Group Management Schemes for Implementing MPI Collective Communication over IP–Multicast

Xin Yuan Scott Daniels Ahmad Faraj Amit Karwande
Department of Computer Science, Florida State University, Tallahassee, FL 32306
{xyuan,sdaniels,faraj,karwande}@cs.fsu.edu

Abstract—

Recent advances in multicasting present new opportunities for improving communication performance for clusters of workstations. Realizing collective communication over multicast primitives can achieve higher performance than over unicast primitives. However, implementing collective communication using multicast primitives presents new issues and challenges. Group management, which may result in large overheads, is one of the important issues. In this paper, we propose three group management schemes that can be used to implement MPI collective communication routines, static group management, dynamic group management, and compiler–assisted group management. We implement the three schemes in a prototype MPI library and evaluate the performance of the schemes. We further compare our multicast based implementation with an efficient unicast based MPI implementation, LAM/MPI. The results show that multicasting can greatly improve the performance of collective communication in many situations. In particular, combining multicasting with compiler–assisted group management can deliver very high performance.

I. Introduction

As microprocessors become more and more powerful, clusters of workstations have become one of the most common high performance computing environments. Many institutions have Ethernet–connected clusters that can be used to perform high performance computing. One of the key building blocks for such systems is the message passing library. Standard message passing libraries, including Message Passing Interface (MPI) [6] and Parallel Virtual Machine (PVM) [7], have been implemented for such systems. Current implementations, such as MPICH[3] and LAM/MPI[8], focus on providing the functionality, that is, moving data across processors, and addressing the portability issues. To achieve interoperability, these implementations are built over unicast primitives supported by the TCP/IP protocol suite. However, studies have shown that the current implementations of message passing libraries for networks of workstations are not tailored to achieve high communication performance over clusters of workstations [1].

Recent advances in multicasting over the Internet present new opportunities for improving communication performance for clusters of workstations without sacrificing the portability of the message passing libraries. Specifically, coupling local area network (LAN) hardware, which supports broadcast (multicast) communication, with IP–multicast [2], multicast communication is now supported by the TCP/IP protocol suite and can be utilized in commodity LAN environments without any modification of the hardware and the operating system. Using multicast primitives to realize collective communication routines can potentially improve the communication performance significantly since multicasting reduces both message traffic over the network and the CPU processing at the end hosts.

Implementing collective communication routines over multicast primitives, however, introduces new issues and challenges. Some issues to be addressed include designing reliable multicast protocols, group management schemes and message scheduling methods. Our previous work [5] studied efficient reliable multicast protocols in the LAN environment. This paper focuses on the group management issue. Basically, a multicast group must be created before any multicast communication can be performed on that group. A group management scheme determines when to create/destroy a multicast group. Given a set of N processes, the number of potential groups is 2^N. Thus, it is impractical to establish all potential groups for a program and group management must be performed as the program executes. Since the group management operations require the coordination of all members in the group and are expensive, group management must be carefully handled for multicasting to be effective.

In this paper, we study the MPI collective communication routines whose performance can potentially be improved by multicasting and discuss how these routines can be implemented using multicast primitives. We propose three group management schemes, the static group management scheme, the dynamic group management scheme and the compiler–assisted group management scheme. The static group management scheme associates a multicast group with each *communicator*, which is an MPI entity to identify the group of participating processes. This scheme is ideal for collective communications, such as broadcast, that involve all nodes in a communicator. The static group management scheme does not support one–to–many communication effectively. The dynamic group management scheme creates/destroys multicast groups dynamically and allows multicasting to be performed only on the nodes that are interested in the communication. This scheme is flexible, but incurs large group management overheads. The compiler–assisted group management scheme takes advantages of the fact that most of the communications in parallel programs are static, that is, can be determined at the compile time [4]. This is a form of the compiled communication technique [10]. The communication library provides routines to allow the compiler to determine when to create/destroy groups and to allow the data movement to be performed more efficiently since more restricted assumptions can be made in the implementation of the data movement routines under the compiled communication model. Like the static group management scheme, the compiler–assisted group management scheme allows the group management overheads to be amor-

This project was supported in part by NSF grants: CCR-9904943, CCR-0073482 and ANI-0106706

tized over multiple communications.

