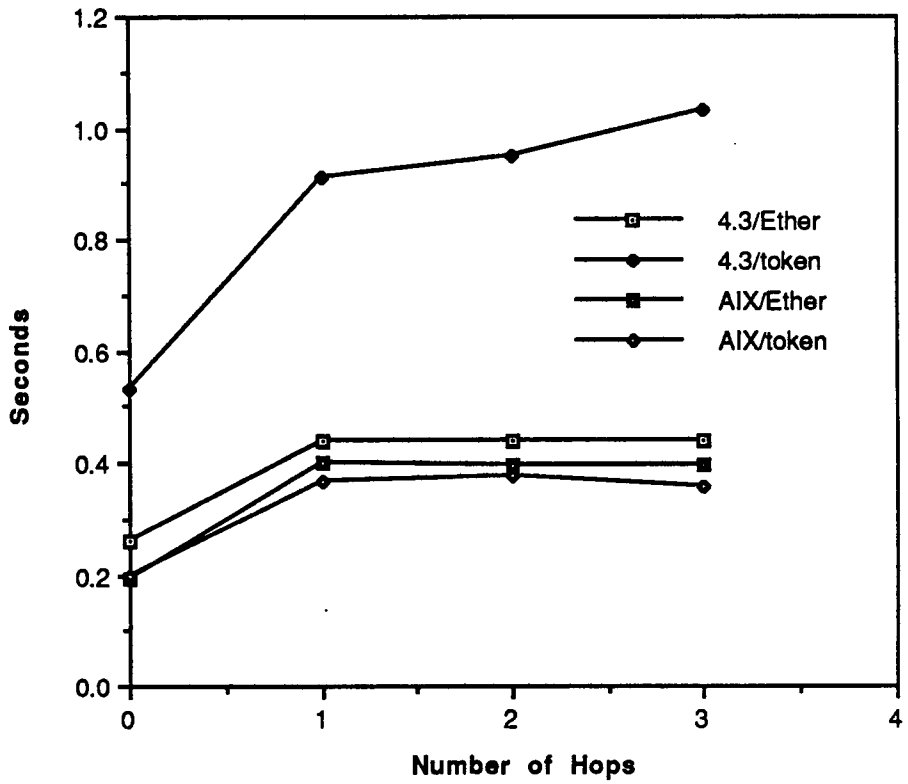


Figure VII. Results of Echo Message Bouncing Benchmarks

The final set of benchmarks provides a little more information on the amount of time required to send an individual message from one node to another through the simulated hypercube. Once again, all node-to-node messages are actually passed via the front-end host machine.

As was the case in the Ring and Cholesky benchmarks, the results indicate that the 4.3 operating system does not work comparatively well with a token ring network, performing only half as fast as even 4.3 over an Ethernet. As discussed in the previous section, this behavior is probably due to the fact that the release of 4.3 on which these benchmarks were conducted had not been well tuned to run on the hardware in comparison to AIX or to newer releases of 4.3.

It is important to remember that the table of results reflects a logical number of hops through the simulated cube and does not necessarily correspond to actual physical distance between machines. For instance, in the benchmarks of 4.3 running over a mixture of Ethernet and token ring, the nodes located 0 and 3 "hops" away were actually on the Ethernet side of the mixed network whereas the other two nodes were across the host bridge on the token ring.

A related aspect of the data as displayed in Figure VII is that the difference in time required to pass the message 1, 2, or 3 hops over the Ethernet differs only by a negligible amount. This is the result that would be expected since the Ethernet functions essentially as a shared bus on which the machines which are logically 1, 2, and 3 hops distant are effectively equidistant in the underlying network configuration.

5.4 Effect of Operating System on Simulator Performance

5.4.1 RECTANGLE RULE BENCHMARK

A comparison of the performance of the simulators running under the two operating systems indicates a clear performance advantage on the part of AIX. This can be easily explained since the AIX system has been more finely tuned to run on the RT hardware. AIX, though based on System V Unix, was developed internally at IBM; and the RT hardware, which was also developed by IBM, is one of the platforms on which it was expected to operate. Thus it comes as no surprise that the two work together relatively well.

On the other hand, the version of Berkeley 4.3 which runs on the RTs is ported software which was not originally designed to run on the hardware in question. In addition, an early release of the ported software was used in these benchmarks, as discussed in Section 4.4. It is therefore reasonable to assume that the operating system software in question may not have been well tuned to support the RISC processor in the PC RT.

Evidence to support this conclusion is also provided by recently published benchmarks of the latest release of IBM's implementation of 4.3 (ACSC, 1988). Significant improvements in computational efficiency of the newest release, AOS 2.0, over the release on which the simulator was benchmarked are clearly evident. It is probable that if it had been possible to obtain AOS version 2.0 in time to conduct the 4.3 benchmarks on that platform the additional tuning that has been performed on that system would have compensated for most, if not all, of the performance advantages displayed in these benchmarks by AIX.

Benchmarks for the same problem running on a simulator comprised of equal numbers of AIX and 4.3 machines in each case returned results between those of the pure AIX and 4.3 simulators but closer to the higher values returned by 4.3. This performance is expected since the entire simulation is not complete until the last node has finished its computations and reported its results back to the host. Thus a mixture of fast and slower machines would be expected to perform somewhere between the two pure cases, but the limiting factor on its performance would be the response time of the slower machines.

5.4.2 CHOLESKY BENCHMARK

A comparison of the performance under the different operating systems of this, the only benchmark to combine significant levels of concurrent computation and communication, reveals the only case in our set of tests in which 4.3 outperforms AIX.

A situation such as that produced by the Cholesky benchmarks, in which high levels of computation and communication are interleaved, would result in more frequent context switches by the operating system as both activities are carried on. It would become necessary to repeatedly perform context switches as the user process performing the computations was interrupted by the arrival of incoming packets from the network.

Low-level facilities such as interrupt handling are handled for AIX by a separate underlying program, the VRM or Virtual Resource Manager. The use of a separate program to handle these functions, which are handled directly by 4.3, inserts an additional layer between the operating system and

the hardware which almost certainly degrades the performance of the system.

Such degradation is not seen in any of the other benchmarks since only one or the other of the two types of activity, communication or computation, is present at a significant level. Thus it is not necessary to constantly switch contexts between the two and AIX, free to concentrate on a single task for which it is well tuned, outperforms 4.3.

5.4.3 COMMUNICATIONS BENCHMARKS

The comparison of results from the 4.3 and AIX simulator runs of these benchmarks once again indicate a definite performance advantage for AIX. This difference is again attributed to better tuning of the AIX system for execution on the RT hardware, especially in the case of the token ring adapter as discussed in Section 5.3.3.

6 OPERATING SYSTEM CONCERNS

6.1 Advantages and Disadvantages of ACIS/4.3

The availability of operating system source code was a major advantage of 4.3. The maximum number of processes on the front-end host machine had to be raised in order to successfully run 4-dimensional simulations. The maximum number of open files and maximum number of subprocesses per user also had to be raised. Each of these modifications required changes to operating system source code and subsequent recompilation of the kernel for the front-end host machine. Such modifications would not have been possible without source code, and simulations would have been limited to a maximum hypercube size of 3 dimensions, as was the case with AIX.

