Efficient Hardware Multicast Group Management for Multiple MPI Communicators over InfiniBand

Amith R Mamidala, Hyun-Wook Jin , Dhabaleswar K Panda

Technical Report
OSU-CISRC-5/05-TR34

Efficient Hardware Multicast Group Management for Multiple MPI Communicators over InfiniBand ⋆

Amith R. Mamidala, Hyun-Wook Jin, and Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
{mamidala, jinhy, panda}@cse.ohio-state.edu

Abstract. MPI provides a set of primitives that allow processes to dynamically create communicators on the fly. This set of primitives can be exploited by the applications where only a certain group of processes need to participate at any given time. Also, these primitives play an important role in the context of dynamic process management of MPI-2. To utilize the hardware multicast of InfiniBand for implementing many collective operations for any aribitrary communicator, we need to pay special attention to the overhead of creation of the communicator. This is because, the construction of communicator in the context of hardware multicast is a complex and costly operation involving interaction with an external multicast management entity. In this paper, we propose different design alternatives of efficiently creating the communicators dynamically. The basic idea behind the schemes proposed is to remove most of the overhead of the hardware multicast group construction from the critical path of the application. Our results indicate that by using Multicast Pool and Lazy approaches of group construction proposed in the paper, we can significantly reduce the overhead of creation of the communicator with hardware multicast support by a factor of as much as 4.8 and 3.9, repectively compared to the Basic approach.
Keywords: MPI, Communicator, Multicast, InfiniBand and Subnet Management

1 Introduction

Message Passing Interface(MPI) [10] programming model has become the de-facto standard to develop parallel applications. MPI provides a rich collection of *point-to-point* and *collective* communication primitives for the application to take advantage of. These primitives are associated with a well defined *Communicator* object in MPI. Communicators provide a mechanism to construct distinct communication spaces for process groups to operate, isolating them from the rest of the communication flow. Also, they encapsulate several internal communication data structures during the program execution.

InfiniBand Architecture (IBA) [6] which is emerging as the next generation interconnect for I/O and interprocessor communication, has several features which directly impact the performance of the application. One of the notable features of InfiniBand is its support for hardware multicast. By using this feature, a message posted to a hardware multicast group is delivered to all the processes attached to this group in

⋆ This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, National Science Foundation's grants #CCR-0204429 and #CCR-0311542, and grants from Intel and Mellanox.

an efficient and scalable manner. We have shown that significant performance can be achieved by leveraging this primitive to implement collective operations like MPI_Bcast, MPI_Barrier and MPI_Allreduce [7] [8]. One primary assumption taken in the above approaches is that all the processes communicate within a single communicator context. Thus, it was suffice to construct a single hardware multicast group statically at the initialization phase serving all the processes.

However, majority of the applications use more than one communicator object during their execution. This is because all the processes may not need to communicate with each other. Also, the creation of a new communicator is imperative in the context of dynamic process management of MPI-2, where new processes can be spawned from an already existing group of processes. To utilize the hardware multicast of InfiniBand, these communicators have to be mapped to hardware multicast groups and this mapping needs to be done on the fly. More importantly, these multicast groups have to be dynamically set up.

In IBA, construction of hardware multicast groups involves a series of management actions. Some of these involve the interaction of the MPI processes with an external IBA multicast management and the rest pertain to the fabric configuration by the multicast management entity. Only after the successful completion of these management actions the multicast group can be used. Depending on the size of the hardware multicast group and the IBA fabric, all these tasks can take considerable amount of time. From the MPI application perspective, the overhead of these operations should be as minimal as possible.

In this paper, we present several ways of constructing hardware multicast groups dynamically. We propose several design alternatives to efficiently map the communicators to these newly created hardware multicast groups. Our designs of using Multicast Pool and Lazy approaches of group construction outperform the Basic approach by a factor of as much as 4.8 and 3.9 respectively on a 32-node cluster. We have implemented our proposed designs and integrated them into MVAPICH [2], a popular implementation of MPI over InfiniBand which is being used by more than 200 organizations world-wide. The rest of the paper is organized as follows. In Section 2 we provide the background, Section 3 provides the motivation for our work, Section 4 presents the various design alternatives followed by performance evaluation, related work and conclusion.

2 InfiniBand Hardware Multicast Groups

The InfiniBand Architecture (IBA) [6] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. Especially, it provides support for hardware multicast. A hardware multicast group in IBA is realized as a set of ports connected together using a logical spanning tree. Each hardware multicast group has a unique Multicast Group IDentifier (MGID). The routing of multicast packets posted on a multicast group is handled using routing tables present in all the participating switches of the IBA fabric. The nodes join and leave a multicast group through a management action involving Subnet Management and Subnet Administration classes of IBA management. In the remaining part of the paper we use the term multicast management entity to describe the body which implements the functionality of these classes.

The multicast management entity is responsible for handling all the operations specific to multicast group construction from the end nodes. These operations are the following: **Multicast Group Create** which is issued by an end node to create a

multicast group. This is an explicit operation in IBA to provide a single control of group characteristics like Message Transfer Unit, etc. and allow members to join subversively. **Multicast Group Join** which is issued by the end node to join the multicast group and **Multicast Group Leave** which is issued for leaving the group. All these requests are transported using MAnagement Datagrams called MADs. The multicast management entity on receiving the Join/Leave requests, constructs the multicast spanning tree and updates the participating switches in the IBA fabric with the new routing information.

