5-2000

MPI Collective Operations over IP Multicast

Amy Apon
Clemson University, aapon@clemson.edu

H A. Chen
University of Arkansas - Main Campus

Y O. Carrasco
University of Arkansas - Main Campus

MPI Collective Operations over IP Multicast [*]

Hsiang Ann Chen, Yvette O. Carrasco, and Amy W. Apon

Computer Science and Computer Engineering
University of Arkansas
Fayetteville, Arkansas, U.S.A
{hachen,yochoa,aapon}@comp.uark.edu

Abstract. Many common implementations of Message Passing Interface (MPI) implement collective operations over point-to-point operations. This work examines IP multicast as a framework for collective operations. IP multicast is not reliable. If a receiver is not ready when a message is sent via IP multicast, the message is lost. Two techniques for ensuring that a message is not lost due to a slow receiving process are examined. The techniques are implemented and compared experimentally over both a shared and a switched Fast Ethernet. The average performance of collective operations is improved as a function of the number of participating processes and message size for both networks.

1 Introduction

Message passing in a cluster of computers has become one of the most popular paradigms for parallel computing. Message Passing Interface (MPI) has emerged to be the *de facto* standard for message passing. In many common implementations of MPI for clusters, MPI collective operations are implemented over MPI point-to-point operations. Opportunities for optimization remain.

Multicast is a mode of communication where one sender can send to multiple receivers by sending only one copy of the message. With multicast, the message is not duplicated unless it has to travel to different parts of the network through switches. Many networks support broadcast or multicast. For example, shared Ethernet, token bus, token ring, FDDI, and reflective memory all support broadcast at the data link layer.

The Internet Protocol (IP) supports multicast over networks that have IP multicast routing capability at the network layer. The goal of this paper is to investigate the design issues and performance of implementing MPI collective operations using multicast. IP multicast is used to optimize the performance of MPI collective operations, namely the MPI broadcast and MPI barrier synchronization, for this preliminary work. The results are promising and give insight to work that is planned on a low-latency network. The remainder of this paper describes IP multicast, design issues in the implementations, experimental results, conclusions, and future planned work.

[*] This work was supported by Grant #ESS-9996143 from the National Science Foundation

2 IP Multicast

Multicast in IP is a receiver-directed mode of communication. In IP multicast, all the receivers form a group, called an IP multicast group. In order to receive a message a receiving node must explicitly join the group. Radio transmission is an analogy to this receiver-directed mode of communication. A radio station broadcasts the message to one frequency channel. Listeners tune to the specific channel to hear that specific radio station. In contrast, a sender-directed mode of communication is like newspaper delivery. Multiple copies of the paper are delivered door-to-door and the newspaper company must know every individual address of its subscriber. IP multicast works like radio. The sender only needs to send one copy of the message to the multicast group, and it is the receiver who must be aware of its membership in the group.

Membership in an IP multicast group is dynamic. A node can join and leave an IP multicast group freely. A node can send to a multicast group without having to join the multicast group. There is a multicast address associated with each multicast group. IP address ranges from 224.0.0.0 through 239.255.255.255 (class D addresses) are IP multicast addresses. Multicast messages to an IP multicast group will be forwarded by multicast-aware routers or switches to branches with nodes that belong to the IP multicast group. IP multicast saves network bandwidth because it reduces the need for the sender to send extra copies of its message and therefore lowers the latency of the network.

In theory, IP multicast should be widely applicable to reduce latency. However, one drawback of IP multicast is that it is unreliable. The reliable Transmission Control Protocol(TCP) does not provide multicast communication services. The User Datagram Protocol (UDP) is used instead to implement IP multicast applications. UDP is a “best effort” protocol that does not guarantee datagram delivery. This unreliability limits the application of IP multicast as a protocol for parallel computing.