In addition, Berkeley Unix is a well established operating system with a large body of supporting documentation and expertise readily available from a variety of sources. It was thus possible to draw on experience gained from work with other Unix systems, as well as resources available with those systems, in the process of working with the RT implementation of 4.3.

A wide variety of development and system monitoring tools are also available for the 4.3 operating system, most of which were either already available or easily ported to the RT implementation. Notable examples of such programs which were used in the course of this research are *strings* and *ttcp*.

The availability of the online Unix *manual* pages under 4.3 was of invaluable assistance during the course of this work.

The ability of 4.3 to *autoconfig* new hardware additions also saved a great deal of trouble which was required to install the same options under AIX, as described in the following section.

6.2 Advantages and Disadvantages of AIX

The greatest advantage of AIX is clearly the superior performance it displays in the majority of the benchmarks of the simulator. As we have discussed previously, this is probably due to the extent to which it has been tuned to work efficiently with the RT hardware. However, other aspects of the system may outweigh this advantage, at least in the release of the software under which these tests were conducted.

The documentation provided with the AIX system was mostly of a very simplistic nature and in many cases failed to provide sufficient technical detail. No documentation whatsoever was provided for system calls and associated libraries and AIX has only very limited online help facilities. The overall approach of the documentation seems to attempt to make things easier for a completely naive user to understand. Technical documentation of the sort required by someone experienced with Unix systems is neglected in favor of this approach.

The lack of documentation on the system libraries for AIX also complicated the process of implementing the simulator on that operating system. During the linking of the simulator it became clear that there were additional library routines which would have to be explicitly linked with the simulator code. Since no documentation could be found to provide an indication of which libraries were needed it became necessary to use *egrep* to search the system library

files for the missing routines. This method revealed that the required libraries were */usr/lib/libbsd.a* and *libsock.a*.

While installation of a basic AIX system was fairly easy and straightforward, customization of the system was considerably more difficult. For instance, the addition of an RT baseband adapter to connect the machine to the Ethernet was particularly complicated. Once the board had been physically installed it was necessary to run the *devices* command to add the device to the system configuration. This addition could not be accomplished with a simple command but instead required traversal of several layers of menus. Once the correct menu for the baseband adapter was finally reached it was necessary to change four hexadecimal address values on the menu to the correct settings since the defaults did not match the settings of the hardware in any of the four cases.

The default interrupt level specified for the baseband adapter in *devices* was also incompatible with the default hardware settings on the card. In this case, however, there were only three possible values that the command would accept. The default setting of the interrupt control jumper on the baseband adapter as shipped was not one of the acceptable values, so it was necessary to remove the card, change the hardware jumper to one of the three settings that *devices* would accept, and reinstall the card.

On the other hand, the 4.3 *autoconfig* procedure polls interrupts and figures out proper address values at system startup and thus not only supports whatever value to which the jumper may be set but also figures out the proper interrupt level and address settings for itself.

Once the baseband adapter had been successfully configured into that system and the AIX machine brought up on the Ethernet it was still necessary to run *devices* again to add pseudo-terminal devices before it became possible to

login in to the machine over the network. Once again, these additions could not be accomplished with simple commands but required traversal of several menus. In addition, multiple devices had to be added individually even if the devices in question were identical.

The menu orientation of the *devices* command does make it somewhat easier to understand for a naive user as opposed to the comparatively cryptic syntax of the 4.3 *ifconfig* command. On the other hand, an experienced user can accomplish with a single *ifconfig* command a task which requires the traversal of numerous menus with *devices*, and it would hopefully be a comparatively rare case for a naive user to be modifying system configurations. In this case, AIX tries to be friendly and winds up getting in the way.

AIX also seems to be particularly sensitive to the status of its configuration files at boot time. If any of these files have been inadvertently stored in an inconsistent state when the system goes down, it is generally not able to successfully reboot. In several such cases it was necessary to boot the system in single user mode and manually perform an *fsck* of the */usr* filesystem before the system was able to boot.

One way in which such inconsistencies in system files may come about is through an apparent problem in the AIX installation procedure for system options. If the installation does not complete successfully for whatever reason, such as insufficient space available on the hard disk, the files may be left in an inconsistent state.

Another way such inconsistencies may come about is by direct editing of the files, which was by far the easiest way to accomplish a configuration change such as converting the system to operate over a token ring interface rather than an Ethernet baseband adapter. If the files were changed and the

system subsequently rebooted, the system would regularly come back up with an apparently random subset of the configuration changes which had been made. If the resulting combination of features were mutually inconsistent, the system might not successfully reboot at all. Single user mode repairs would then be necessary as described above. The cause of this behavior is the fact that AIX does not always *sync* its disks prior to rebooting, thus some changes had actually been written out to disk while others might not have been.

Such behavior is probably a feature left over from the System V origins of AIX. However, in a system such as this one which attempts to combine so many features of 4.3 and System V there is often no way of knowing which of the two it will behave like. Such features need to be clearly documented, and such is not the case under AIX version 2.2.

Even if all the system configuration files have been stored consistently and with the proper information, it is still possible to keep the system from successfully rebooting by making a simple hardware change. For instance, if the system attempts to configure a token ring interface but is not currently attached to a token ring network the system will hang and fail to complete its reboot. A much more reasonable approach would be to automatically unconfigure that device and bring the system up without it.

Another outstanding disadvantage of AIX is the presence of the Virtual Resource Manager, VRM, as discussed in Section 5.4.2. Generally, the primary objective of a virtual machine interface is the ability for different users to run different operating systems concurrently on the same real machine (Beck, 1985). On a workstation class machine such as the RT where the machine is usually primarily devoted to a single user, a virtual machine interface is probably not worth the additional overhead it incurs. In any event, since AIX is

currently the only publicly available RT operating system which runs over VRM, the possibility of running different operating systems concurrently does not exist. The probable reason for the presence of VRM is upward compatibility with versions of AIX for larger IBM systems where virtual machine interfaces are regularly employed.

6.3 Operating System Interoperability

Of the nine possible combinations of the two operating systems, the two underlying network technologies, and mixtures of each, hypercube simulations were successfully conducted on seven. It was not possible to get RTs running ACIS/4.3 and those running AIX to communicate with each other over a token ring network under the releases of the operating system software with which this research was conducted. Since this combination constitutes a subdivision of a configuration combining both operating systems with both underlying network technologies, it has not yet been possible to run simulations on this configuration either.

Packets directed from a 4.3 machine to an AIX machine using *ping* do appear to reach the AIX machine but are then discarded. This behavior has been observed using the *netstat -v* command on the AIX machine in question to obtain network information from the Virtual Resource Manager (VRM). Simulations running under the same combination of operating systems have been successfully tested over Ethernet, therefore the inconsistency which produces the above behavior must exist at a relatively low level, probably in the token ring drivers, and is probably a result of incompatibility in the old release of 4.3 under which the tests were conducted.