We implement the three group management schemes in our prototype MPI library where a number of collective communication routines are built over IP–multicast using the standard UDP interface. We evaluate the performance of the schemes using the prototype library. We further compare the performance of our multicast based implementation with that of an efficient unicast based MPI implementation, LAM/MPI[8]. We conclude that multicasting can significantly improve the performance of collective communication in many cases and that multicasting combined with compiler–assisted group management can deliver very high performance. The rest of the paper is structured as follows. Section II briefly describes MPI collective communication routines. Section III presents the three group management schemes. Section IV reports the results of our performance study. Section V concludes the paper.

II. MPI COLLECTIVE COMMUNICATION ROUTINES

MPI is a library specification for message–passing, proposed as a standard by a broadly based committee of vendors, implementors and users. A set of standard collective communication routines is defined in MPI. Each collective communication routine has a parameter called *communicator*, which identifies the group of participating processes. MPI collective communication routines include the followings.

- Barrier. The routine *MPI_Barrier* blocks the callers until all members in the communicator call the routine.
- Broadcast. The routine *MPI_Bcast* broadcasts a message from a process called *root* to all processes in the communicator.
- Gather. The routines *MPI_Gather* and *MPI_Gatherv* allow each process in the communicator to send data to one process.
- Scatter. The routine *MPI_Scatter* allows one process to send a different message to each other process. The message size is the same for all processes when using this routine. A more general form of scatter, where messages can be of different sizes, is realized by the routine *MPI_Scatterv*. *MPI_Scatterv* can also perform one–to–many communications within the communicator by carefully selecting the input parameters.
- Gather–to–all. The routine *MPI_Allgather* gathers the information from all processes and puts the results to all processes. Another routine, *MPI_Allgatherv*, allows the data sizes for different processes to be different.
- All–to–all. The routine *MPI_Alltoall* is a generalization of gather–to–all in that different messages can be sent to different processes. The most general form of all–to–all communication is the *MPI_Alltoallw* routine in the MPI 1–2 specification, which allows the general many–to–many or one–to–many communications to be performed by carefully selecting the input arguments.
- Global reduction. These functions perform global reduce operations (such as sum, max, logical AND, etc.) across all the members of a group.
- Reduce–scatter. The routine *MPI_Reduce_scatter* performs the global reduce operation and makes the result appear in all the processes.
- Scan. Similar to the reduce function, but performs a prefix reduction on data distributed across the group.

In order to use multicasting to improve performance of a communication routine, the routine must contain one–to–many or one–to–all communication patterns. Among the collective communication routines defined in MPI, the gather, reduce and scan operations do not have one–to–many or one–to–all communications. We will not consider these routines in this paper. Other routines can potentially benefit from multicasting since multicasting distributes information to multiple destinations more efficiently. Four types of collective communications, one–to–all communication, one–to–many communication, all–to–all communication and many–to–many communication, can benefit from a multicast based implementation. Here, the word 'all' means all processes in a communicator. The one–to–all communication routines include *MPI_Bcast*, *MPI_Scatter* and *MPI_Reduce_Scatter*. The one–to–many communication can be realized by *MPI_Scatterv*. The all–to–all communication routines include *MPI_Allgather*, *MPI_Alltoall* and *MPI_Allreduce*. The many–to–many communication routines include *MPI_Alltoallw*.

III. GROUP MANAGEMENT SCHEMES FOR A MULTICAST BASED IMPLEMENTATION

Group management determines when to create/destroy a multicast group. Its functionality is similar to the management of communication end points in unicast communication. Using unicast communication, for N processes, the number of communication end points per process is $O(1)$ for UDP unicast communication, and $O(N)$ to establish direct TCP connections to all other processes. Since the number of communication end points is small, the communication end points can be opened at the system initialization and closed at the system finalization. Using multicast communication, however, the number of potential groups is 2^N. It is, thus, impossible to establish all groups simultaneously and group management must be performed on the fly as the program executes.

