

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview of CM-5</b>	<b>2</b>
<b>3</b>	<b>CMMD Library</b>	<b>4</b>
<b>4</b>	<b>Timing and Method of Computation</b>	<b>6</b>
<b>5</b>	<b>Cooperative Message Passing Functions</b>	<b>8</b>
<b>6</b>	<b>Global Communications</b>	<b>21</b>
<b>7</b>	<b>Circular Shift</b>	<b>24</b>
<b>8</b>	<b>Summary</b>	<b>26</b>

# List of Figures

1	The components of a typical processing node. . . . .	4
2	NI provides access to the facilities of Data and Control Networks. . . . .	4
3	A close view of the time versus message size curve for cooperative message passing functions. . . . .	9
4	Blocking message passing functions with message sizes multiple of 4 bytes.	10
5	Blocking message passing functions with message sizes not multiple of 4 bytes. . . . .	11
6	Vector versions of send and receive ( <b>echo</b> ) with various vector strides. . .	13
7	Vector version of <b>exchange</b> for various vector strides. . . . .	14
8	Vector version of <b>swap</b> for various vector strides. . . . .	15
9	<b>Async</b> version of the point-to-point communications. . . . .	16
10	<b>Noblock</b> version of the point-to-point communications. . . . .	17
11	The performance of the message passing functions depend on the starting position of the message buffer.(Message size 16,000 bytes) . . . . .	19
12	Dependency of <b>swap</b> on the starting position of the message buffer. . . .	19

13	The performance of the message passing functions depends the number of repeats of the same function. . . . .	20
14	Comparison of broadcast and scatter. . . . .	22
15	Comparison of scatter, gather and data transpose. . . . .	23
16	Performance of the user-constructed circular shift operation. . . . .	25

## List of Tables

1	Performance of the blocking point-to-point communication functions. . .	9
2	Vector version of the point-to-point communication. . . . .	12
3	Performance of cooperative communication functions with respect to the levels of the hierarchy traversed. . . . .	18
4	Global communication functions (pure communication). . . . .	21
5	Speed (MFLOPS) of the reduction and scan operations. . . . .	24
6	Performance of the shift operation, where <i>shift</i> indicates the size of the shift. . . . .	25

# PERFORMANCE OF THE CM-5 MESSAGE PASSING PRIMITIVES\*

Kesheng Wu and Youcef Saad  
Computer Science Department

March 4, 1993

## Abstract

This report presents a number of experiments conducted on the CM-5 to analyze the performance of the CMMD message passing functions. We measure the various parameters of the communication primitives such as bandwidth, communication latency, effect of distance, etc.. Overall the communication functions have high bandwidth and variable latency. For practical uses, odd size message and misaligned message buffer are to be avoided.

*The results in this paper are based upon a test version of the software where the emphasis was on providing functionality and the tools necessary to begin testing the CM5 with vector units. This software release has not had the benefit of optimization or performance tuning and, consequently, is not necessarily representative of the performance of the full version of this software.*

## 1 Introduction

Thinking Machines Corporation's CM-5 provides an extensive message passing library, CMMD. It is quite important to analyze interprocessor communication performance alone since communication often determines overall performance in computational algorithms. The goal of this report is to provide preliminary measurements in this framework.

The analysis of the communication primitives provided in this report was started as part of a class project in the course "Issues in parallel programming and performance" taught by the second author in the Winter quarter of 1992. The goal of the project was to conduct a large number of experiments involving communication alone. The first task that one must accomplish when receiving a new equipment, be it a computer or any other machine, is to test its performance under various conditions. This is particularly true for new models because of the likelihood of minor design flaws not seen by the manufacturer.

---

\*This work is supported in part by DARPA/NIST under grant number 60NANB2D1272, and by AHPCRC (University of Minnesota) under Army Research Office grant number DAAL03-89-C-0038.

A large number of measurements were conducted and a number of observations have been made. A few anomalies were found. We noted that other students in the class have also made similar observations. One of the ‘anomalies’ observed was that the performance seems to exhibit jumps by a factor of 4, when the length of the messages passed is a multiple of 16 bytes. For example, for the swap operation, one can reach either close to 16MB/s speed or 4MB/s depending on whether or not the message length is a multiple of 16 bytes.

These anomalies were observed when the CMMD was at version 1.0. Currently on the CM-5 at the Army High Performance Computing and Research Center, University of Minnesota, the operating system CMOST is version 7.2, the CMMD version number is 2.0. In this new released software, some of the earlier bugs have been fixed. The benchmarks with these new libraries should be closer to what one can expect from the machine.

The rest of this report is organized as follows. Section 2 gives a full description of the CM-5. Section 3 introduces the functions in the CMMD library. Section 4 explains the details of the experiments and the computation of the start-up time and the bandwidth. Section 5 will show data to illustrate various aspects of the Data Network, which handles point-to-point communications. The section which follows will present the performance of the global communication functions which are related to Control Network of the CM-5. Section 7 gives an example of user-constructed communication function, a circular shift operation. The last section gives a few brief comments about the observed performance of the CM-5.

## 2 Overview of CM-5

The basic component for carrying out computational tasks on the CM-5 is called a Processing Node(PN)[5], or simply a *node*. It consists of one basic SPARC chip, one Network Interface (NI) chip, some memory, and four vector units, see fig. 1. The NI chip is physically part of the SPARC chip, though its functions are independent of the regular components of a SPARC chip. Each node has 16 megabytes of memory.

The processors are separated by electronic protective firewall into *partitions*. Each partition has a single control processor that is called Partition Manager(PM). Currently the PMs are separate Sun workstations. They execute the sequential parts of the user program, act as the host part in all the host-node communication, and are also responsible for managing the execution of the user program in the partition. From a user point of view, the machine is accessible from the PM’s and execution of a program is initiated from it. Each program is executed on one partition of the machine; any access to the resources outside of the partition must go through the PM.

The partition manager has a full implementation of an enhanced Unix operation system (CM Operating System, or CMOST). The PNs also have a limited implementation

of CMOST, which support some system (CMOS) calls.

The program languages supported on the CM-5 are: C and Fortran 77 for using the MIMD mode; C\*, CMF and \*lisp for the SIMD mode. In SIMD mode, the system takes care of the communications for the user. In MIMD mode, users are to use the communication library functions to explicitly program the communication. For doing node level communication, usually the CMMD[8] library is used. If desired, lower level library functions are also available for directly programming the NI chip[6]. Besides the communication library, the CM-5 also has utilities such as CMRTS (CM Fortran Run-Time system), CMSSL(CM Fortran Scientific Software Library), CMFS (File system library) etc.

All the PNs are connected together with three networks, namely, the Data Network (DN), the Control Network (CN) and the diagnostic network. The diagnostic network provides the CM-5 with the ability to isolate any hardware component and test both that component and all connections to other components. It involves all the hardware system, but is invisible to the user. The DN is a high-bandwidth, asynchronous, point-to-point network. Its main responsibility is to route messages to and from PNs and I/O devices. Message delivery is guaranteed to be dead-lock free, and conflicts are fairly arbitrated. The control network performs the synchronization, broadcast, scatter, gather, scan and reduction operations. It has very low latency.

The Data Network has a topology of a fat-tree[1, 4]. This hierarchical structure makes it possible to sustain a high bandwidth between any pair of nodes in the partition. Every four PNs are connected to two identical data routers. These four PNs may communicate with each other through the two routers. Every communication that involves two nodes that are not within a group of 4 will require the message to pass through a higher level in the hierarchy. The raw data injection rate of the NI chip is 20 MegaBytes/second (MB/s). The data router has the capability of passing the data to or from the four PNs at 20MB/s. In the fat-tree, each of the processing nodes and the two lowest levels of the data routers has two parents, i.e. two upward paths; each higher level data router has four such paths. This structure should be able to maintain a bandwidth of 20 MB/s within a group of 4 nodes, 10 MB/s within a group of 16 nodes, and 5 MB/s within a group of 64 nodes or more.

The DN is divided into left and right halves, which may work independently, or with some collaboration. Both halves act identically, but they are not mutually exclusive. From a PN's view, there are two connections to the DN, so it may treat the DN as two parts. Though the DN does have the capability of processing two messages from one node, the network is not really separated.

The Control Network composed of three sub-networks, each with a unique function, (1) the *broadcast network* distributes a single numerical value to every node; (2) the *combine network* receives a single value from each node, combines them arithmetically or logically, then distributes the combined result to all nodes; (3) the *global network* handles global synchronization of nodes. It is to be noted that just like the left and

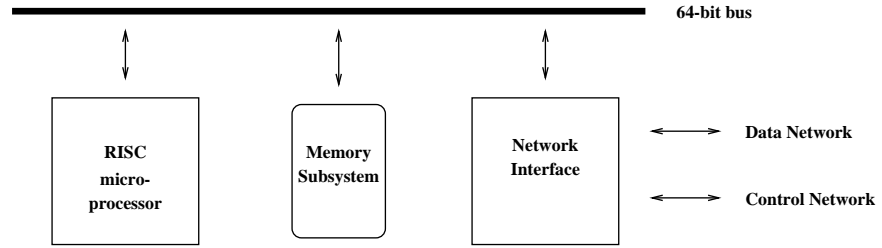


Figure 1: The components of a typical processing node.