7 CONCLUSIONS

The *dcube* distributed hypercube simulator described herein can be a useful tool on which to develop hypercube applications and on which to learn the basic concepts of programming a hypercube. Unlike most other simulators and real hypercubes, it is capable of providing a complete trace of all messages passed anywhere in the cube for debugging purposes.

In addition, the simulator provides real parallel processing with nearly linear speedups evident for computationally intensive applications. In such cases the simulator compares very favorably with even a second generation true hypercube.

Efficiency of the simulator degrades when high volumes of communication are required, especially when there are significant volumes of computation taking place concurrently with the communication. Communication performance could be improved by routing node-to-node messages directly, but this capability would be acquired at the cost of providing a complete message tracing facility. Nevertheless, communication overhead clearly comprises the great majority of the cost encountered by the simulator. Although computational speed rivals that of a true hypercube communication speed is orders of magnitude slower. The great majority of the overhead thus incurred can be attributed to the hardware involved and the communications protocol, in this case TCP/IP. Communication speed might therefore be improved through the use of alternative network protocols other than TCP/IP.

It is also important to remember that all network-related benchmarks reported were performed on an isolated

network. In a typical real-world environment there will be additional traffic on the network with which the hypercube simulation will have to contend, and communication performance will degrade further as a result. Nevertheless, the simulator is still useful for development, debugging, and training.

The simulator is based on general purpose workstations which are useful for any number of individual applications aside from the hypercube simulator. In an environment where such workstations exist the simulator can also provide a useful means to utilize a vast number of potentially wasted computing cycles.

Impact of the underlying network technology varies greatly depending on such factors as the nature of the application, the topology of the underlying physical network, which operating system is being used, and so forth. The primary consideration which seems to influence performance of a given network technology and configuration is the nature of the application to be simulated.

Of the two operating systems for which the simulator was developed and tested, AIX performed better in either computation intensive or communication intensive applications. This difference appears to be due primarily to performance tuning which should be much less of a factor in more recent releases of ACIS/4.3, now known as AOS.

On the other hand, when both computation and communication are found concurrently in significant volume in the same application 4.3 actually outperforms AIX, though neither performs extremely well under such circumstances. A number of factors may contribute to these performance differences, including system tuning and basic differences in the structures and designs of the two operating systems.

Although AIX generally seemed to outperform the tested release of 4.3, it was much more difficult to get the software running on AIX in the first place. Ease of configuration, development environment, versatility, ease of use, and accessibility and usefulness of system services were all far superior on 4.3. In addition, the machines running 4.3 seemed to run far more reliably. In the course of this research project as many as seven AIX machines were unusable at one point or another, most with system problems such as corrupted hard disks directly traceable to AIX. Only one 4.3 machine was unexpectedly down during the entire course of the project and that was due to hardware problems unrelated to the operating system.

8 DIRECTIONS FOR FUTURE WORK

The results discussed in this paper clearly indicate several areas in which to continue related research in distributed hypercube simulation.

The obvious area in which to concentrate attempts to improve the efficiency of the simulator is that of communication among the various machines involved in the simulation.

Direct logical communication channels between node processors without involvement of the front-end host machine will remove a potential communications bottleneck and decrease the total volume of network traffic generated. In practice the advantages derived from such connections would depend to a great extent on the underlying network configuration. In the mixed network examples documented in this paper, for example, the additional hop would still be required in order to bridge the two networks thus the realized gain would be minimal. In addition, the availability of potentially valuable debugging information on the host machine would be sacrificed in order to use direct node-to-node communications.

A related improvement would be the use of true broadcast messages when available in the underlying network technology to distribute information which must be sent to all the systems which comprise the simulated hypercube. In the current setup such a message must be sent to each of the machines individually in a serial procession.

Another source of considerable communications overhead appears to be the use of TCP/IP. Exploration of alternative networking protocols such as UDP for machine-to-machine

communications might provide communications at a lower cost than is possible with TCP/IP.

A different area in which the simulator might be improved by further work is ease of use. In the current implementation the user is required to specify a large amount of information such as the machine names on which to run node processes. Although such information may be stored in a file which will then be used repeatedly as input to the simulator, such usage implies a certain loss of flexibility as the same set of machines are always used. It would be highly desirable at the time of any given simulation to use the set of machines which already had the lightest workload at that time. Such a capability would increase efficiency of the simulation, utilization of the individual machines, and, if automated, would make the simulator much easier to use.

Finally, extension of the simulator to support a heterogeneous set of machines would add considerable flexibility, but is a difficult problem to address since different architectures differ greatly in such aspects as data storage formats and byte ordering.

LIST OF REFERENCES

REFERENCES

(ACSC, 1988) USC Advanced Computing Support Center, *Infrequent Flyer*, November 2, 1988, p. 4.

(Beck, 1985) Leland L. Beck, *System Software, An Introduction to Systems Programming*, Addison-Wesley Publishing Co., Reading, Mass., 1985.

(Dunigan, 1987) T. H. Dunigan, *A Portable Hypercube Simulator*, Tech. Rept. ORNL/TM-10410, Oak Ridge National Laboratory, Oak Ridge, TN.

(Intel, 1986) Intel, *iPSC Hypercube Simulator*, Intel 310104-003, Portland, Oregon, October, 1986 cited in (Dunigan, 1987).

(Quinn, 1987) Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Co., New York, NY, 1987.

APPENDIXES

APPENDIX A
dcube Manual Page

NAME

dcube - distributed cube simulator

SYNOPSIS

dcube commands:

c [filename]	disable logging or log to *filename*
h [-h remote] file	load file as host on remote
l [-n nodeno] [-c dim] [-h remote] file	load *nodeno* node or all nodes of a cube of dimension *dim* with file on remote
q	quit
s	start simulation
t [on]	enable or disable trace
u file	use file for command input

The host and node programs should be linked with the library *libdcube.a* which contains the following subroutines:

```
int copen(int pid);
void cclose();

void send(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendw(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);
void sendmsg(int d, int type, char *msg, int msglth,
        int dstnode, int dstpid);

void recv(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvw(int d, int type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);
void recvmsg(int d, int *type, char *msg, int maxlth,
        int *msglth, int *srcnode, int *srcpid);

int probe(int d, type);
int status(int d);

int mynode();
int mypid();
int cubedim();
int clock();
void syslog(int pid, char *msg);
void flick();
```

DESCRIPTION

A simulator driver task, *dcube*, provides a set of commands for loading and controlling a set of hypercube application programs built with the simulator library *libdcube.a* and optionally distributed over a network of computers. The simulator can be used to execute programs developed for the Intel hypercube, and a trace file is provided to assist in debugging or performance analysis. The simulator uses the UNIX fork, *rsh*, and TCP/IP facilities and has been tested on clusters of Sun workstations, IBM RT's, DEC VAX with UNIX 4.3, and a Sequent

Balance 8000 with DYNIX. The various application processes that constitute the hypercube simulation may be run on one machine or distributed over a network of machines. Since the simulation will spawn multiple processes when run on a single machine, you may wish to use *nice(1)* to reduce system load. When run on a single parallel-processor machine such as the Sequent or Encore, the simulation will utilize multiple processors. The simulation does not provide the detailed timings, large number of processors, or message-passing model of the simulator described in *ppsim(l)*, but does permit porting of programs between the Intel cube and the simulator with no source code changes. With *dcube* one can develop applications taking advantage of the large virtual memory and I/O services available from a collection of remote machines. The programmer compiles and links his host and node programs with *libdcube.a* and then runs *dcube* to load the application programs into the simulator cube and start the simulation.