In this section, we will describe three group management schemes that can be used to implement MPI collective communications, the static group management scheme, the dynamic group management scheme, and the compiler–assisted group management scheme. We will also discuss how the collective communication routines can be implemented using multicasting. Since all–to–all communication and many–to–many communication are typically built on top of one–to–all and one–to–many communications, we will focus on one–to–all and one–to–many communications in this paper. We will use *MPI_Bcast* and *MPI_Scatter* as representative routines for one–to–all communication, and *MPI_Scatterv* as the representative routine for one–to–many communication.

The static group management scheme associates a multicast group with each communicator. The multicast group is created/destroyed when the communicator is created/destroyed. Since a communicator in a program is usually long–lived (used by multiple communications), the static group management scheme can amortize the group management overheads and the group management overhead is usually negligible. This scheme is ideal for one–to–all communications.

Using the static group management scheme, *MPI_Bcast* can be implemented by having the root send a reliable broadcast message to the group. This implementation does not introduce any extra overheads.

A multicast based *MPI_Scatter* is a little more complicated. In the scatter operation, different messages are sent to different receivers. To utilize the multicast mechanism, the messages to different receivers must be aggregated to send to all receivers. For example, if messages $m1$, $m2$ and $m3$ are to be sent to processes $p1$, $p2$ and $p3$, the aggregate message containing $m1$, $m2$ and $m3$ will be sent to all three processes as one multicast message. Once a process receives the aggregated multicast message, it can identify its portion of the message (since the message sizes to all receivers are the same) and copy the portion to the user space. In comparison to the unicast based *MPI_Scatter* in LAM/MPI, where the sender loops through the receivers sending a unicast message to each of the receivers, the multicast based implementation increases the CPU processing in each receiver since each receiver must now process a larger aggregated message, but decreases the CPU processing in the sender (root) as fewer system calls are needed. Since the bottleneck of the unicast implementation of *MPI_Scatter* is at the sender side, it can be expected that the multicast based implementation can offer a better performance when the aggregated message size is not very large. When the size of the aggregated message is too large, the multicast based implementation will have negative impacts on the performance since it slows down the receivers.

Realizing *MPI_Scatterv* is similar to realizing *MPI_Scatter*. But there are some complications. In *MPI_Scatterv*, different receivers can receive different sized messages and each receiver only knows its own message size. The receivers do not know the size and the layout of the aggregated message. We resolve this problem by using two broadcasts in this function. The first broadcast tells all processes in the communicator the amount of data that each process will receive. Based on this information, each process can compute the memory layout and the size of the aggregated message. The second broadcast sends the aggregate message. *MPI_Scatterv* can realize one–to–many communication by having some receivers not receive any data. Using the static group management scheme, the one–to–many communication is converted into an one–to–all communication since all processes in the communicator must receive the aggregated message. This is undesired since it keeps the processes that are not interested in the communication busy. In addition, this implementation sends a (reliable) multicast message to a group that is larger than needed, which decreases the communication performance. The dynamic group management scheme and the compiler–assisted group management scheme overcomes this problem.

The dynamic group management scheme creates a multicast group when it is needed. This group management scheme is built on top of the static group management scheme in an attempt to improve the performance for one–to–many communications. To effectively realize one–to–many communication, the dynamic group management scheme dynamically creates a multicast group, performs the communication, and destroys the group. In *MPI_Scatterv*, only the sender (root) knows the group of receivers. To dynamically create the group, a broadcast is performed using the static group associated with the communicator to inform all members in the communicator the nodes that should be in the new group. After this broadcast, a new group can be formed and the uninterested processes can move on. After the communication is performed, the group is destroyed. Using the dynamic group management scheme, the uninterested processes will only involve in group creation, but not data movement. Dynamically group management introduces group management overheads for each communication and may not be efficient for sending small messages.

```
(1)  DOi = 1, 1000
(2)      MPI_Scatterv(....)
```