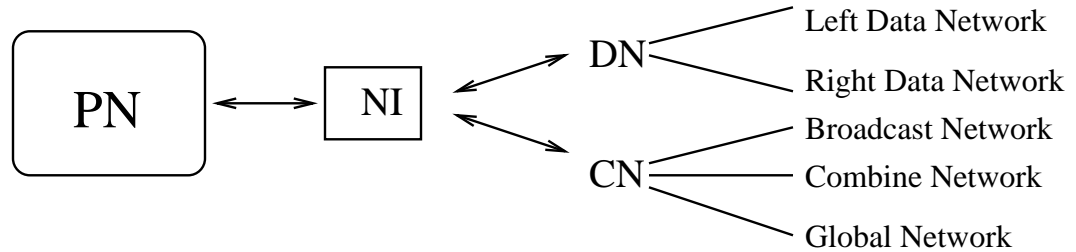


Figure 2: NI provides access to the facilities of Data and Control Networks.

right half-network of the DN, these three sub-networks here are also not exclusive.

Within each node, the NI chip provides the PN with a standard interface with the networks, see fig. 2. It provides a memory mapped control-register interface to the memory bus. Its network functions are controlled entirely by reading and writing registers that have been mapped into the memory address space of the microprocessor. From another angle, the NI is a special area of memory with unique contents in the memory cells. FIFO queues are used to buffer the messages for sending and receiving.

### 3 CMMD Library

The CMMD is the main library provided for programming in MIMD mode on the CM-5. The programming mode can be either host/node mode or hostless node[7, 8]. When an executable program is called on the host (PM), the host will invoke the initialization procedures necessary, then run the copy of the sequential code on itself, and each node in the partition will get an identical copy of the parallel code and start execution.

CMMD supports both *blocking* and *non-blocking* send and receive. For most of the functions there also exist a vector version of the message passing functions.

There are two general types of message passing functions, (1) point-to-point; and (2) global communication. The first type is supported in the lower level by the Data Network (DN). The type (2) is supported by the Control Network (CN). Global communication will be explained in detail later.

For the point-to-point communication, there are three groups of commands that can be used to achieve the same purpose. Command type (1) is a pair called *send* and *receive*. Command type (2) is named *send\_and\_receive*. Command type (3) is a special case of the type (2), called *swap*. It is clear what the operations *send*, *receive* and *swap* should perform. The *send\_and\_receive* receives a message from one node, and at the same time sends a message from the current node somewhere else. In the current version, all the nodes involved in this command have to call the same command, in other words, a message sent out by *send\_and\_receive* will have to be received by the same function. When this command is invoked with same node for the source and destination, it forms a *swap* operation. Commands type (2) and (3) may operate on one node, and furthermore both the buffer for input and output may be the same one. The outgoing buffer is duplicated to avoid the possibility of sending out a buffer with an unintended message.

Each of the above functions also has a vector version. In this case, the message to be sent can consist of pieces from buffers that are not contiguous, and this type of receiving function will buffer the message, then distribute the elements of the message to positions in the buffer with a specified stride. This type of function needs extra time in packing and distributing the message. It should be very useful when processing vectors with regular stride, e.g. sending every other element somewhere, or receiving a sequence of numbers and treating them as the first row of a 2-D array (Fortran column-major notion).

The asynchronous functions are of two different types, a **noblock** version which copies the message buffer if it can not deliver the message at the time of the call; an **async** version which returns a Message Control Block (MCB). Only **send** and **receive** are available for asynchronous message passing. The **noblock** version leaves the message buffer reusable immediately. The **async** version leave the program with more control of the message being processed.

Global communication functions include broadcast, scatter, gather, data transpose, scan, and reduce. All these functions involves all the nodes and may or may not include the host in addition. The broadcast operation can be from host to nodes, or from one node to the rest of the partition. The scatter operation distributes elements of the given length from the specified buffer to each node in processor order. The gather operation is the reverse of this, one element from each node is collected and then an array on the host is formed from the data. The scan operation is also referred to as parallel prefix operation in the literatures. Both scan and reduction functions perform tasks like,

- Summing all the values across all the nodes.
- Finding the maximum or the minimum value from the values contributed by each node.

- Performing bitwise logical operations in the values.

The result of the reduction may either return to all nodes or to the host. On the other hand, the scan operation will leave partial results in each processor. All nodes must take part in all these operations.

Communication between the host and one node is not a real point-to-point communication as is the case between two nodes in a partition[6]. It is actually a ‘global communication’ of some type. When the host sends data to a node, all the nodes will receive the data, but those nodes that are not intended targets will discard the message received. When a node sends data to the host, all nodes in the partition contribute data, the real sender will contribute the data to be sent, the rest contribute null data.

Besides communication functions, the CMMD library also includes functions that control the partition (like enabling or disabling a partition), provides information about the partition (like partition size), about a node (like node number) and the status of the message queue, etc. Synchronization among all nodes or only two nodes can be achieved with separate functions. Using blocking communication synchronizes two nodes involved as well.

Overall, CMMD permits cooperative concurrent processing, and synchronization only occurs between the nodes involved in a blocking communication. At all other times, computation on each node proceeds asynchronously.

## 4 Timing and Method of Computation

The issues related to the way in which the timing is done and how the bandwidth and start-up time are computed will be briefly explained here.

For the functions that allow communication between two nodes, timings are done on one of the processors involved. Since the `send_and_receive` and `swap` operations act identically on both sides, the two times measured on each end are very close, so using one of them is enough. In case a send and a receive are separately issued, there must be one processor that sends out a message first, and the second processor will receive it and send it back to the first one. It is reasonable to use the time that it takes the first node to send and receive the message as the time for this complete cycle of message passing. In computing the rate of the message passing, we consider the time measured as for passing the same message two times, i.e.,

$$\text{rate} = 2 \times \text{message size} / \text{time}$$

In the case of the broadcast, scatter, gather and data transpose, the timing is done by starting a timer on each node from the host, then taking the largest time among all the nodes. The communication rate is taken to be the size of the buffer on the host divided

by the time to complete the operation. Since the start-up time could take a significant part of the total time needed to send the message, it is important to use a timing model that incorporates it. We use the following simple model for the timing,

$$\text{time} = \text{start-up time} + \text{message size} / \text{bandwidth} \quad (1)$$

This equation has been proven by the results of the experiments to be good approximation of the relation between time and the message size. The start-up time and the communication bandwidth are computed from fitting the above linear equation to the observed times and message sizes. We take this as the definition of the start-up time and the communication bandwidth in this report. Notice that this definition of the start-up time and bandwidth is not the same as what is commonly used. Some authors[4, 1, 2, 3] define the start-up time to be the observed time needed for sending an empty message, and the bandwidth to be the measured asymptotic communication speed as the message size goes to infinity. From an application programmer's point of view, the above equation provides a way of estimating the communication time for various message sizes. The two constants obtained from fitting the data to the equation should give a closer representation of the actual time versus the message size relation. For the cases where the above linear relation does not hold over a large range, we may use linear approximations for subregions of the data. The zero-byte speed definition of start-up time and bandwidth may be too simplistic in a general situation. An ambiguity arises in the asymptotic bandwidth (or sustainable bandwidth) also: one obvious question is how large a message must be in order to be considered large enough? In most architectures, there exist several levels in the memory hierarchy, and using different message sizes may require very different access times. This could change the observed communication rate significantly. Based on the linear fitting definition (1), the start-up time may sometimes be negative due to following reasons:

1. Statistical error, for example, a start-up time may be computed to be  $-10 \pm 20\mu\text{s}$ . statistically, this start-up time can be taken to be very close to zero.
2. Poor representation of the actual time by formula (1). This can be caused by an unusual behavior in the software or hardware which leads to quadratic or higher order dependencies. Due to the extreme complexity involved in the communication software and hardware, the presence of higher order dependency is quite conceivable. This effect might be especially noticeable for small messages.

From the experiments on the communication time, we observe in general that

- There is an overall start-up time for a communication channel. Depending on the message size, it could vary form 0.1ms(milliseconds) to 5ms.
- This is only associated with the first 1 or 2 messages passed through the same channel. Thereafter, for further messages along the same channel the communication time is reduced.

- If we discard the time from the first one or two iterations, the timing shows much better consistency.

As a result, all the data presented here is taken after the channel selected has been used sufficiently. Therefore this overall start-up time for the communication channel is not counted as a part of the normal time needed for passing any message.

As a reminder, we should again mention that most of the data presented here has been collected during the month of December of 92. The CMMD library is at version 2.0 and the CMOST is at release 7.2.

## 5 Cooperative Message Passing Functions

This section is devoted to the cooperative message passing functions. There are three ways of exchanging messages between two nodes. First we use two pairs of `send` and `receive` functions, which we will call *echo*. Second, we can use the general `send_and_receive` function, which we will call *exchange*. This function can make use of the both halves of the DN. The third choice is to use a swap function. In the case of *echo*, the start-up time refers to the start-up time for one pair of send and receive functions. The time for one send and receive pair is assumed to be half the time for the whole echo process.