The *c* commmand (cubelog) turns off logging with no arguments, or if a file name is provided, enables logging to the given file. Information will be appended to the file. The *h* command (host) specifies an executable file that will be the application host program and, optionally, a remote machine on which the application is to be run. It is recommended that the host program be run on the same machine as *dcube* (omit the *-h* option) so that the host program may perform terminal input and output. The *l* command (load – you may substitute *m* for *l*) loads the simulator nodes with the program file. A specific node may be loaded with the *-n* option. The *-h* option can be used to specify a remote machine on which the node program(s) are to be run. The *-c* option specifies the dimension of the cube and must be in the range 0-4. The number of processes that can be controlled by *dcube* is limited by the maximum number of files a process may open. The *t* command (trace) enables or disables tracing of simulator events *send*'s and *recv*'s. Once all application programs are "loaded" the *s* command (start) will start the simulation, you can exit with the *q* command (quit). If all processes exit then the simulator will exit, otherwise it will be necessary to use CTRL-C to terminate the simulation. Commands can be placed in a file and used with the *u* command. A sample session for running the programs on one machine might be

```
cc -o host host.c libdcube.a -lm
cc -o node node.c libdcube.a -lm
dcube
  t on
  l -c 3 node
  h host
  s
```

To run the node programs on a collection of machines on the network, one could construct the following command file, *sample.in*,

```
h host
l -c 2
l -n 0 -h mthsun node
l -n 1 -h crfsun node
l -n 2 -h puwsun node
l -n 3 -h yhesun node
```

then invoke *dcube* and use the following commands:

```
use sample.in
s
```

The functions provided in *libdcube.a* mirror those described in the Intel *iPSC User's Guide*. The programmer must first establish one or more cube communication channel data structures with calls to *copen*. *copen* takes a value to be used as process identifier and returns a descriptor, of type int, that is used in subsequent message-passing functions. A process is addressed by its node number and process identifier. *cclose* frees the channel data structure.

send and *sendw* send the message pointed to by msg to the process at node dstnode with process id dstpid. The type and size of the message (msglth) in bytes are also provided. *send* returns immediately, but one cannot use the message area until *status* returns FREE (0), indicating that the kernel has sent the message. *sendw* does not return until the message has been sent. (This does not imply that the message has been received.) *sendmsg* behaves exactly like *sendw* but is intended as the host version for compatibilty with INTEL cube.

recv and *recvw* await the arrival of a message of the given type for the node and process id associated with int d. The functions provide addresses to store the message, the actual length of the message, and the node and process id of the sender. For *recvw* the process blocks until a message of the given type arrives. For *recv* the process may continue processing after issuing the *recv*, and when *status* returns a value of FREE (0), then a message of the given type has arrived. Upon receipt of the message, the simulator sets the srcnode and srcpid to those of the sender, sets the msglth to the length of the received message, and copies the message into msg. No more than maxlth bytes are copied. Messages are handled in a FIFO fashion. *recvmsg* behaves like *recvw* except it does not discriminate on message type, rather the type of the message is returned along with the message in accordance with the INTEL cube. *sendmsg* and *recvmsg* are intended (by INTEL) to be used only by the host processes, but the simulator permits node usage as well.

probe determines if a message of the given type is available for the node and process id associated with the given int. If a message is available, *probe* returns the length of the message; otherwise, a value of -1 is returned. One must issue a *recv* actually to fetch the message. Note, the channel data structure should not be in use by other message-passing functions. *status* returns a value of BUSY (1) or FREE (0) indicating whether the given int data structure is in use or not. For *send*, BUSY indicates that the kernel has not yet sent the message. For *recv*, BUSY indicates that the desired message has not arrived.

mynode returns the node number of the process. The host has a node number of x8000 (32768). *cubedim* returns the dimension of the cube. *clock* returns the present value of the time-of-day clock in milliseconds. *syslog* places the given message and pid in the trace file. *flick* relinquishes control from the given process to other runnable processes. *flick* is usually used in busy-wait conditions with *status* following a *recv* or with *probe*.

Note that *dcube* uses one file descriptor (logical unit number 3) in managing the simulation through TCP sockets.

POST PROCESSORS

If tracing has been enabled then a trace file is produced with simulator data that can be summarized by *nstats* or *ccplot*. *nstats tracefile* will produce a per-node summary of compute time and sends and receives. *ccplot tracefile > plotfile* will produce a plotfile that can be plotted with various plotting programs such as *graph(1)*.

FORTRAN

The message passing subroutines may also be called from *f77* programs.

FILES

The following files are provided; the actual location is site dependent.

libdcube.a	simulator subroutines
dcube	simulator driver
rmtdcube	remote simulator application initiator
dbld	sample C build script

dfbld sample f77 build script

ASSUMPTIONS

It is assumed that *rmtdcube* is in the default execution path on the remote. It is sufficient for it to reside in the user's execution path. It is assumed that the executables to be run on the remote exist on the remote and that the user has accounts under the same name on all remotes. The various remotes should all be in the */etc/hosts.equiv* or the user's *.rhost* file.

SEE ALSO

mpsim(l), ppsim(l), hep(l), intel(l) and Intel's *iPSC User's Guide*

BUGS

The delay in message passing is due to delays in relaying the messages through TCP sockets and does not reflect a hypercube structure. Messages are passed through the simulator driving task *dcube*. The clock used is just the computer's time of day clock and has a resolution of only 20 milliseconds. Passing time-stamps between remote applications is not recommended since the clocks of the remote machines are not likely synchronized. Using a collection of heterogeneous machines as remotes may require additional application programming to account for difference in data representations (byte order, floating point format, etc.). When *dcube* terminates you may ignore the messages *nn: no such process*. The *handler* function and the Intel dynamic loader functions are not presently implemented.