There are three kinds of unreliability problems with implementing parallel collective operations over IP multicast. One comes with unreliability at the hardware or data link layer. An unreliable network may drop packets, or deliver corrupted data. In this work, we assume that the hardware is reliable and that packets are delivered reliably at the data link layer. It is also possible that a set of fast senders may overrun a single receiver. In our experimental environment we have not observed these kind of errors. However, a third problem is related to the software design mismatch between IP multicast and parallel computing libraries such as MPI. In WAN’s, where IP multicast is generally applied, receivers of a multicast group come and go dynamically, so there is no guarantee of delivery to all receivers. The sender simply does not know who the receivers are. However, in parallel computing all receivers must receive.

With IP multicast, only receivers that are ready at the time the message arrives will receive it. However, the asynchronous nature of cluster computing makes it impossible for the sender know the receive status of the receiver without some synchronizing mechanism, regardless of how reliable the underlying hardware is. This is a paradigm mismatch between IP multicast and MPI. This paper explores two synchronizing techniques to ensure that messages are not lost because a receiving process is slower than the sender.

This work is related to other efforts to combine parallel programming and broadcast or multicast messaging. In work done on the Orca project [8], a technique was developed for ensuring the reliability of a broadcast message that uses a special sequencer node. In research done at Oak Ridge National Laboratory, parallel collective operations in Parallel Virtual Machine (PVM) were implemented over IP multicast[2]. In that work, reliability was ensured by the sender repeatedly sending the same message until ack's were received from all receivers. This approach did not produce improvement in performance. One reason for the lack of performance gain is that the multiple sends of the data cause extra delay.

The goal of this work is to improve the performance of MPI collective calls. This work focuses on the use of IP multicast in a cluster environment. We evaluate the effectiveness of constructing MPI collective operations, specifically broadcast and barrier, over IP multicast in a commodity-off-the-shelf cluster.

3 MPI Collective Operations

The Message Passing Interface (MPI) standard specifies a set of collective operations that allows one-to-many, many-to-one, or many-to-many communication modes. MPI implementations, including LAM[6] and MPICH[7], generally implement MPI collective operations on top of MPI point-to-point operations. We use MPICH as our reference MPI implementation.

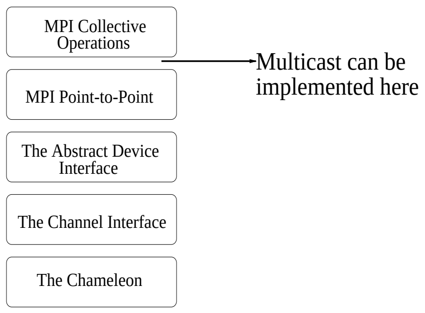
MPI Collective Operations
MPI Point-to-Point
The Abstract Device Interface
The Channel Interface
The Chameleon
Multicast can be implemented here

Fig. 1. MPICH Layers

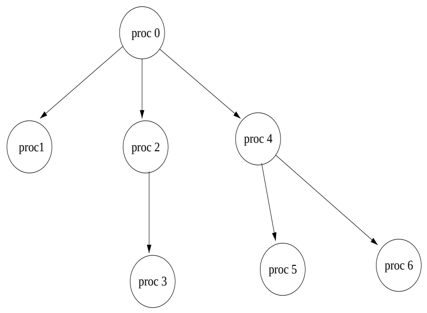

Fig. 2. MPICH Broadcast mechanism with 4 nodes

MPICH[3] uses a layered approach to implement MPI. The MPICH layers include the Abstract Device Interface (ADI) layer, the Channel Interface Layer, and the Chameleon layer. Portability is achieved from the design of the ADI layer, which is hardware dependent. The ADI provides an interface to higher layers that are hardware independent. The MPICH point-to-point operations are built on top of the ADI layer. To avoid implementing collective operations over MPICH point-to-point functions, the new implementation has to bypass all the MPICH layers, as shown in Fig. 1.