Figure 3 provides us with a very first look at the performance of the three types of communication routines. In the plot, the message size changes from 0 to 60 bytes. The y-axis shows the times for completing the communications. With an empty message, the time to complete the desired communication is obviously shorter. Intuitively the start-up time should be close to the time to complete a communication with 1-byte (or 1-word) message which is in the range of 220-280  $\mu$ s for a complete *echo*. Using the linear fit, the start-up time for one pair of send and receive operation is estimated to be 119  $\mu$ s. Another observation we get from this plot is that the message size can be roughly divided into two categories: multiple of 4 bytes and not multiple of 4 bytes. Later experiments will confirm this observation. Figure 4 shows the performance of the cooperative message passing functions with message size of multiple of 4 bytes. Figure 5 shows the performance with the other type of message size. Table 1 shows the summary of the performance measured in start-up time and communication bandwidth.

From the data in table 1, we can see that the send and receive pair has a smaller start up time. However, to complete the *echo*, we must have two of these start-ups. It's surprising to see that the bandwidth of `send_and_receive` function is slightly lower than that of send and receive pair. The `swap` operation has a considerably higher communication rate.

In terms of the communication bandwidth, there is about a two fold difference between messages containing only whole words (multiple of 4 bytes) and those containing broken words. In numerical applications, one would be passing messages containing complete

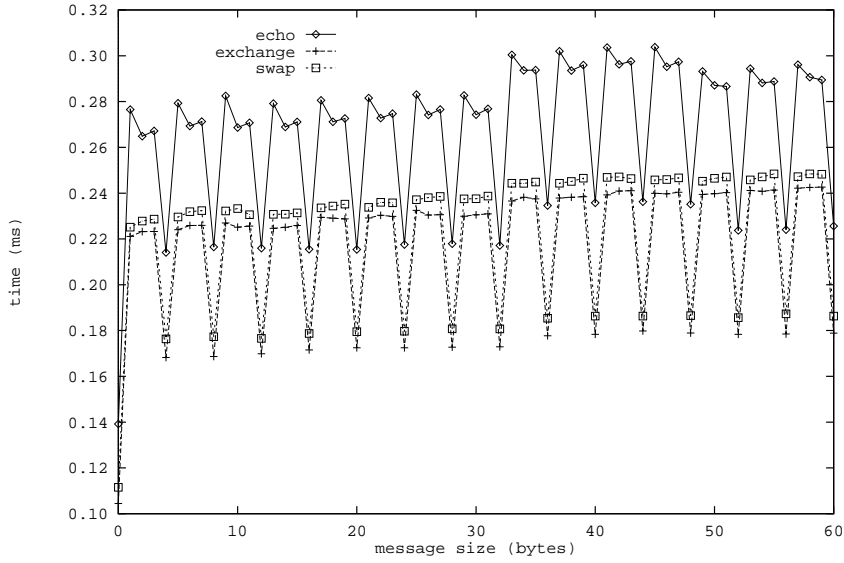


Figure 3: A close view of the time versus message size curve for cooperative message passing functions.

Message size	multiple of 4		not multiple of 4	
	$\mu\text{s}$	MB/s	$\mu\text{s}$	MB/s
<i>echo</i>	119	9.4	41	3.6
<i>exchange</i>	159	9.3	141	5.3
<i>swap</i>	192	11.9	132	5.3

Table 1: Performance of the blocking point-to-point communication functions.

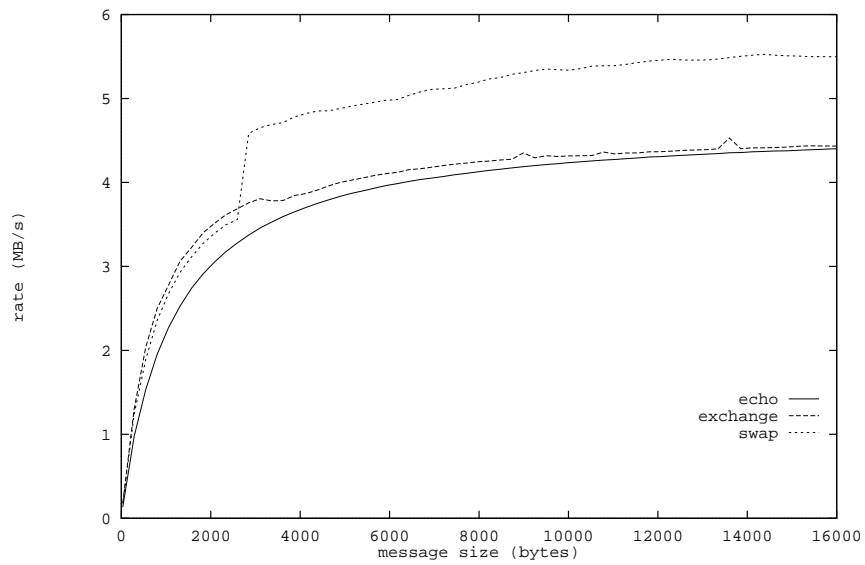
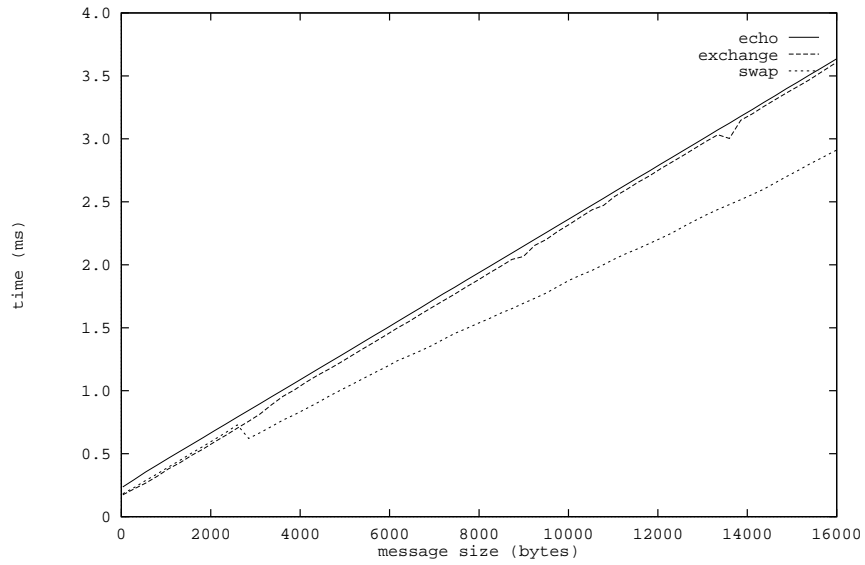


Figure 4: Blocking message passing functions with message sizes multiple of 4 bytes.

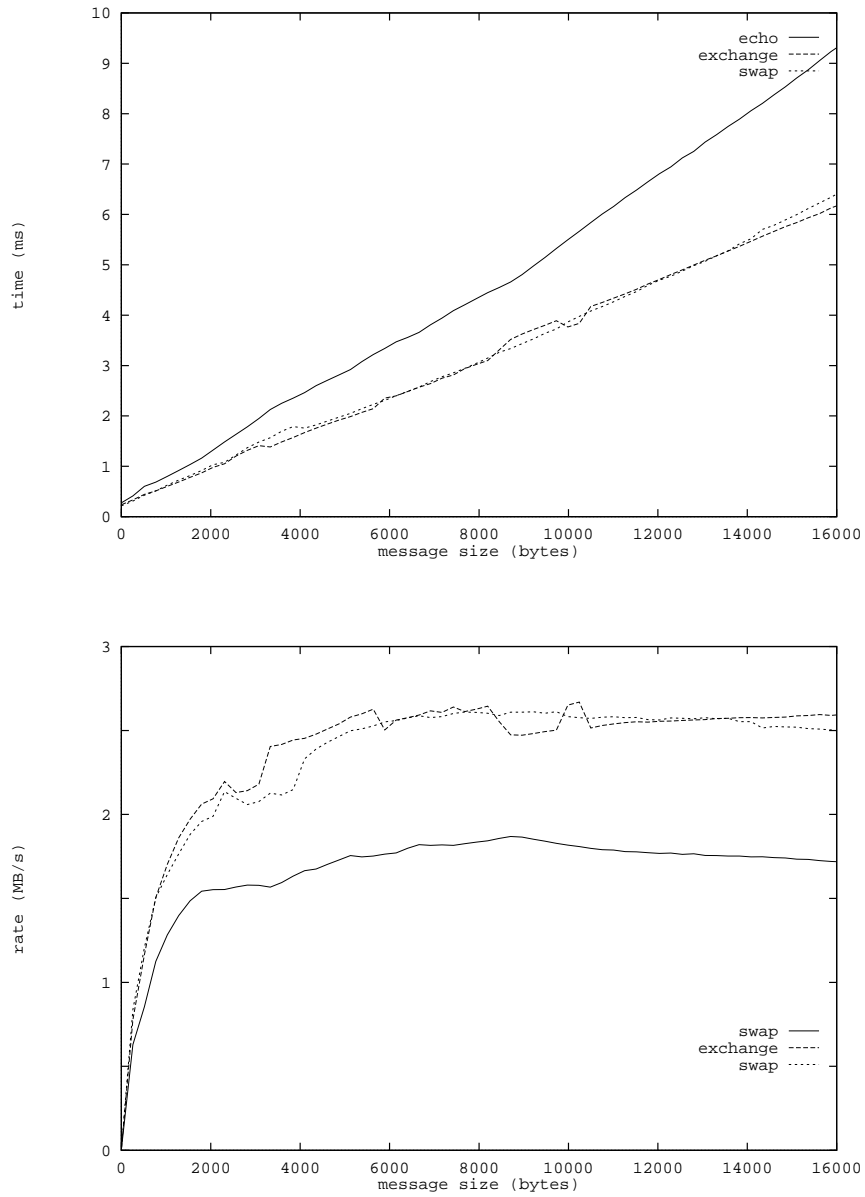


Figure 5: Blocking message passing functions with message sizes not multiple of 4 bytes.

stride	4 bytes		8 bytes		16 bytes		32 bytes	
	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s
<i>echo</i>	125	9.7	144	5.9	145	3.2	133	2.4
<i>exchange</i>	190	6.9	207	5.4	201	4.0	206	4.8
<i>swap</i>	198	6.9	209	5.2	200	4.6	171	3.1

Table 2: Vector version of the point-to-point communication.

words, therefore this performance degradation should have no effect. In the application that uses character strings, care should be taken to avoid this pitfall.