AUTHOR

T. H. Dunigan

APPENDIX B
Source Code For Benchmarks

```
/* recth.c     rectangle rule     host    */
#define PID 0
#define TYPE1 1
#define TYPE2 2
struct Result {
      double pi;
      long ticks;
} result;
long n;
main()
{
      int p;
      int c,type,tmp, i;

      p = 1 << cubedim();
      c = copen(PID);
      while(1){
            printf("rectangles "); scanf("%ld",&n);
            for(i=0;i<p;i++)sendmsg(c,TYPE1,&n,sizeof(n),i,PID);
            recvmsg(c,&type,&result,sizeof(result),&tmp,&tmp,&tmp);
            printf(" pi %f in %ld\n",result.pi,result.ticks);
      }
}
```

```
/* rectn.c     rectangle rule     node    */
#define PID 0
#define TYPE1 1
#define TYPE2 2
#define TYPE3 3
#define ROOT 0

#define F(x) 4.0/(1.0 + x * x )

struct Result {
      double pi;
      long ticks;
} result;
long n;
main()
{
      int p,host,pid;
      int me,c,type,tmp;
      long i;
      double sum,x;
      long clock(), s;

      p = 1 << cubedim();
      c = copen(PID);
      me = mynode();
      while(1){
            recvw(c,TYPE1,&n,sizeof(n),&tmp,&host,&pid);
            s = clock();
            sum = 0.0;
            for(i=me;i<n; i += p) {
                  x = (i + 0.5)/(double)n;
                  sum += F(x);
            }
            if (me == ROOT){
              for (i=1;i<p;i++){
               recvw(c,TYPE3,&x,sizeof(x),&tmp,&tmp,&tmp);
               sum += x;
              }
              result.pi = sum/n;
              result.ticks = clock()-s;
              sendw(c,TYPE2,&result,sizeof(result),host,PID);
            } else sendw(c,TYPE3,&sum,sizeof(sum),ROOT,PID);
      }
}
```

```
/*  chol.h  */

#include <stdio.h>
#include <math.h>

#ifndef HOST
#define  HOST 0x08000
#endif
#define  PID       1
#define  FORWARD   1
#define  BACKWARD  0
#define  NATURAL   0
#define  GRAY      1
#define  I     32000
#define  M     32001
#define  B     32002

typedef float REAL ;
typedef int SHORT ;
typedef int LONG ;
typedef int CHNL ;

char *malloc() ;
```

```
/* cholh.c    Cholesky    host    */
#include "chol.h"

/* Message types used (defined in globals.h):
   I : problem info
   M : map
   B : right hand side vector
*/

/* Order of matrix is n.  Columns of matrix and components of rhs and
solution
   are numbered 0, 1, ... , n-1.  Nodes are numbered 0, 1, ..., np-1.
*/

main()
/* host process */
{
      CHNL ci ;
      REAL ti[5], mn[5], mx[5] ;
      int np, n, whichmap, whichord, whichcomm, maxblk, blksize,
            i, j, type, cnt, node, pid, info[4] ;
      int *map ;
      long clock(),s;
      static char *ordering[] = { "natural", "gray" } ;
      static char *mapping[] = { "wrap", "reflect" } ;
      static char *commun[] = { "bcube", "bcast", "ring" } ;

      ci = copen(PID) ;

      while (1)
      {
      printf("Enter number of processors:") ;
      scanf("%d", &np) ;
      printf("Enter order of matrix:") ;
      scanf("%d", &n) ;
      if ( np == 0 || n == 0 ) break ;
      if ( n < np ) break ;
      maxblk = n/np + ( n%np ? 1 : 0 ) ;
      printf("Enter wrap(0) or reflect(1) mapping:") ;
      scanf("%d", &whichmap) ;
      printf("Enter natural(0) or gray(1) ordering:") ;
      scanf("%d", &whichord) ;
      printf("Enter blocksize (1-%d):", maxblk) ;
      scanf("%d", &blksize) ;
      printf("Enter bcube(0), bcast(1), or ring(2) communication:") ;
      scanf("%d", &whichcomm) ;
      printf("Number of processors = %d, order of matrix = %d\n", np,n);
      printf("%s mapping, %s ordering, blocksize = %d, %s
communication\n",
            mapping[whichmap], ordering[whichord], blksize,
            commun[whichcomm]) ;

      s = clock();
      info[0] = np ;
      info[1] = n ;
```

```
        info[2] = whichord ;
        info[3] = whichcomm ;
        sendmsg ( ci, I, info, 4*sizeof(int), 0, PID ) ;

        map = (int *)malloc(n*sizeof(int)) ;
        whichmap = (whichmap<<1) + whichord ;
        setmap ( whichmap, map, n, np, blksize ) ;
        sendmsg ( ci, M, map, n*sizeof(int), 0, PID ) ;

        for ( j = 0 ; j < 5 ; j++ )
        {
              mn[j] = 1.0e+20 ;
              mx[j] = -1.0e+20 ;
        }
        for ( i = 0 ; i < np ; i++ )
        {
              recvmsg(ci, &type, ti, 5*sizeof(float), &cnt, &node, &pid);
              printf("node %d, fac =%7.2f, fslv =%7.2f, bslv =%7.2f",
                    node, (ti[1]-ti[0])/1000, (ti[2]-ti[1])/1000,
                    (ti[3]-ti[2])/1000) ;
              printf(" maxerr = %e\n", ti[4] ) ;
              for ( j = 1 ; j < 5 ; j++ )
              {
                    if (j==4)mx[j] = ti[j] > mx[j] ? ti[j] : mx[j] ;
                    else mx[j-1] = (ti[j]-ti[j-1])> mx[j-1] ?
                      (ti[j]-ti[j-1]) : mx[j-1];
              }
        }
        s = clock() - s;
        printf("totals , fac =%7.2f, fslv =%7.2f, bslv =%7.2f",
              mx[0]/1000, mx[1]/1000, mx[2]/1000) ;
        printf(" maxerr = %e\n", mx[4] ) ;
        printf(" In %ld ms\n",s);

        free(map) ;
        }

        cclose(ci) ;
}

setmap ( whichmap, map, n, np, blksize )
        int whichmap, *map, n, np, blksize;
{
        int i, gray();
        switch (whichmap)
        {
        case 0:                                                 /* wrap */
              for ( i = 0 ; i < n ; i++ ) map[i] = (i/blksize)%np;
              break;
        case 1:                                                 /* gray wrap */
              for ( i = 0 ; i < n ; i++ ) map[i] = gray((i/blksize)%np);
              break;
        case 2:                                                 /* reflect */
              for ( i = 0 ; i < n ; i++ ) map[i] = ((i/blksize)/np)%2
                    ? np-1 - (i/blksize)%np
                    : (i/blksize)%np;
```

```
            break;
        case 3:                                     /* gray reflect */
            for ( i = 0 ; i < n ; i++ ) map[i] = ((i/blksize)/np)%2
                  ? gray(np-1 - (i/blksize)%np)
                  : gray((i/blksize)%np);
            break;
        }
}

int gray (i)
    int i ;
{
   return( (i>>1)^i ) ;
}
```

```
/* choln.c    Cholesky    node   */

#include "chol.h"

main()
/* node process */
{
      CHNL ci ;
      long clock() ;
      REAL ti[5], maxerr ;
      int n, i, j, k, maxlth, ncols, np, me, cnt, node, pid,
            whichord, whichcomm, info[4] ;
      int *map, *collth, *mycols ;
      REAL *b, **col, *colk, *p, *q, *y ;
      REAL x, t ;

      me = mynode() ;
      ci = copen(PID) ;

      while (1)
      {

/* receive problem data */

      recvw ( ci, I, info, 4*sizeof(int), &cnt, &node, &pid ) ;
      np = info[0] ;
      n = info[1] ;
      whichord = info[2] ;
      whichcomm = info[3] ;
      if ( me == 0 ) for ( i = 1 ; i < np ; i++ )
            send ( ci, I, info, 4*sizeof(int), i, PID ) ;
      if ( np == 0 || n == 0 ) break ;
      maxlth = n*sizeof(REAL) ;
      map = (int *)malloc(n*sizeof(int)) ;
      recvw ( ci, M, map, n*sizeof(int), &cnt, &node, &pid ) ;
      if ( me == 0 ) for ( i = 1 ; i < np ; i++ )
            send ( ci, M, map, n*sizeof(int), i, PID ) ;
      ncols = 0 ;
      for ( i = 0 ; i < n; i++ ) if ( map[i] == me ) ncols++ ;

/* allocate storage */

      collth = (int *)malloc(ncols*sizeof(int)) ;
      mycols = (int *)malloc(ncols*sizeof(int)) ;
      b = (REAL *)malloc(n*sizeof(REAL)) ;
      col = (REAL * *)malloc(ncols*sizeof(REAL *)) ;
      colk = (REAL *)malloc(n*sizeof(REAL)) ;
      y = (REAL *)malloc(ncols*sizeof(REAL)) ;

/* set up data structures for problem to be solved */

      j = 0 ;
      for ( k = 0 ; k < n ; k++ )
            if ( map[k] == me )
```

```
            {
                  p = (REAL *)malloc((n-k)*sizeof(REAL)) ;
                  col[j] = p ;
                  mycols[j] = k ;
                  collth[j] = n-k ;
                  j++ ;
                  *p++ = 2 ;
                  if ( n-k > 1 ) *p++ = -1 ;
                  for ( i = 2 ; i < n-k ; i++ ) *p++ = 0 ;
            }
            for ( i = 0 ; i < n ; i++ ) b[i] = 0 ;
            b[n-1] = n+1 ;

      maxerr = 0 ;
      ti[0] = clock(0) ;

/* Cholesky factorization */

      j = 0 ;
      if ( map[0] == me )
      {
            t = 1.0/sqrt(*col[j]) ;