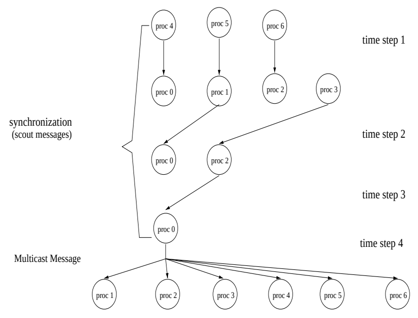

Fig. 3. MPI broadcast using IP multicast (Binary Algorithm)

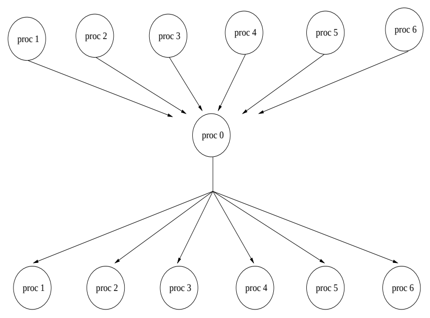

Fig. 4. MPI broadcast using IP multicast (Linear Algorithm)

3.1 MPI Broadcast

Since the new layer for MPI collective operations using multicast is compared experimentally with the original MPICH implementation, it is helpful to understand how these functions are implemented in MPICH. MPICH uses a tree structured algorithm in its implementation of MPI broadcast operation (MPI_Bcast). In the broadcast algorithm, the sender sends separate copies of the message to some of the receivers. After they receive, the receivers at this level in turn send separate copies of the message to receivers at the next level. For example, as illustrated in Fig. 2, in an environment with 7 participating processes, process 0 (the root) sends the message to processes 4, 2, and 1. Process 2 sends to process 3 and process 4 sends to processes 5 and 6. In general, if there are N participating processes, the message size is M bytes and the maximum network frame size is T bytes, it takes $(\frac{M}{T}+1)\times(N-1)$ network frames for one broadcast.

When IP multicast is used to re-implement MPI broadcast, the software must ensure that all receivers have a chance to receive. Two synchronization mechanisms have been implemented, a binary tree algorithm and a linear algorithm. In the binary tree algorithm, the sender gathers small scout messages with no data from all receivers in a binary tree fashion before it sends. With K processes each executing on a separate computer, the height of the binary tree is log_2K+1. In the synchronization stage at time step 1, all processes at the leaves of binary tree send. Scout messages propagate up the binary tree until all the messages are finally received at the root of the broadcast. After that, the root broadcasts the data to all processes via a single send using IP multicast. For example, as illustrated in Fig. 3 in an environment with 7 participating processes, processes 4, 5, and 6 send to processes 0, 1, and 2, respectively. Next, process 1 and process 3 send to processes 0 and 2, respectively. Then process 2 sends to process

0. Finally, process 0 sends the message to all processes using IP multicast. In general, with N processes, a total of $N-1$ scout messages are sent. With a message size of M, and a maximum network frame size of T, $\frac{M}{T}+1$ network frames need to be sent to complete one message transmission. Adding the $N-1$ scout messages, it takes a total of $(N-1)+\frac{M}{T}+1$ frames to send one broadcast message.

The linear algorithm makes the sender wait for scout messages from all receivers, as illustrated in Fig. 4. Then the message with data is sent via multicast. With K processes in the environment, it takes $K-1$ steps for the root to receive all the scout messages since the root can only receive one message at a time. As illustrated in Fig. 4 with N processes, the root receives $N-1$ point-to-point scout messages before it sends the data. With 7 nodes, the multicast implementation only requires one-third of actual data frames compared to current MPICH implementation. Since the binary tree algorithm takes less time steps to complete, we anticipate it to perform better than the linear algorithm.