The CMMD provides a set vector version[7] for most of the message passing functions. Here we list the performance of the vector versions of the cooperative message passing functions in table 2, and plot the time versus message size in figure 6, 7, and 8. The vector version of the communication functions generally first pack the vector elements from the memory, then send the message through the network. Upon receiving such a message, the vector receiving functions, will unpack the message and place the elements into the correct positions in the message buffer. The specification of the vector elements include the element length (in bytes), stride (in bytes) and number of elements. For the messages tested here, the element size is 4 bytes. The start-up time and the bandwidth are obtained from fitting the data with a number of elements varying from 0 to 200. For this test, both the send and the receive functions use the same vector element specification. It is clear from the three plots and the table that as the stride increases, it takes longer to complete the communication. Notice that the performance of the *echo* when the stride equals the element size is about the same as that of the sequential version, but the *exchange* and *swap* operations take a serious performance degradation.

For point-to-point communication, nonblocking versions of the send and receive functions are available. Tests of the nonblocking send and the blocking receive yield a start-up time of 1.2 millisecond and a communication bandwidth of 11.2 MB/s. In this test the message sizes are multiple of 4 bytes. And the sizes range form 32 bytes to 16384 bytes. For message size of about 16 Kbytes, the time for the *echo* with blocking functions is about 6% shorter than that of the nonblocking version. Similar tests with asynchronous send and receive functions show a start-up time of about 223  $\mu$ s and a bandwidth of about 12.0 MB/s. This start-up time includes the clean-up operation required. The difference between the *noblock* version and the *async* version of the send and receive are described in detail in [7]. Figure 9 shows the performance of the *async* version of the *echo*. Figure 10 shows the *noblock* ones. Notice that the start-up time and the bandwidth for the *async* version are computed using only message sizes ranging from 32 bytes to 6000 bytes. With larger message sizes, the time versus message size deviates significantly from our expectations. The reason for this is currently unknown to us.

The messages passed between the closest two nodes, i.e. nodes within the smallest

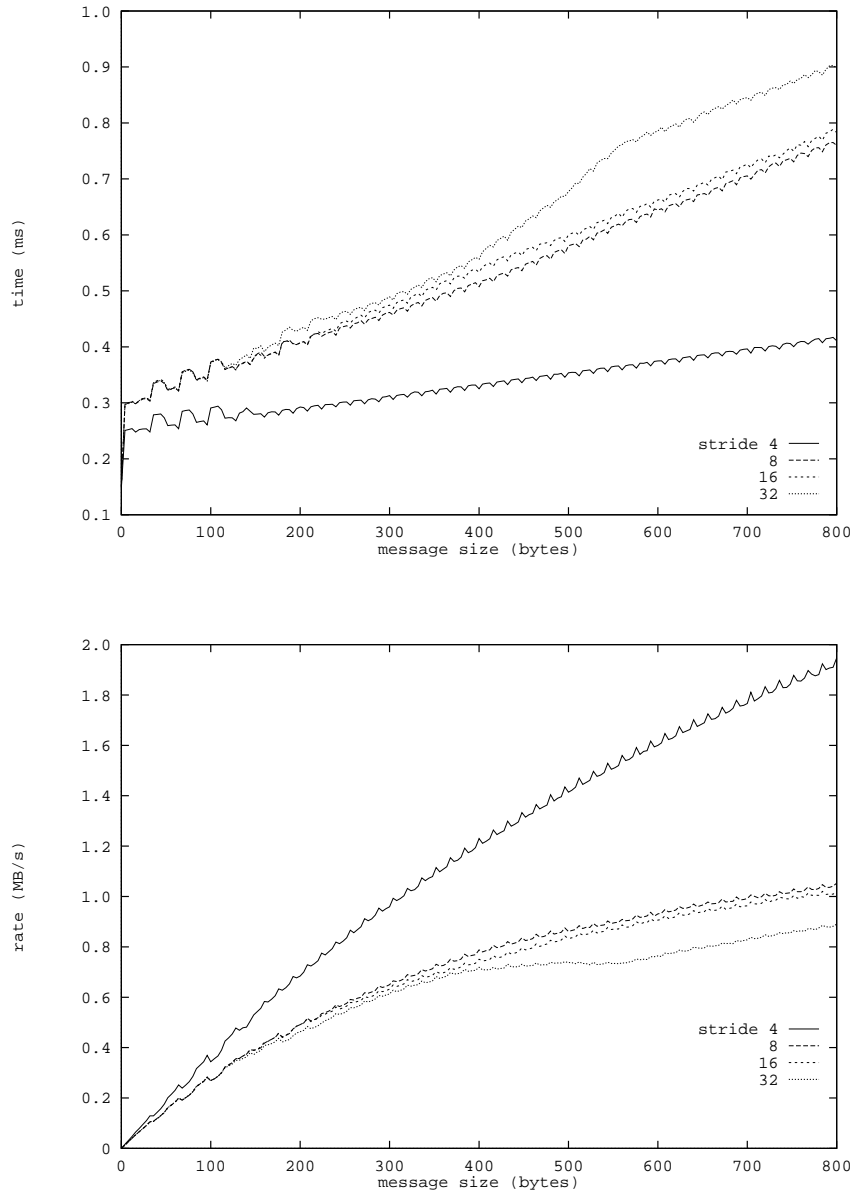


Figure 6: Vector versions of send and receive (`echo`) with various vector strides.

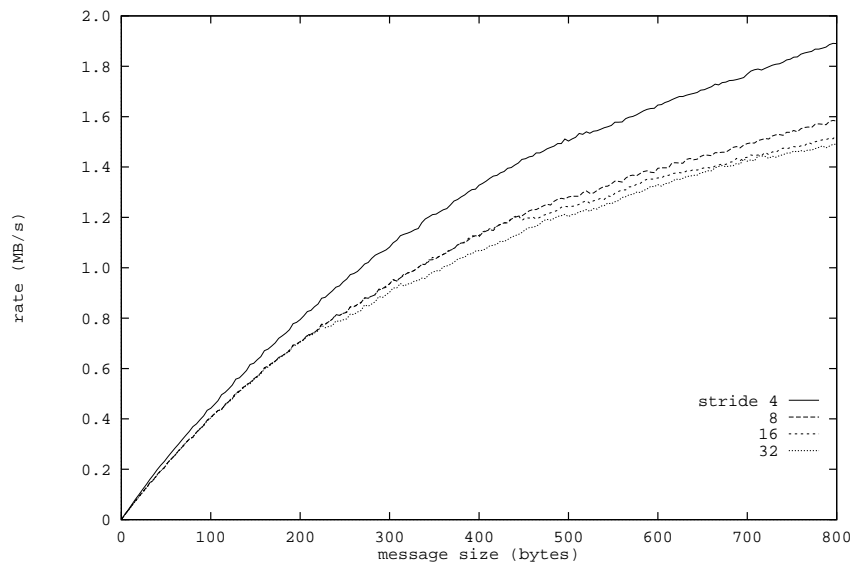
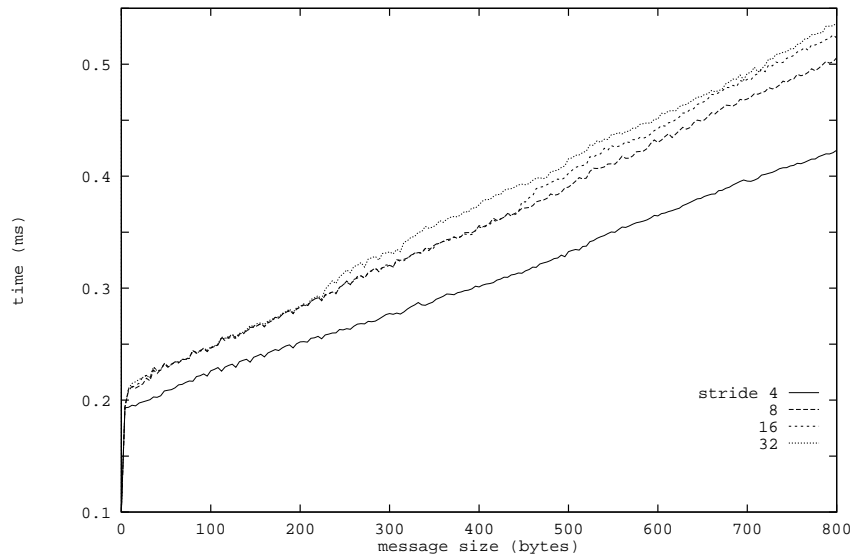


Figure 7: Vector version of `exchange` for various vector strides.