            for ( p = col[j] ; p < col[j] + collth[j] ; p++ )
                  *p *= t ;
            j++ ;
            comm ( ci, 0, col[0], n*sizeof(REAL), map[0], np,
                  FORWARD, whichord, whichcomm ) ;
      }
      for ( k = 0 ; k < n ; k++ )
      {
            recvw( ci, k, colk, maxlth, &cnt, &node, &pid ) ;
            if ( whichcomm != 1 && me != map[k] )
                  comm ( ci, k, colk, cnt, map[k], np,
                        FORWARD, whichord, whichcomm ) ;
            for ( i = j ; i < ncols ; i++ )
            {
                  for ( p = col[i] , q = colk + mycols[i]-k , t = *q ;
                        p < col[i] + collth[i] ; p++ , q++ )
                        *p -= *q * t ;
                  if ( k == mycols[j]-1 )
                  {
                        t = 1.0/sqrt(*col[j]) ;
                        for ( p = col[j] ; p < col[j] + collth[j] ;
                              p++ ) *p *= t ;
                        comm ( ci, mycols[j], col[j],
                              collth[j]*sizeof(REAL), me, np,
                              FORWARD, whichord, whichcomm ) ;
                        j++ ;
                  }
            }
      }

      ti[1] = clock(0) ;

/* forward substitution */
```

```
        j = 0 ;
        for ( k = 0 ; k < n ; k++ )
        {
                if ( map[k] == me )
                {
                        if ( k > 0 ) recvw ( ci, B, b, maxlth, &cnt, &node,
                                &pid ) ;
                        y[j] = b[k] / *col[j] ;
                        for ( i = k+1 ; i < n ; i++ )
                                b[i] -= y[j] * *(col[j]+i-k) ;
                        if ( k < n-1 ) send ( ci, B, b, n*sizeof(REAL),
                                map[k+1], PID ) ;
                        j++ ;
                }
        }

        ti[2] = clock(0) ;

/* back substitution */

        j = ncols-1 ;
        if ( map[n-1] == me )
        {
                x = y[j] / *col[j] ;
                if ( maxerr < fabs(x-n) )
                        maxerr = fabs(x-n) ;
                j-- ;
                comm ( ci, n-1, &x, sizeof(REAL), map[n-1], np,
                        BACKWARD, whichord, whichcomm ) ;
        }
        for ( k = n-1 ; k >= 0 ; k-- )
        {
                recvw( ci, k, &x, sizeof(REAL), &cnt, &node, &pid ) ;
                if ( whichcomm != 1 && me != map[k] )
                        comm ( ci, k, &x, cnt, map[k], np,
                                BACKWARD, whichord, whichcomm ) ;
                for ( i = j ; i >= 0 ; i-- )
                {
                        y[i] -= x * *(col[i]+k-mycols[i]) ;
                        if ( k == mycols[j]+1 )
                        {
                                x = y[j] / *col[j] ;
                                if ( maxerr < fabs(x-k) )
                                        maxerr = fabs(x-k) ;
                                comm ( ci, mycols[j], &x, sizeof(REAL), me, np,
                                        BACKWARD, whichord, whichcomm ) ;
                                j-- ;
                        }
                }
        }

        ti[3] = clock(0) ;
        ti[4] = maxerr ;
        send ( ci, 0, ti, 5*sizeof(float), HOST, PID ) ;

/* free storage */
```

```
        for ( j = 0 ; j < ncols ; j++ ) free(col[j]) ;
        free(y) ;
        free(colk) ;
        free(col) ;
        free(b) ;
        free(mycols) ;
        free(collth) ;
        free(map) ;
        }

        cclose(ci) ;
}

comm ( ci, msgtype, vec, bytes, root, np, dir, ord, which )
        CHNL ci ;
        int msgtype, bytes, root, np, dir, ord, which ;
        char *vec ;
/*
 *  Communication driver
 */

{
        switch (which)
        {
        case 0:
                bcube( ci, msgtype, vec, bytes, root, np ) ;
                break ;
        case 1:
                bcast( ci, msgtype, vec, bytes, root, np, dir, ord ) ;
                break ;
        case 2:
                ring( ci, msgtype, vec, bytes, root, np, dir ) ;
                break ;
        }
}

bcube ( ci, msgtype, vec, bytes, root, np )
        CHNL ci ;
        int msgtype, bytes, root, np ;
        char *vec ;
/*
 *  Broadcast vector, vec, of length bytes
 *  to all processors using a minimum spanning
 *  tree with given root.
 */
{
        int i, me, cnt, node, pid ;

        me = mynode()^root ;
        for ( i = np/2 ; i > me ; i /= 2 )
                send ( ci, msgtype, vec, bytes, (me+i)^root, PID ) ;
        if ( me == 0 )
                send ( ci, msgtype, vec, bytes, root, PID ) ;
}
```

```
bcast ( ci, msgtype, vec, bytes, root, np, dir, ord )
      CHNL ci ;
      int msgtype, bytes, root, np, dir, ord ;
      char *vec ;
/*
 *  Broadcast vector, vec, of length bytes
 */

{
      int i, cnt, node, pid, gray(), invgray() ;

      switch (ord)
      {
      case NATURAL:
            if ( dir == FORWARD )
            {
                  for ( i = 0 ; i < np ; i++ )
                        send ( ci, msgtype, vec, bytes,
                              (root+i)%np, PID ) ;
            }
            else
            {
                  for ( i = 0 ; i < np ; i++ )
                        send ( ci, msgtype, vec, bytes,
                              (root+np-i)%np, PID ) ;
            }
            break ;
      case GRAY:
            if ( dir == FORWARD )
            {
                  for ( i = 0 ; i < np ; i++ )
                  {
                        send ( ci, msgtype, vec, bytes,
                              succ(root,i,np), PID ) ;
                  }
            }
            else
            {
                  for ( i = 0 ; i < np ; i++ )
                        send ( ci, msgtype, vec, bytes,
                              pred(root,i,np), PID ) ;
            }
            break ;
      }
}

ring ( ci, msgtype, vec, bytes, root, np, dir )
      CHNL ci ;
      int msgtype, bytes, root, np, dir ;
      char *vec ;
/*
 *  Send vector, vec, of length bytes to
 *  all processors, using an embedded ring
 *  beginning at root.
 */
{
```

```
        int me, next, cnt, node, pid, pred(), succ() ;

        me = mynode() ;
        switch (dir)
        {
        case FORWARD:
              send ( ci, msgtype, vec, bytes, succ(me,1,np), PID ) ;
              break ;
        case BACKWARD:
              send ( ci, msgtype, vec, bytes, pred(me,1,np), PID ) ;
              break ;
        }
}

int pred ( node, i, np )
        int node, i, np ;
{
        int gray(), invgray() ;
        return( gray((invgray(node)+np-i)%np) ) ;
}

int succ ( node, i, np )
        int node, i, np ;
{
        int gray(), invgray() ;
        return( gray((invgray(node)+i)%np) ) ;
}

int gray (i)
    int i ;
{
   return( (i>>1)^i ) ;
}

int invgray (i)
    int i ;
{
   int k ;
   k = i ;
   while ( k > 0 )
       {
         k = k>>1 ;
         i = i^k ;
       }
   return (i) ;
}
```

```
/* rings.h   2 rings */
#define HOST 0x8000
#define START 0
#define RING 1
#define RESULT 2
#define PID 0
#define RPID 1
#define OFF 0
#define ON 1

struct PARM {
      int reps;  /* number of repetitions */
      int nnodes; /* number of nodes in ring */
      int msglth;
} parm;
```

```
/* host  ring*/
#include <stdio.h>
#include "rings.h"

main()
{
      int ci,lth,pid,node;
      int type;
      int maxnodes;
      long tim;
      double tdr;
      int i;

      maxnodes = 1<< cubedim();
      ci=copen(PID);
      while(1){
            printf("Number of nodes: ");scanf("%d",&parm.nnodes);
            printf("Number of revolutions: "); scanf("%d",&parm.reps);
            printf("Message size :"); scanf("%d",&parm.msglth);
            if (parm.nnodes>maxnodes) parm.nnodes = maxnodes;
            if (parm.msglth < 4 ) parm.msglth =4;
            if (parm.msglth > 15000 ) parm.msglth =15000;
            sendmsg(ci,START,&parm,sizeof(parm),0,PID);
             recvmsg(ci,&type,&tim,sizeof(tim),&lth,&node,&pid);
             printf(" Total time %ld ms  avrg/revolution %ld ms %ld ms
node\n",
             tim, tim/parm.reps, tim/parm.reps/parm.nnodes);
             tdr = (double)parm.reps * (double) parm.msglth *
parm.nnodes /
              tim ;
             printf(" Data rate %g KBs\n", tdr);
      }
}
```