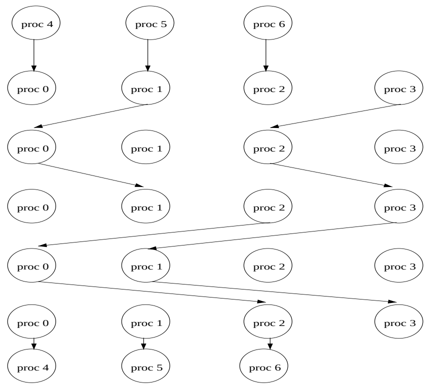

Fig. 5. MPICH barrier synchronization with 7 processes

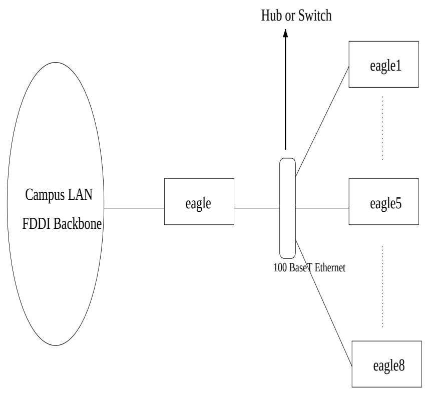

Fig. 6. The Eagle Cluster

3.2 MPI Barrier Synchronization

Another MPI collective operation re-implemented was MPI_Barrier. MPI_Barrier is an operation that synchronizes processes. All processes come to a common stopping point before proceeding. The MPICH algorithm for barrier synchronization can be divided into three phases. In the first phase, processes that cannot be included in sending pair-wise point-to-point operations send messages to processes who can. In the second phase, point-to-point sends and receives are performed in pairs. In the third phase, messages are sent from the processes in the second phase to processes from the third phase to release them. Figure 5 illustrates MPICH send and receive messages for synchronization between 7 processes. In this example, processes 4, 5, and 6 send messages to processes 0, 1 and 2. In the second phase, point-to-point message are sent between processes 0, 1, 2, and 3. In the third phase, process 0, 1, and 2, send messages to 4, 5, and 6 to release them. If there are N participating processes, and K is the biggest power of 2 less than N, a total of $2 \times (N - K) + log_2 K \times K$ messages need to be sent.

By incorporating IP multicast into the barrier algorithm, we were able to reduce the number of phases by two. The binary algorithm described above is used to implement MPI_Barrier. First, point-to-point messages are reduced to process 0 in a binary tree fashion. After that, a message with no data is sent using multicast to release all processes from the barrier. In general, with N processes in the system, a total of $N - 1$ point-to-point messages are sent. One multicast message with no data is sent.

4 Experimental Results

The platform for this experiment consists of four Compaq PentiumIII 500MHZ computers and five Gateway PentiumIII 450 MHZ computers. The nine workstations are connected via either a 3Com SuperStack II Ethernet Hub or an HP ProCurve Switch. Both the hub and the switch provide 100 Mbps connectivity. The switch is a managed switch that supports IP multicast. Each Compaq workstation is equipped with 256 MB of memory and an EtherExpress Pro 10/100 Ethernet card. Each Gateway computer has 128MB of memory and a 3Com 10/100 Ethernet card.

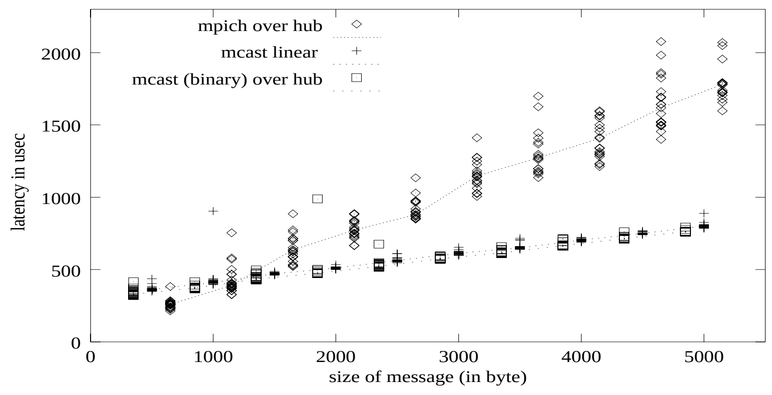

Fig. 7. MPI_Bcast with 4 processes over Fast Ethernet Hub

The performance of the MPI collective operations is measured as the longest completion time of the collective operation. among all processes. For each message size, 20 to 30 different experiments were run. The graphs show the measured