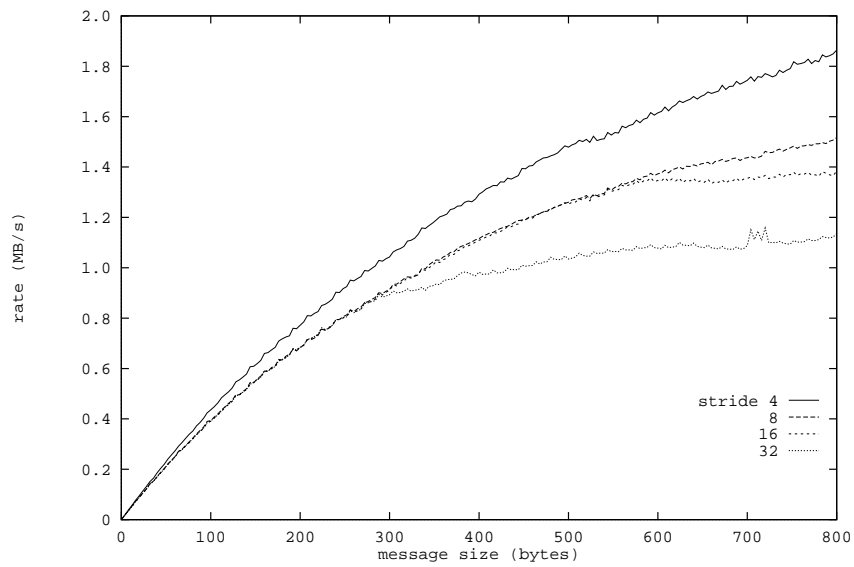
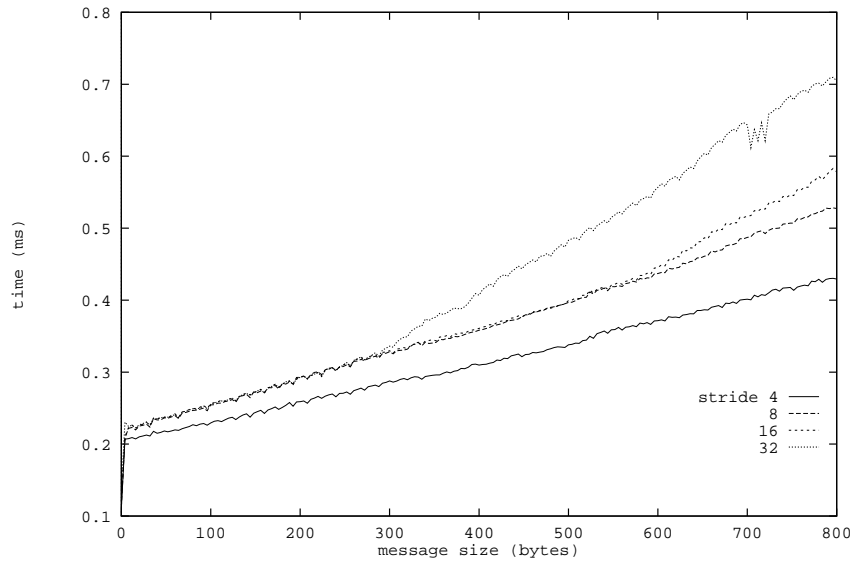


Figure 8: Vector version of `swap` for various vector strides.

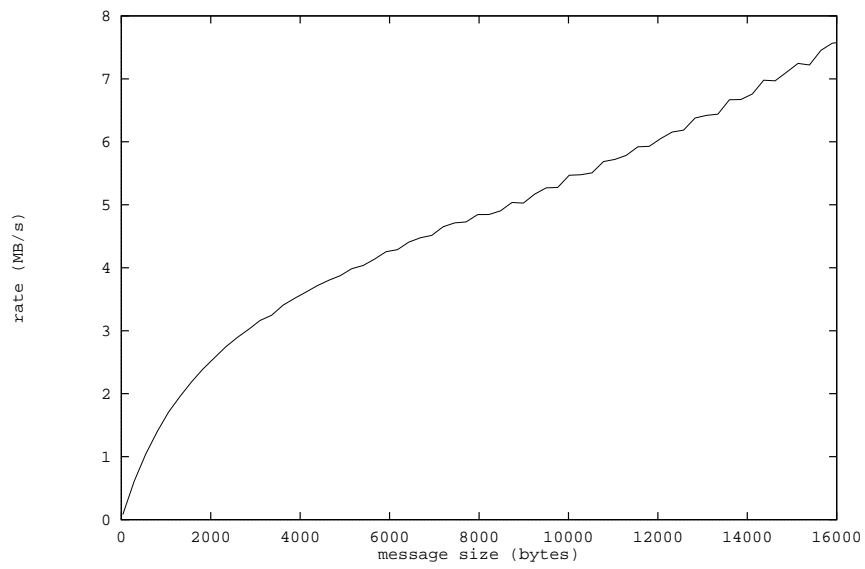
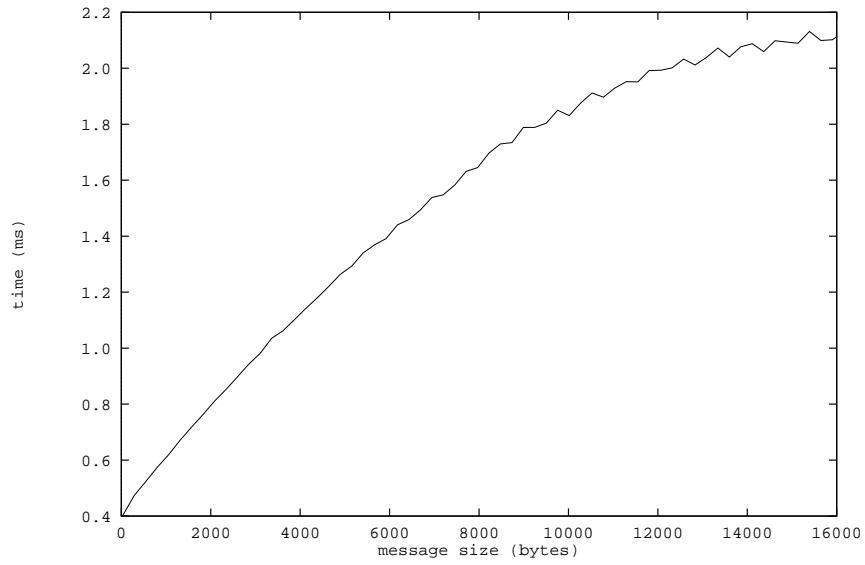


Figure 9: Async version of the point-to-point communications.

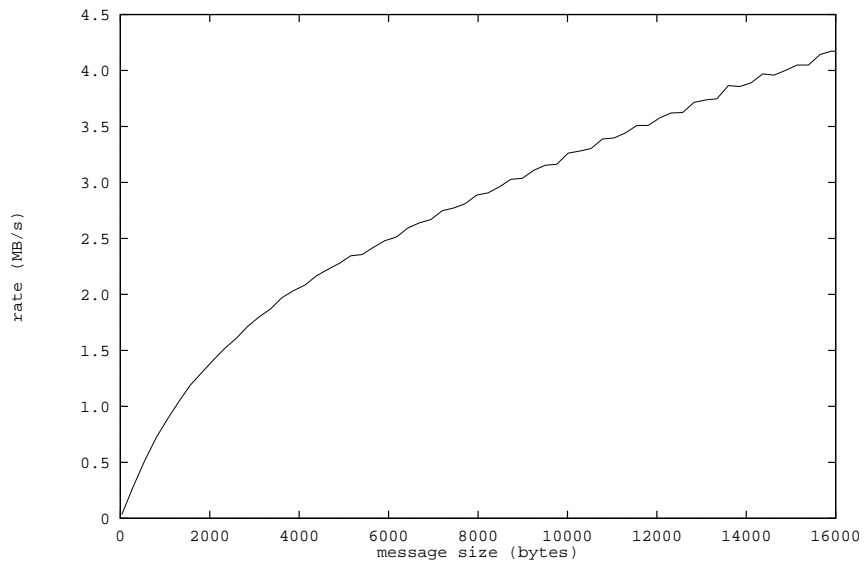
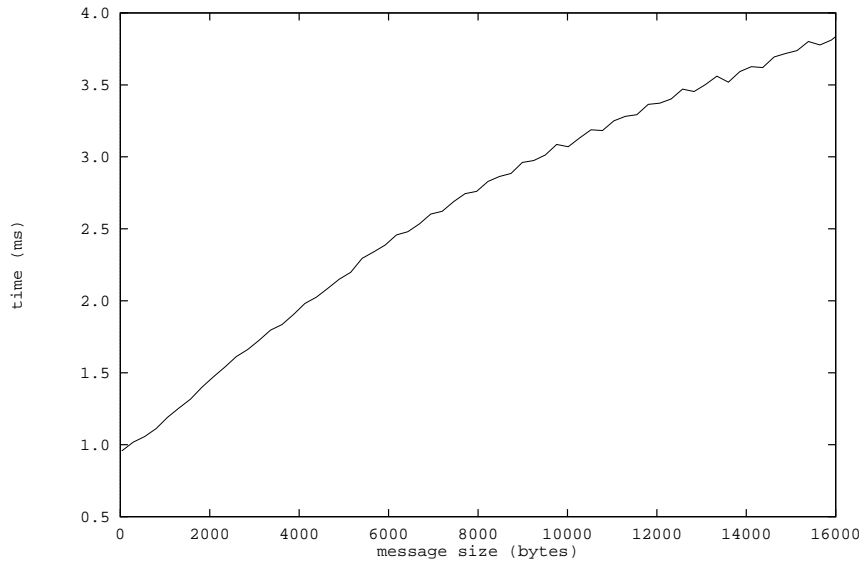