/* ring node */

```
/* ringn.c
 * get parms if node 0 then spin the ring
 */
#include "rings.h"
int np;
long tim, start, clock();
int msg[8000];
main()
{
      int me,mepid;
      int ci, pid, node, lth;
      int cnt;
      int prev, next;

      mepid = PID;
      ci=copen(mepid);
      me=mynode();
      cnt = 0;
      while(1){
            if (me==0 && cnt ==0){  /* msg from host */
                  recvw(ci,START,&parm,sizeof(parm),&lth,&node,&pid);
                  np=parm.nnodes;
                  msg[0]=np;
                  next = gray((invgray(me)+1)%np);
                  cnt = parm.reps;
                  start = clock();
                  sendw(ci,RING,msg,parm.msglth,next,mepid);
            }
            recvw(ci,RING,msg,15000,&lth,&node,&pid);
            np=msg[0];
            next = gray((invgray(me)+1)%np);
            if (me==0 && --cnt == 0){ /* got ring message back */
                  tim = clock() - start;
                  sendw(ci,RESULT,&tim, sizeof(tim),HOST,PID);
            } else sendw(ci,RING,msg,lth,next,mepid);
      }
}
gray (i)
int i ;
{
      /* binary reflected gray code */
      return( (i>>1)^i ) ;
}

invgray (arg)
int arg ;
{
      int i ;

      for (i=0;arg;arg>>=1) i ^= arg;
      return (i) ;
}
```

```
/* echo host */

/* echo.c
 *   echo test to measure transmission delays
 * -r rate. -d cubedim -s startup -n repetition -l msglth  -p packetsize
 */

#include <stdio.h>

#define PID 11


#define CDIM 6
#define MSGSIZE 15000
#define REP 10
long times();
long ttemp[4];
struct PARM {
      long Reps;
      long Msglth;
      long Cdim;
};
#define reps parms.Reps
#define msglth parms.Msglth
#define cdim parms.Cdim

float atof();
      char buff[MSGSIZE];
      float total;
      float totals[CDIM+1];

main(argc,argv)
int argc;
char *argv[];
{
      int i;
      int d;
      long tmp;
      long end, start;
      float rate = 1.0;
      int startup =0;
      int psize =1;
      struct PARM parms;


      msglth = 1024;
      cdim = cubedim();
      reps = REP;
      while (--argc > 0 && **++argv == '-')switch ((*argv)[1]) {
            case 'r':   /* rate */
                  rate = atof(argv[1]);
                  argc--; argv++;
                  break;
            case 'n':   /* reps */
                  reps = atoi(argv[1]);
```

```
                argc--; argv++;
                break;
            case 's':   /* startup */
                startup = atoi(argv[1]);
                argc--; argv++;
                break;
            case 'l':   /*  msg lth */
                msglth = atoi(argv[1]);
                argc--; argv++;
                break;
            case 'p':   /* packet size */
                psize = atoi(argv[1]);
                argc--; argv++;
                break;
            case 'd':   /* cube dimension*/
                cdim = atoi(argv[1]);
                argc--; argv++;
                break;
            default:
                break;
        }
        d = copen(PID);
        while(1){
        printf("reps lth"); scanf("%d%d",&reps, &msglth);
        printf(" repetition(n) %ld cube dimension(d) %ld  msg lth(l)
%ld\n",
         reps,cdim,msglth);

        /* now start node tests */
        sendmsg(d,1,&parms,sizeof(struct PARM),0,PID);
        recvmsg(d,&tmp,totals,sizeof(totals),&tmp,&tmp,&tmp);
        for (i=0;i<cdim+1;i++)
         printf("node 0 hop of %d average (roundtrip) time  %g sec %g KBs
\n",
          i,totals[i]/(reps*1000),reps*2.0*msglth/totals[i]);
        printf("tests complete\n");
        }
}
```

```
/* echo node */

/*  echon.c     run with echoh to do roundtrip timing tests */

#define PID 11
#define HOST 0x08000
#define CDIM 6
#define MSGSIZE 15000
struct PARM {
      long Reps;
      long Msglth;
      long Cdim;
};
#define reps parms.Reps
#define msglth parms.Msglth
#define cdim parms.Cdim
char buff[MSGSIZE];
long clock();

main()
{
      /* do node to node tests */
      long tmp;
      int d;
      int lth;
      int srcnode, srcpid;
      long start;
      int i, dim, n;
      float totals[CDIM+1];
      struct PARM parms;

      d = copen(PID);
if (mynode() == 0)
    while(1){
      /* wait for host to say we can start */
      recvw(d,1,&parms,sizeof(struct PARM),&tmp,&srcnode,&srcpid);
      n=0;
      for(dim=0;dim<cdim+1;dim++){
            totals[dim] = 0.;
            start = clock();
            for(i=0;i<reps;i++){
                  sendw(d,1,buff,(int)msglth,n,PID);
                  recvw(d,1,buff,(int)msglth,&tmp,&tmp,&tmp);
            }
            totals[dim] = clock()-start;
            n = (n<<1) + 1;
      }
      sendw(d,1,totals,sizeof(totals),HOST,PID);  /* tell em we are done
*/
      syslog(PID,"node 0 restart");
  }

 else
      /* if not node 0 just echo back whatever we get */
```

```
        while(1){
                recvw(d,1,buff,MSGSIZE,&lth,&srcnode,&srcpid);
                sendw(d,1,buff,lth,srcnode,srcpid);
        }
}
```