(a) An example program

```
(1)  MPI_Scatterv_open_group(...)
(2)  DOi = 1, 1000
(3)      MPI_Scatterv_data_movement(....)
(4)  MPI_Scatterv_close_group(...)
```

(b) The compiler–assisted group management scheme

Fig. 1. An example for the compiler–assisted group management

The compiler–assisted group management scheme allows the compiler to perform group management. As shown in [4], most communication in parallel programs can be determined by the compiler. Thus, in most situations, the compiler knows the appropriate points to create and destroy a group. This scheme is not implemented in the library. Instead, we extend the MPI interface to provide routines to allow the compiler to manage the groups. For *MPI_Scatterv*, we provide three functions, *MPI_Scatterv_open_group*, *MPI_Scatterv_data_movement*, and *MPI_Scatterv_close_group*. *MPI_Scatterv_open_group* takes exactly the same parameters as *MPI_Scatterv*, creates a new group for the participating processes in the one–to–many communication, and initializes related data structures. *MPI_Scatterv_close_group* destroys the group created by *MPI_Scatterv_open_group*. *MPI_Scatterv_data_movement* performs the data movement assuming that the group is created and that the related information is known to all participated parties. Notice that *MPI_Scatterv_data_movement* is simpler than *MPI_Scatterv*. Since more information related to the communication, including the message size for each process, is known to all processes, only one broadcast is needed to implement *MPI_Scatterv_data_movement* instead of two in *MPI_Scatterv*.

Fig. 1 shows an example of the compiler–assisted group management scheme. Fig. 1 (a) is an example program that performs *MPI_Scatterv* in a loop. If the compiler can determine the group information used by the *MPI_Scatterv* routine, it can perform group management as shown in Fig. 1 (b). Two factors can contribute to the performance improvement over the dynamic group management scheme. First, the group management overhead is amortized among all the communications in the loop. Second, the data movement is efficient.

IV. Performance Study

We implement a prototype MPI library where most collective communication routines are realized using multicasting. The three proposed group management schemes are implemented in the prototype. In this section, we compare the performance of our prototype library with that of LAM/MPI, an efficient unicast–based MPI implementation, and evaluate the performance of the three group management schemes.

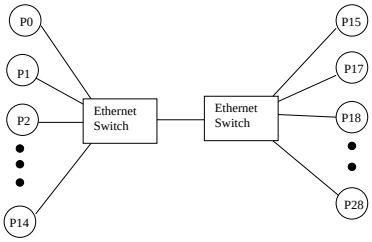

Fig. 2. Performance evaluation environment

The performance is evaluated in a cluster with 29 Pentium III–650MHz processors. Each machine has 128MB memory and 100Mbps Ethernet connection via a 3Com 3C905 PCI EtherLink Card. All machines run RedHat Linux version 6.2, with 2.4.7 kernel. The machines are connected by two 3Com SuperStack II baseline 10/100 Ethernet switches as shown in Fig. 2. To obtain accurate experimental results, we measure the communication time three times for each experiment and report the average of the three measurements. We use LAM version 6.5.4 with the direct client to client mode communication for the comparison. Since collective communication involves multiple nodes, if not specified otherwise, we will use the *average* time among all nodes for a given collective communication as the performance metric.