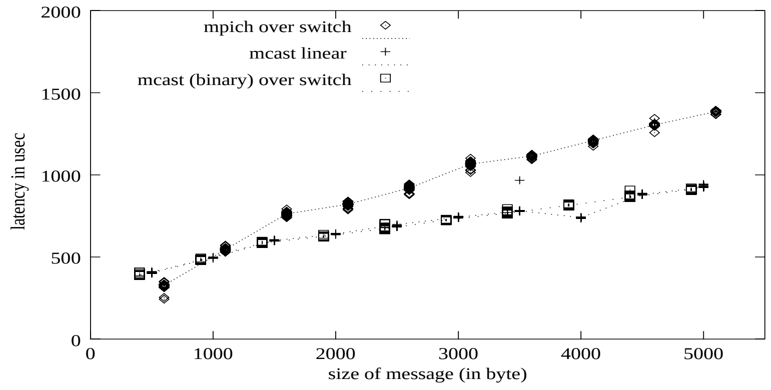

Fig. 8. MPI_Bcast with 4 processes over Fast Ethernet Switch

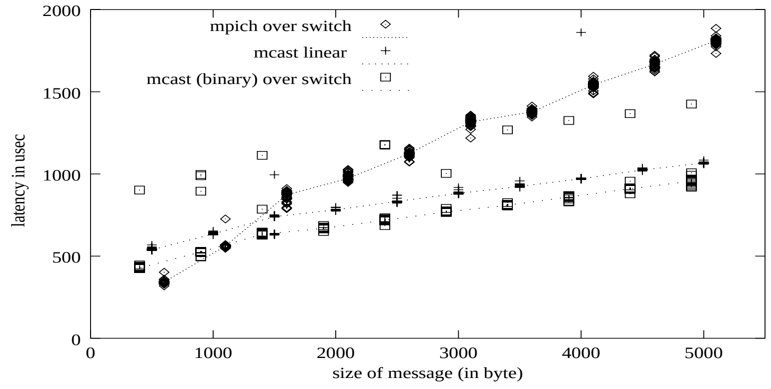

Fig. 9. MPI_Bcast with 6 processes over Fast Ethernet Switch

time for all experiments with a line through the median of the times. The graphs illustrate the sample distribution of measured times.

Figure 7 shows the performance of MPI_Bcast of both implementations over the hub with 4 processes. The figure shows that the average performance for both the linear and the binary multicast implementation is better for message sizes greater than 1000 bytes. With small messages, the cost of the scout messages causes the multicast performance to be worse than MPICH performance. The figure also shows variations in performance for all implementations due to collisions on the Fast Ethernet network. The variation in performance for MPICH is generally higher than the variation in performance for either multicast implementation.

Figures 8, 9,and 10 describe the performance with the switch for 4, 6, and 9 processes respectively. Both the linear and the binary algorithm using multicast show better average performance for a large enough message size. The crossover point of average MPICH performance and the average performance of using