APPENDIX C
Ethernet Traffic Generated By Benchmarks

Ethernet Traffic In Rectangle Rule Benchmark With Operating System IBM ACIS/4.3 - Packets over Time in 7 Second Increments.

0-dimensional	21	0	1	2	3	0	10	3	35	64	3
1-dimensional	14	0	39	25	50	37	3	0	1	0	0
2-dimensional	28	27	43	8	44	3	3	0	1	0	0
3-dimensional	54	63	29	6	3	0	1	4	0	0	0

Ethernet Traffic In Rectangle Rule Benchmark With Operating System AIX. - Packets over Time in 7 Second Increments.

0-dimensional	12	1	0	0	1	9	12	1	0
1-dimensional	16	1	3	17	2	1	0	1	2
2-dimensional	25	30	0	0	3	4	0	0	1
3-dimensional	55	53	0	1	0	0	3	2	0

Ethernet Traffic In Rectangle Rule Benchmark With Mixed Operating Systems. - Packets over Time in 7 Second Increments.

1-dimensional	44	28	36	58	34	14	4	0	1
2-dimensional	31	8	26	2	1	1	2	3	1
3-dimensional	42	41	4	0	2	0	1	2	4

Ethernet Traffic In Cholesky Matrix Factorization Benchmarks With Operating System IBM ACIS/4.3. - Packets over Time in 7 Second Increments.

0-dimensional	133	151	155	148	150	153	149	150	53
1-dimensional	198	289	262	150	159	196	268	268	124
2-dimensional	395	611	509	174	159	312	544	520	137
3-dimensional	612	1120	1016	355	165	490	1046	1032	397

Ethernet Traffic In Cholesky Matrix Factorization Benchmarks With Operating System AIX. - Packets over Time in 7 Second Increments.

0-dimensional	134	140	141	140	140	141	143	145	109
1-dimensional	242	252	241	147	148	175	246	253	142
2-dimensional	462	497	434	206	157	155	400	446	449
3-dimensional	1038	1029	908	155	163	552	978	955	418

Ethernet Traffic In Cholesky Matrix Factorization Benchmarks With Mixed Operating Systems. - Packets over Time in 7 Second Increments.

1-dimensional	249	267	242	152	155	221	271	260	30
2-dimensional	410	583	499	161	158	344	536	523	163
3-dimensional	847	1133	964	168	166	673	1058	1021	183

Ethernet Traffic In Ring Message Passing Benchmarks With Operating System IBM ACIS/4.3. - Packets over Time in 7 Second Increments.

0-dimensional	471	634	105	166	18	2	1	1	2
1-dimensional	577	782	781	338	61	6	1	2	1
2-dimensional	448	788	786	791	786	766	573	0	42
3-dimensional	575	783	791	784	792	780	787	786	782

Ethernet Traffic In Ring Message Passing Benchmarks With Operating System AIX. - Packets over Time in 7 Second Increments.

0-dimensional	947	262	6	0	0	2	1	0	1
1-dimensional	961	1117	322	0	0	1	0	0	3
2-dimensional	936	1124	1134	1133	529	8	7	0	1
3-dimensional	1573	1896	1741	1847	1926	1858	1905	1960	

Ethernet Traffic In Ring Message Passing Benchmarks With Mixed Operating Systems. - Packets over Time in 7 Second Increments.

1-dimensional	698	856	853	63	98	0	2	4	0
2-dimensional	725	857	865	858	860	764	37	132	11
3-dimensional	701	861	847	866	858	848	854	855	854

Ethernet Traffic In FORTRAN Ring Message Passing Benchmarks With Operating System IBM ACIS/4.3. - Packets over Time in 7 Second Increments.

0-dimensional	423	635	150	0	1	3	1	0	4
1-dimensional	378	780	780	524	81	21	1	2	3
2-dimensional	375	784	783	790	792	790	625	3	1
3-dimensional	558	783	784	791	784	787	745	772	779

Ethernet Traffic In Echo Message Passing Benchmarks. - Packets over Time in 7 Second Increments.

4.3	467	646	761	779	780	754	782	781	778
AIX	1862	1909	1950	1850	1968	1905	1872	1907	1955
Mixed	606	637	781	778	781	778	779	779	826

VITA

Steven L. Kilgore was born in Louisville, Kentucky on February 9, 1965. He attended elementary schools in Knoxville, Tennessee and was graduated from Tennessee High School in Bristol, Tennessee in June 1983. The following September he entered the University of Tennessee, and in December 1986 he received a Bachelor of Arts degree in Computer Science with minors in Mathematics and Psychology. The following January he re-entered the University of Tennessee and began study toward a Master's degree in Computer Science.