TABLE I
Message round-trip time (ping-pong)

Msg size (bytes)	LAM/MPI	Our MPI
1	0.13ms	0.23ms
10	0.13ms	0.22ms
100	0.17ms	0.24ms
1000	0.53ms	0.60ms
10000	2.17ms	2.38ms
100000	18.92ms	22.43ms
1000000	205.10ms	224.38ms

Table I shows the baseline comparison of our implementation and the LAM/MPI implementation. This table compares the performance of the point–to–point communication by measuring the ping-pong style communication, that is, two processes sending messages of a given size to each other. LAM/MPI is based on TCP and the communication reliability is provided by the operating system. Our implementation is based on UDP and the communication reliability is enforced in the user domain, that is, acknowledgement packets are sent by the user. Hence, our implementation incurs more switchings between the user mode and the system mode. This impact is significant when the message size is small. When the message size is large, the UDP based implementation may have advantage since TCP must go through the slow start to reach the maximum throughput of the network [9], while the UDP based implementation can directly send large packets to fully utilize the network bandwidth. The maximum UDP packet size is $16KB$ in our implementation. Table I shows the overall effect. For point–to–point communication with small sized messages ($< 1KB$), LAM/MPI is about 75% more efficient. For large sized messages ($> 1KB$), LAM/MPI is about 10% more efficient.

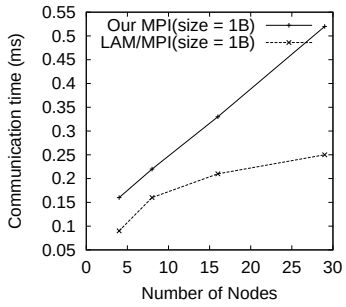

Fig. 3. Performance of *MPI_Bcast* (message size = 1B)

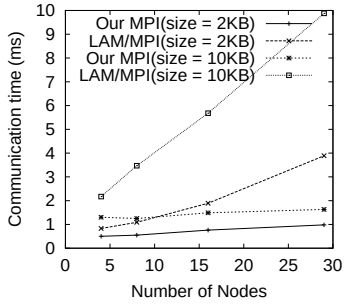

Fig. 4. Performance of *MPI_Bcast* (large message sizes)

Figures 3 and 4 show the performance of *MPI_Bcast*. As can be seen from Fig. 3, multicasting does not guarantee to improve communication performance even for the broadcast communication. The reason that LAM/MPI broadcast implementation is more efficient than our multicast based implementation when message size is 1 byte is that LAM/MPI has an efficient logical tree based broadcast implementation when the group is larger than 4 processes. This distributes the broadcast workload to multiple nodes in the system. While in our implementation, the root, although sends only one multicast packet, must process all acknowledgement packets from all receivers. As a result, for small sized messages, our multicast based implementation performs worse. However, when the message size is larger, the acknowledgement processing overhead is insignificant and the saving of sending one multicast data packet instead of sending multiple unicast packets dominates. In this case, our multicast based implementation is much more efficient as shown in Fig. 4.

Figures 5 shows the performance of *MPI_Scatter* on a 29 node system. In the scatter operation, the root sends different data to different receivers. This has two implications. For unicast implementation, the root must iterate to send a message to each of the receivers and the sender is the bottleneck. The tree–based

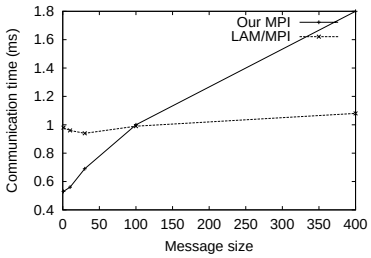

Fig. 5. Performance of *MPI_Scatter* on a 29-node system

implementation used in broadcast cannot be utilized. For multicast implementation, the messages must be aggregated so that each receiver must receive more than what it needs, which decreases the performance. Thus, the multicast based implementation can offer better performance only when the message size is small. As shown in Fig. 5 when message size is less than $100B$, our implementation performs better than LAM/MPI. When the message size is larger than $100B$, LAM/MPI is better.