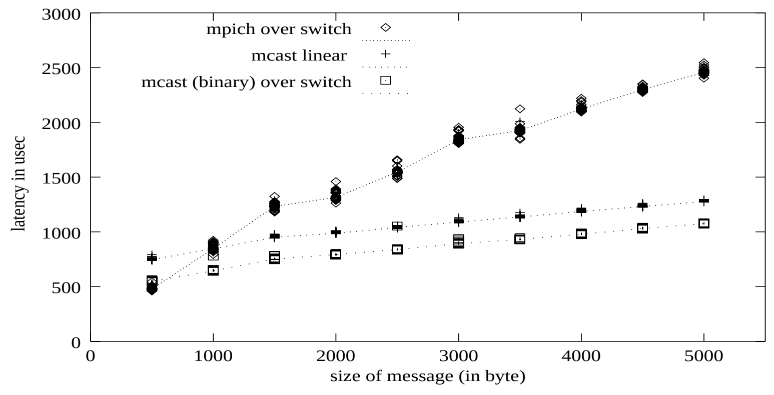

Fig. 10. MPI_Bcast with 9 processes over Fast Ethernet Switch

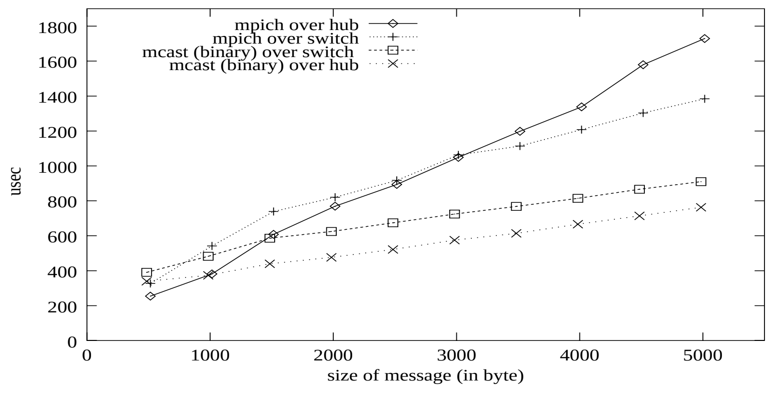

Fig. 11. Performance Comparison with MPI_Bcast over hub and switch for 4 processes

multicast is where the extra latency of sending scout messages becomes less than the latency from sending extra packets of data when the data is large. For some numbers of nodes, collisions also caused larger variance in performance with the multicast implementations. For example, this is observed for 6 nodes as shown in Fig. 9. With 6 nodes using the binary algorithm, both node 2 and node 1 attempt to send to node 0 at the same time, which causes extra delay.

Figure 11 compares the average performance of the switch and the hub for 4 processes. When using IP multicast, the average performance of the hub is better than the switch for all measured message sizes. As for the original MPICH implementation, the average performance of hub becomes worse than the switch when the size of the message is bigger than 3000. The MPICH implementation puts more messages into the network. As the load of the network gets larger, the extra latency of the switch become less significant than the improvement gained with more bandwidth. The multicast implementation is better than MPICH for message sizes greater than one Ethernet frame.

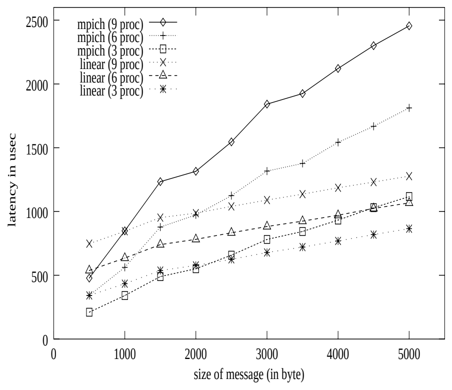

Fig. 12. Performance Comparison with MPI_Bcast over 3, 6, and 9 processes over Fast Ethernet switch

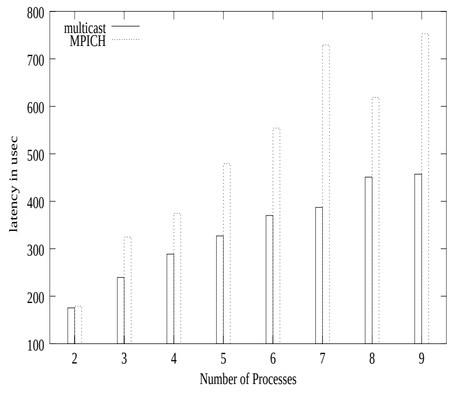

Fig. 13. Comparison of MPI_Barrier over Fast Ethernet hub

Figure 12 compares MPICH and the linear multicast implementation for 3, 6, and 9 processes over the switch. The results show that the linear multicast algorithm scales well up to 9 processes and better than MPICH. With the linear implementation, the extra cost for additional processes is nearly constant with respect to message size. This is not true for MPICH.