Figure 10: Noblock version of the point-to-point communications.

levels	1		2		3		4		5	
	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s
<i>echo</i>	119	9.4	129	9.4	136	9.4	148	9.4	618	9.2
<i>exchange</i>	159	9.3	227	9.8	212	9.7	183	9.5	281	10.7
<i>swap</i>	192	11.9	195	11.6	201	11.7	200	11.7	185	11.6

Table 3: Performance of cooperative communication functions with respect to the levels of the hierarchy traversed.

unit in the hierarchy will leave from one node to a data router then reach the other node directly. This involves one level of the Data Network. For the nodes that are not in the same group, the message passed between them will have to go through higher levels of the Data Network. Table 3 lists the performance of the cooperative message passing with respect to the levels traversed by a message. It can be concluded that with only one or two messages in the network, the performance does not differ significantly when messages travel between different pairs of nodes in the partition.

The performance degradation for non-aligned message buffers is substantial. For example, when *echoing* 16 Kbytes between two nearby nodes, using word-aligned message buffers takes about 3.65 ms, while using non-word-aligned buffers takes 5.22 ms (figure 11). Using non-aligned buffer costs 43% more. Observing from figure 11, it is clear that operations with word-aligned message buffers perform much better than with non-aligned buffers. The best performance for **exchange** and **swap** operations are obtained with double-word-aligned buffers. Figure 12 shows the time of the **swap** under various buffer positions for two different messages, one with message size of 16,000 bytes, the other with 16,001 bytes. It is obvious that the influence of the position of the buffer is much weaker on the message of size 16,001 bytes. This is true for all message sizes not multiple of 4 bytes. Also word-aligned buffers for messages of size 16001 bytes can achieve the best performance.

When time is measured at each iteration of the communication, the first iteration always takes significantly longer than the later iterations, see figure 13. For example, still passing message of 16 Kbytes between neighboring nodes, the first iteration takes about 3.87 ms. This is about 6% longer than the later iterations. When the message size is small, say 16 bytes, the first iteration takes about 494  $\mu$ s, while later iterations only take about 246  $\mu$ s. The first iteration takes more than twice the time. Also note that the odd message sizes seem to cause larger difference between the first and later iterations, see figure 13.

There are a few obvious irregularities in most of the performance curves. Some of them may be due to the random properties of the packet-switching mechanism of the communication network or possibly to software problems.

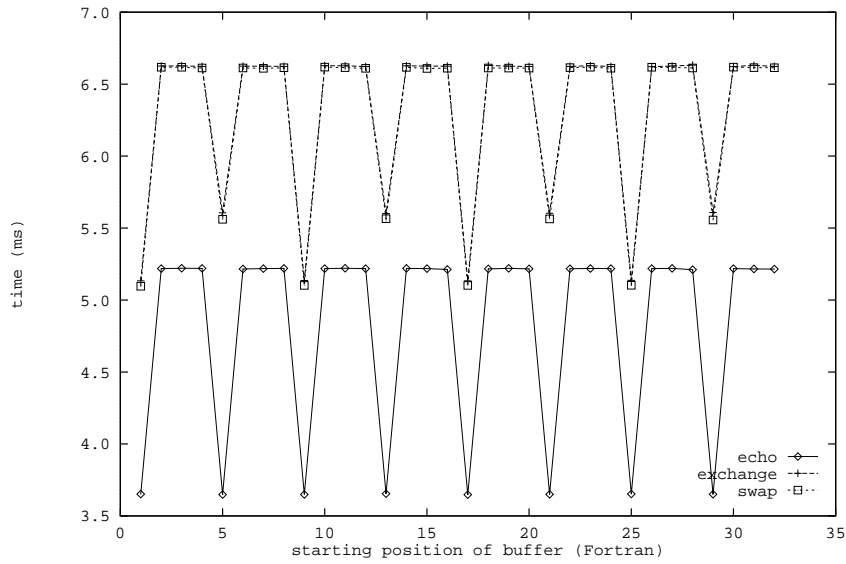


Figure 11: The performance of the message passing functions depend on the starting position of the message buffer.(Message size 16,000 bytes)

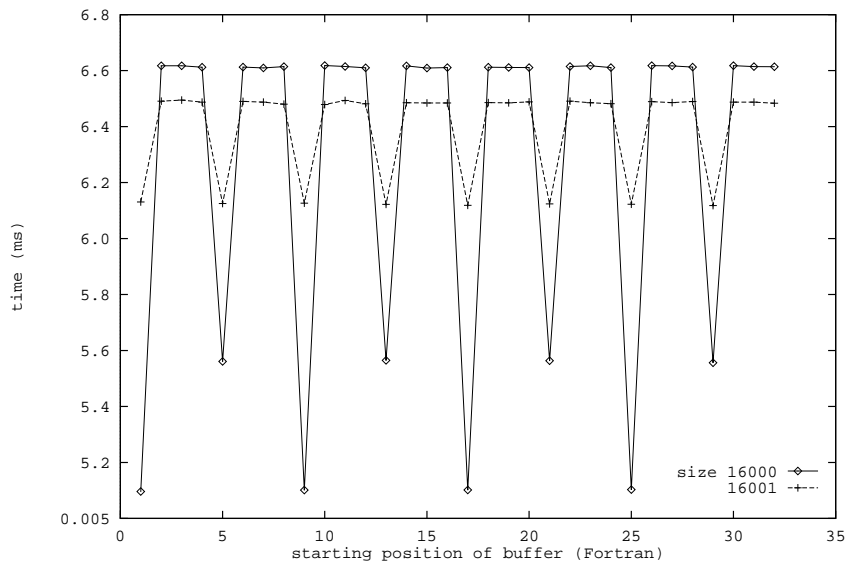


Figure 12: Dependency of swap on the starting position of the message buffer.

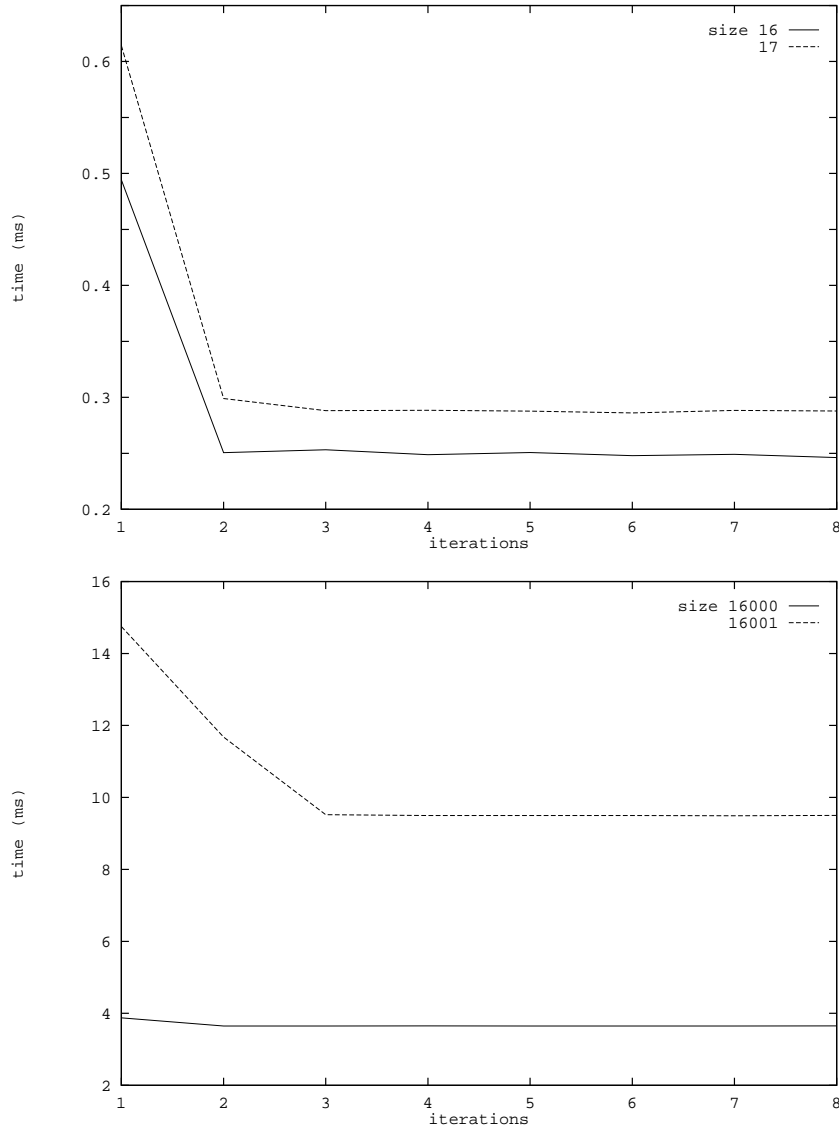


Figure 13: The performance of the message passing functions depends the number of repeats of the same function.