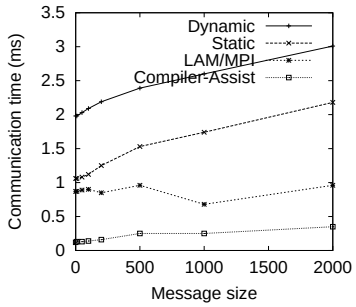

Fig. 6. Performance of one–to–five communication using *MPI_Scatter*

We have evaluated the performance of one–to–all communications, where the static group management scheme is sufficient. The results show that multicasting improves the performance of collective communication in many situations. Next, we will evaluate the performance of one–to–many communication for which dynamic group management and compiler–assisted group management were designed. Fig. 6 shows the performance of *MPI_Scatterv* with different implementations and group management schemes. In this experiment, the root scatters messages of a given size to 5 receivers among the 29 members in the communicator. As can be seen in the figure, the dynamic group management scheme incurs very large overheads and offers the worst performance among all the schemes. The static group management offers better performance. However, since all nodes in the communicator are involved in the collective communication, the average communication time for all nodes is still larger than that provided by LAM/MPI. The compiler–assisted scheme performs the best among all the schemes. We have conducted experiments with different settings, the results demonstrate a similar trend.

It is sometimes more appropriate to compare the maximum communication completion time than the average communication time. Fig. 7 shows the maximum communication completion time. This experiment has the same settings as those in Fig. 6. Using multicast to implement collective communications, processes complete the communication almost simultaneously. The average communication time is close to the maximum communication time. Using unicast to implement collective communications, the communication completion time on different nodes varies significantly since the communication is done sequentially at the end hosts. Comparing Fig. 7 and Fig. 6, we can see that the multicast based implementation performs better in terms of maximum communication completion time.

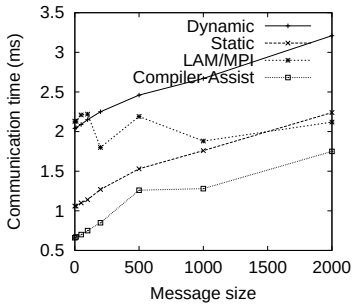

Fig. 7. Maximum communication time for one–to–five communication

We have used the scatter operation in the evaluation of one–to–many communications. For broadcast type one–to–many communications, the multicast–based implementation should give better improvements in comparison to LAM/MPI.

V. Conclusion

In this paper, we present our prototype MPI library where the collective communication routines are implemented based on IP–multicast. Three group management schemes that can be used to implement the collective communication routines are proposed and evaluated. The performance of our prototype MPI library is compared with LAM/MPI, an efficient unicast–based MPI implementation. We conclude that using multicasting can improve the performance of collective communication in many situations. In particular, multicasting combined with compiler–assisted group management offers very high performance.

References

[1] P.H. Carns, W.B. Ligon III, S. P. McMillan and R.B. Ross, "An Evaluation of Message Passing Implementations on Beowulf Workstations", *Proceedings of the 1999 IEEE Aerospace Conference*, March, 1999.

[2] S. Deering, "Multicast Routing in Internetworks and Extended LANs", *ACM SIGCOMM Computer Communication Review*, 1995

[3] William Gropp, E. Lusk, N. Doss and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", *MPI Developers Conference*, 1995.

[4] D. Lahaut and C. Germain, "Static Communications in Parallel Scientific Programs." *Proceedings of PARLE*, 1994.

[5] R. Lane, S. Daniels, X. Yuan, "An Empirical Study of Reliable Multicast Protocols over Ethernet-Connected Networks." *International Conference of Parallel Processing (ICPP)*, pages 553–560, Valencia, Spain, September 3-7, 2001.

[6] The MPI Forum, The MPI-2: Extensions to the Message Passing Interface. *http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*, July, 1997.

[7] R. Manchek, "Design and Implementation of PVM version 3.0", Technique report, University of Tennessee, Knoxville, 1994.

[8] J.M. Squyres, A. Lumsdaine, W.L. George, J.G. Hagedorn and J.E. Devaney "The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI" *MPI Developer's Conference*, Ithica, NY, 2000.

[9] Andrew Tanenbaum, "Computer Networks", 3rd edition, 1996.

[10] Xin Yuan, Rami Melhem and Rajiv Gupta, "Compiled Communication for All–optical Networks." *Supercomputing'96*, Pittsburgh, PA, Nov. 1996.