Figure 13 describes the results of MPI_Barrier operation over the hub. The results for MPI_Barrier show that IP multicast performs better on the average than the original MPICH implementation. The performance improvement increases as the size of the message gets bigger.

In a Single Program Multiple Data (SPMD) environment, message passing using either the linear algorithm or the binary algorithm is correct even when there are multiple multicast groups. However, since the IP multicast implementation requires the receive call to be posted before the message is sent, it is required that each process execute the multicast calls in the same order. This restriction is equivalent to requiring that the MPI code be safe[5]. If several processes broadcast to the same multicast group (in MPI terms, this is the same process group of same context), the order of broadcast will be correctly preserved. For example, suppose in an environment including the 4 processes with ids 4, 6, 7 and 8, processes 6, 7, and 8 all belong to the same multicast group and the broadcast is called in the following order.

```
MPI_Bcast(&buffer, count, MPI_INT, 6, MPI_COMM_WORLD);
MPI_Bcast(&buffer, count, MPI_INT, 7, MPI_COMM_WORLD);
MPI_Bcast(&buffer, count, MPI_INT, 8, MPI_COMM_WORLD);
```

Using either the binary algorithm or the linear algorithm, process 7 cannot proceed to send the the second broadcast until it has received the broadcast message from process 6, and process 8 cannot send in the third broadcast until it has received the broadcast message from process 7. The order of the three broadcasts is carried out correctly. Using a similar argument, when there are two or more multicast groups that a process receives from, the order of broadcast will be correct as long as the MPI code is safe.

5 Conclusions and Future Work

Multicast reduces the number of messages required and improves the performance of MPI collective operations by doing so. Its receiver-directed message passing mode allows the sender to address all the receivers as a group. This experiment focused on a particular implementation using IP multicast.

Future work is planned in several areas. Improvements are possible to the binary tree and linear communication patterns. While we have not observed buffer overflow due to a set of fast senders overrunning a single receiver, it is possible this may occur in many-to-many communications and needs to be examined further. Additional experimentation using parallel applications is planned. Also, low latency protocols such as the Virtual Interface Architecture[9] standard typically require a receive descriptor to be posted before a mesage arrives. This is similar to the requirement in IP multicast that the receiver be ready. Future work is planned to examine how multicast may be applied to MPI collective operations in combination with low latency protocols.

References

[1] D. E. Comer. *Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture* . Prentice Hall, 1995.

[2] T. H. Dunigan and K. A. Hall. PVM and IP Multicast. Technical Report ORNL/TM-13030, Oak Ridge National Laboratory, 1996.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical Report Preprint MCS-P567-0296, Argonne National Laboratory, March 1996.

[4] N. Nupairoj and L. M. Ni. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In *Proceedings of the 1994 Scalable Parallel Libraties Conference*, pages 98–105. IEEE Computer Society Press, October 1994.

[5] P. Pacheo. *Parallel Programming with MPI* . Morgan Kaufmann, 1997.

[6] The LAM source code. `http://www.mpi.nd.edu/lam`.

[7] The MPICH source code. `www-unix.mcs.anl.gov/mpi/index.html`.

[8] A. S. Tannenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, 25(8), 1992.

[9] The Virtual Interface Architecture Standard. `http://www.viarch.org`.

[10] D. Towsley, J. Kurose, and S. Pingali. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *IEEE JSAC*, 15(3), April 1997.