Message size	multiple of 4		not multiple of 4	
	$\mu s$	MB/s	$\mu s$	MB/s
broadcast	23	2.2	-17	2.1
scatter	23	2.2	-151	1.3
gather	39	1.2	14	0.90
transpose	17	0.94	4.5	0.64

Table 4: Global communication functions (pure communication).

## 6 Global Communications

The global communication functions of interest to us here are the broadcast, scatter, gather, data transpose, scan and reduction operations. In addition to these, there are a variety of functions for synchronization and other purposes. All of them are based on the functionalities of the Control Network (CN).

It is obvious (table 4) that the latencies of the global communication functions are extremely low. One can think of the scatter function (`distrib_to_nodes`) as being the same function as the broadcast, only that the first function will discard part of the data coming to the nodes. Figure 14 supports this. The gather function takes a longer time to collect the same number of bytes than the scatter operation, see figure 15. In the case of the transposition, since each node is sending and receiving large amounts of data, congestion may be a serious problem and low bandwidth is expected. The data for figure 14 and 15 are from a 32-node partition. The peaks in the performance in the plots occur at message sizes of multiple of 128 bytes, which is 4 bytes per node. Further experiments on different size partitions agree with this observation. The difference in the performance of the scatter and the broadcast (figure 14) occurs when the message to each node is not a multiple of 4 bytes. Looking at the behavior of the scatter operation, for message size that are not multiple of 4 bytes (per node), the communication rate seems to be decreasing as the message size increases. This behavior agrees with the linear fitting result with negative start-up time (table 4).

There are two global communication functions that can be more appropriately measured in MFLOPS rather than MB/s since arithmetic operations are the core of the functionality, namely the reduction and scan operations. The reduction operation can perform simple arithmetic or logical operations on the data contributed by each node, and return the final result to each node or the host. The scan operation returns a running tally on each node. Both operations have vector versions. The data in the table 5 are collected from repeating the same operation for 10,000 times. Comparing data from the two different size partitions, it is clear that the MFLOPS rates are proportional to the size of the partition. This indicates that the operation takes about the same amount of time on different size partitions.

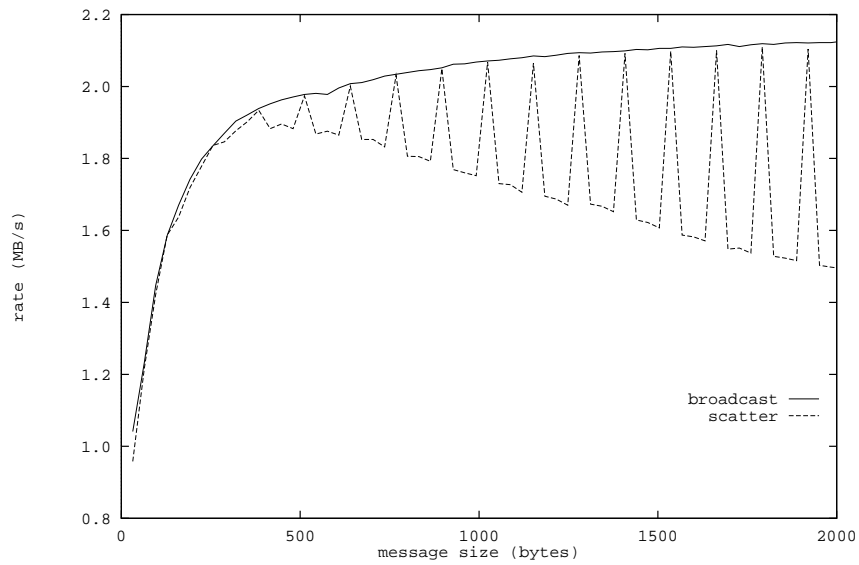
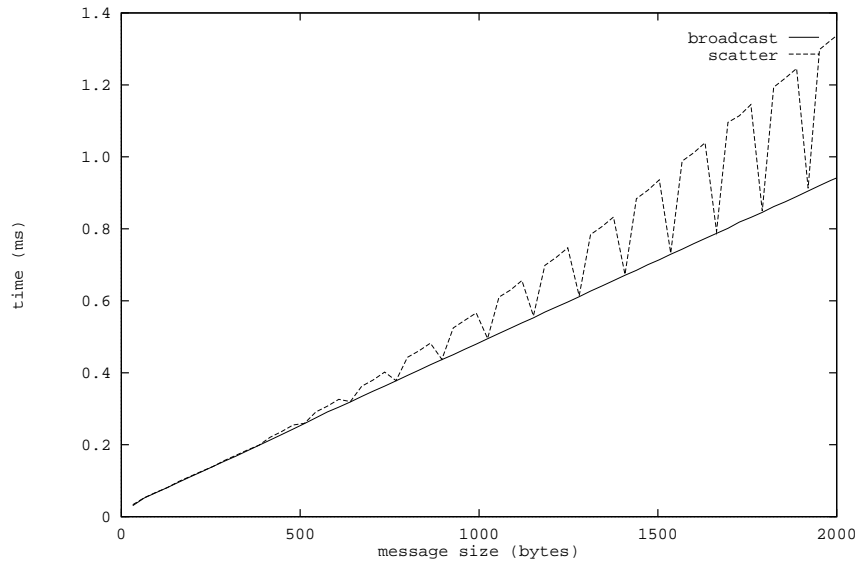


Figure 14: Comparison of broadcast and scatter.

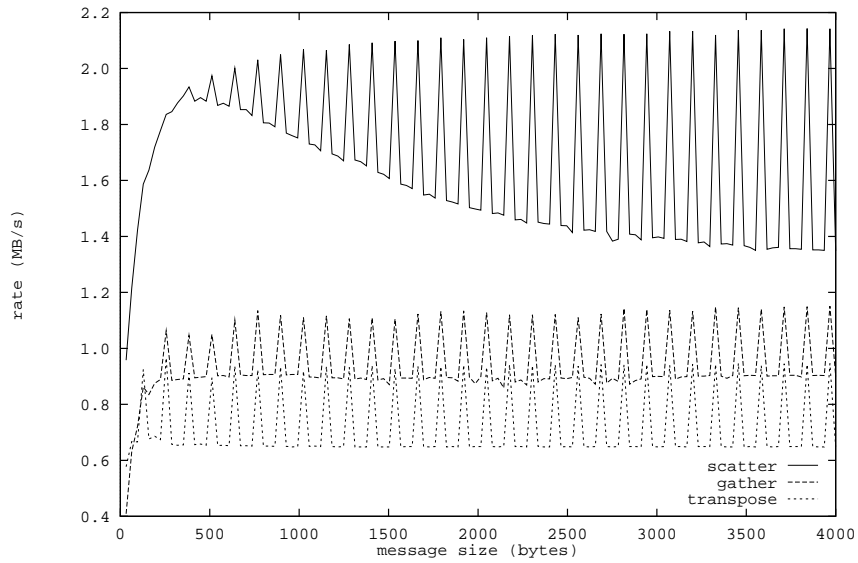
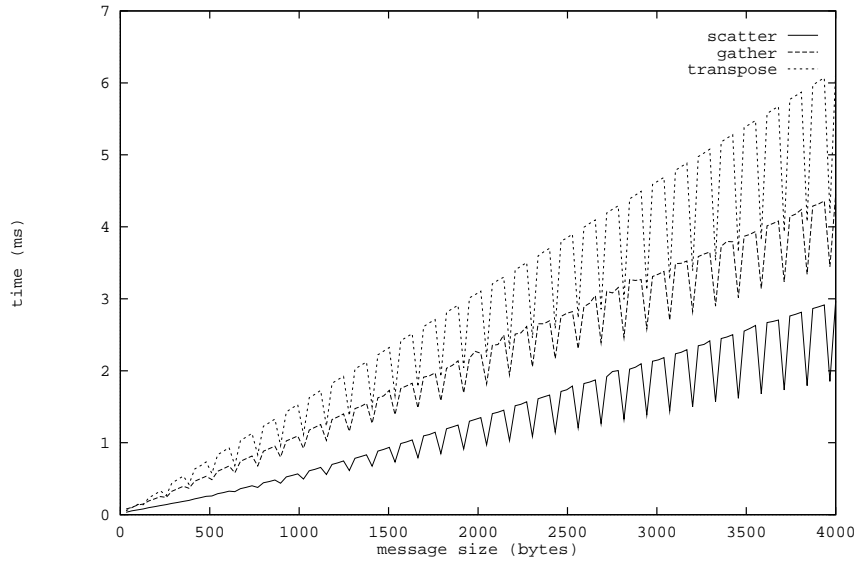


Figure 15: Comparison of scatter, gather and data transpose.

32 PNs	reduction			scan		
operation	add	min	max	add	min	max
integer	4.6	5.0	4.6	3.2	3.3	3.1
single	0.75	0.94	1.0	0.62	0.75	0.81
double	0.96	1.3	1.5	0.79	1.0	1.1
512 PNs	reduction			scan		
operation	add	min	max	add	min	max
integer	75	80	75	51	54	51
single	11	15	16	9.6	12	13
double	15	21	24	12	16	18

Table 5: Speed (MFLOPS) of the reduction and scan operations.

Finally in this section we will briefly mention the communication between host and individual nodes. A simple test shows that for communication between the host and one node, with a message size multiple of 4 bytes, the average communication rate is about 3.0 MB/s. The computed start-up time is about 187  $\mu$ s. For message size not multiple of 4 bytes, the bandwidth is measured to be 2.1 MB/s and the start-up time is 177  $\mu$ s. Notice that these communication speeds are closer to the speed of global communication functions than to the speed of point-to-point communication function supported by the DN.

## 7 Circular Shift

As an example of user constructed functions, we tested a simple circular shift function. The circular shift can be constructed using one of the three ways in which the message passing between two nodes can be performed, i.e. using separate `sends` and `receives`, the general exchange function (`send_and_receive`), or the `swap`. Only blocking message passing functions are used in this test. Notice from table 6 that the general exchange functions are best suited for the circular shift operation. We observe a bandwidth of 11.5 MB/s and less than 100  $\mu$ s start-up time. The bandwidth is measured on per node basis. The performance reduction observed here does not exceed a factor of 3. The particular larger performance reduction in the first type of shift operation may be due to the naive arrangement of the communication pattern which causes congestion in the data network. Figure 16 shows the performance of the three types of the shift functions with message sizes multiple of 4 bytes. All the time versus message size curves are very close to linear.

<i>shift</i>	1		2		4		8	
	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s	$\mu$ s	MB/s
<i>echo</i>	235	9.1	236	9.1	78	3.8	156	4.8
<i>exchange</i>	93	11.5	98	11.6	35	10.5	-2	9.5
<i>swap</i>	124	7.6	126	7.6	100	7.0	174	7.0

Table 6: Performance of the shift operation, where *shift* indicates the size of the shift.

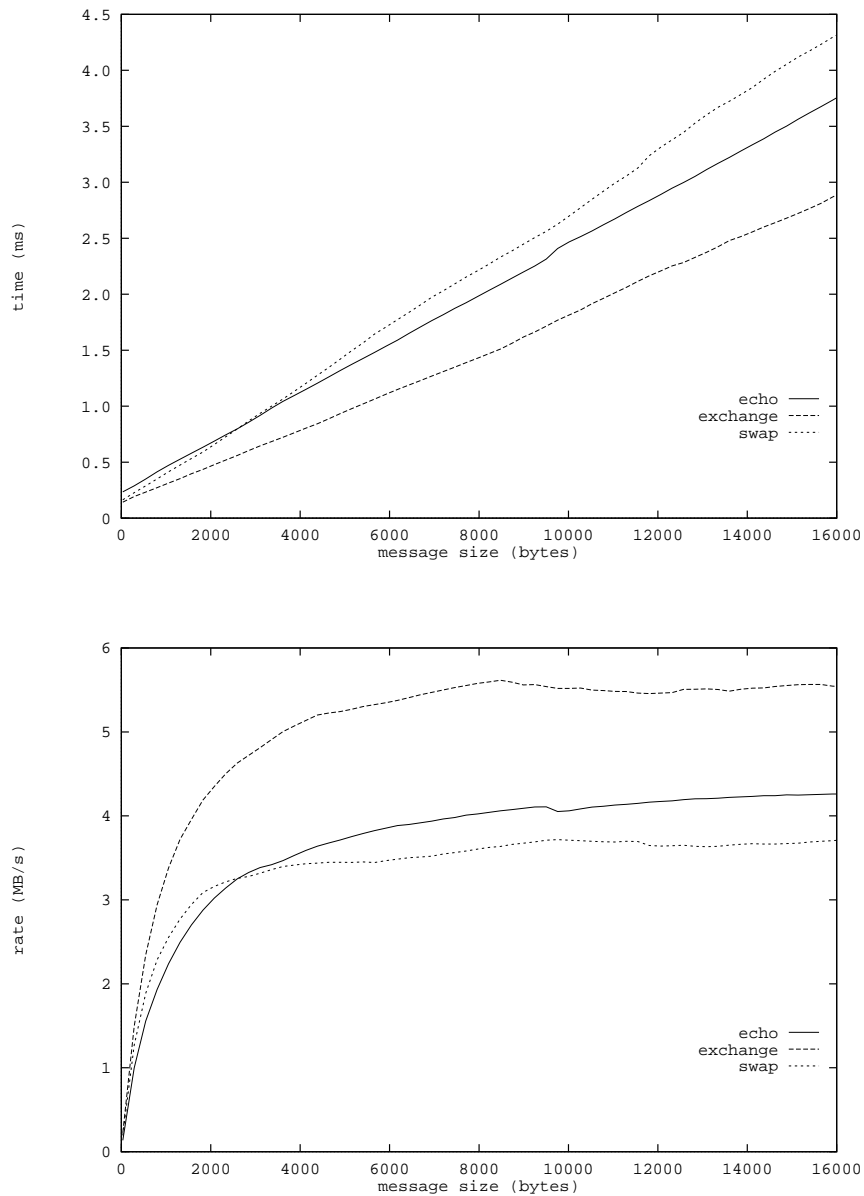


Figure 16: Performance of the user-constructed circular shift operation.

## 8 Summary

With the current version of the software, the bandwidth for passing message with simple send and receive functions is about 9.4 MB/s, and about 11.9 MB/s for simultaneous send and receive in both directions (using the `swap`). These speeds are observed only when the message length is a multiple of 4 bytes. If the message size is not a multiple of 4 bytes, these bandwidths are only 3.6 MB/s and 5.3 MB/s respectively. The start-up time for the simple send and receive pair is about 119  $\mu$ s. The start-up time for the simultaneous send and receive functions are slightly higher. For message size of multiple of 4 bytes, the start-up time for the `send_and_receive` is 159  $\mu$ s, while the start-up time for the `swap` operation is 192  $\mu$ s. The start-up times are smaller for the cases where the message sizes are not multiple of 4 bytes.

The bandwidth is not affected very significantly by the distance between the two nodes involved in the message exchange. From table 3 we can see that when there are only one or two messages in the network, the performance of the cooperative message passing functions is almost independent of the distance.

The broadcast operation can sustain a rate of about 2.2 MB/s from the host to every nodes in the partition, with a start up time of only about 23 $\mu$ s. The scatter operation can operate at the same rate as that of the broadcast. The gather operation can collect messages at a rate of about 1.2 MB/s from all the nodes to the host. Data transposition can achieve the rate of 0.94 MB/s. The start-up time for all four operations are small. Except for the broadcast, there is a serious performance degradation when the message length to or from the nodes are not a multiples of 4 bytes (per node).

The vector versions of the node-to-node communication functions have a very similar characteristic as those of their sequential counterparts. There is a strong evidence indicating that as the vector stride increases, the communication bandwidth decreases.

Another test made here is related to the starting position of the message buffer. Once again, the message size is an important factor. For a message size that is a multiple of 4 bytes, the dependence on the starting point of the message buffer is much stronger than for the message size that is not a multiple of 4 bytes. In most cases the word-aligned buffer can achieve a good performance. There are cases where the double-word-aligned buffer is better.

On the whole the communication primitives have a quite predictable behavior. Generally speaking, the global communication functions of the CM-5 have relatively low latency, high bandwidth (see [2]), and weak dependency on the distance(table 3). Even under high demand, as in the example of the circular shift where every node is sending data to other nodes, the actual communication rate can reach the same level as only one or two messages are exchanged in the network (see table 6).

For an application choosing the most appropriate type of communication function is crucial. For example, to perform a circular shift, the general exchange function is considerably faster than the others we tested. To exchange messages between two nodes,

the `swap` function can perform at higher speed.

## References

- [1] Z. Bozkus, S. Ranka and G. Fox, *Benchmarking the CM-5 multicomputer*, Page 100, Proceeding of Frontiers in Massively Parallel Computing '92.
- [2] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauer, *Active Messages: a Mechanism for Integrated Communication and Computation*, Proceedings of the 19th International Symposium on Computer Architecture, ACM Press, May 1992, Gold Coast, Australia.
- [3] M. Lin, R. Tsang, D. Du, A. Klietz, S. Saroff, *Performance Analysis of the CM-5 Interconnection Network*, Proceedings of CompCon 1993.
- [4] R. Ponnusamy, A Choudhary and G. Fox, *Communication Overhead on CM5: An Experimental Performance Evaluation*, Page 108, Proceeding of Frontiers in Massively Parallel Computing '92.
- [5] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, October 1991.
- [6] Thinking Machines Corporation, *Programming the NI Version 7.1*, March 1992.
- [7] Thinking Machines Corporation, *CMMD reference Manual Version 2.0 Beta*, August 1992.
- [8] Thinking Machines Corporation, *CMMD User's Guide Version 2.0 Beta*, September 1992.