3 Motivation: Mapping between Multicast Groups and MPI Communicators

Communicators play an important role during MPI communication. Communicator objects encapsulate information about all the processes that communicate with each other. This is required for the underlying MPI implementation which interacts with the network device in the forwarding of the messages.

One important information which is required in a communicator to support hardware multicast is that of Multicast Group IDentifier (MGID). Consider a scenario where one process wants to send a message to all the other processes in the communicator. This process issues a MPI_Bcast call with the communicator object as one of its parameter. The underlying MPI layer then posts the message to the multicast group identified by MGID and the actual forwarding is automatically taken care of by the IBA layer.

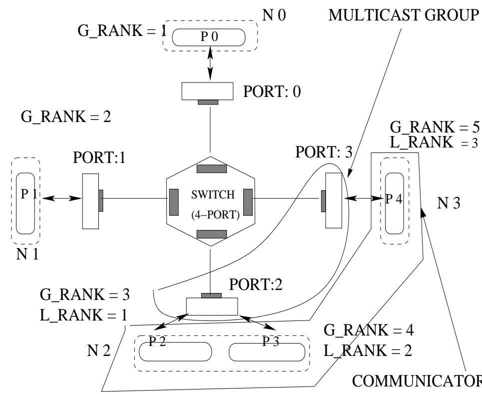

Fig. 1. Mapping between IBA Multicast Groups and MPI Communicators

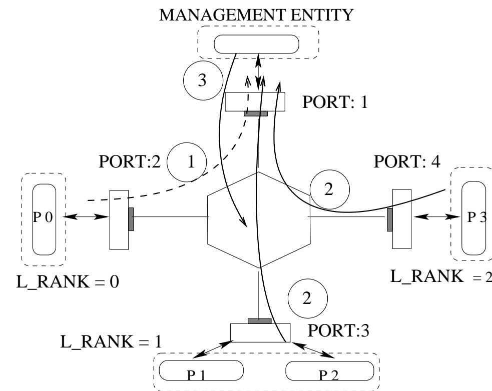

Fig. 2. Multicast Group Setup Operations

Figure 1 illustrates the relationship between the communicators and the hardware multicast groups. Let us consider an MPI application consisting of five processes, (P0-P4) as an example. These processes are launched on a subnet consisting of four end nodes (N0-N3) connected by a switch. Processes with global ranks three, four and five (i.e., P2, P3 and P4) are present in one communicator. The local ranks of these processes in the communicator are indicated in the figure. For these processes to use hardware multicast, the communicator has to be mapped to the hardware multicast group consisting of port numbers 2 and 3.

In the remaining sections of the paper, we explain how this mapping is done during communicator creation. An important factor to consider is that issuing the Create/Join

requests mentioned in the earlier section does not imply that the hardware multicast group is ready for use. This is because the multicast management entity has to first process these requests and construct a spanning tree containing the participating ports. Second, The routing tables in the fabric have to be updated to reflect the logical tree topology. The IBA specification does not define any specific mechanism of informing the processes of the completion of these tasks. Moreover, on large scale clusters, setting up multicast routing information can take considerable time if the size of the multicast group is comparable to the cluster size. This leads to the following questions:

1. How can the MPI application know when the multicast group is ready for use?
2. Can we minimize the overhead of multicast group construction from the MPI perspective?

We address these challenges in the following sections of the paper.

4 Communicator Creation Mechanism

In the following parts of the section, we provide an overview of MPI_Comm_create function and then propose our design alternatives of it. Though there are two types of communicators *intra* and *inter* defined in MPI, we focus on *intra* communicators in this paper.

4.1 MPI_Comm_create

MPI defines multiple function calls for creating new communicators. We have implemented all our designs using the MPI_Comm_create function. The semantics of communicator creation in MPI define that new communicators be derived from already existing communicators. So, the inputs to the function would be the the following objects, an already existing communicator object, a process group object and the pointer to the new communicator object. The process group identifies the subset of processes from the existing communicator that are to be present in the new communicator. MPI_Comm_create is a collective call invoked by all the processes in the existing communicator. In the following discussions, we focus on the communicator creation in the context of mapping these to the hardware multicast groups. All the other steps like the assignment of a unique context and the local ranks have already been done by the time we start constructing the multicast group.

4.2 Basic Design

The following steps are involved in the basic communicator construction. All of these are illustrated in Figure 2.

Multicast create and join: In this step, the process whose local rank is zero issues a create request to the multicast management entity specifying the Multicast Group IDentifier (MGID)(step 1 in Figure 2). The remaining processes then issue join requests to the multicast management entity using the same MGID (step 2 in Figure 2). All these requests carry the port identifiers so that the management entity knows which all ports would like to join a multicast group. The multicast management entity after receiving and validating the requests computes a logical spanning tree containing the ports specified in the requests. It then updates all the routing tables of the participating switches in the fabric (step 3 in Figure 2). At this point of time, the set up of hardware multicast group is complete.

However, the participating processes have no knowledge of this information. One approach to accomplish this would be to let the multicast management entity notify the MPI application after updating the routing tables. Another approach would be to let the MPI application discover about the completion independently. We have taken the latter approach in all our designs as it does not depend on any particular implementation of the multicast management entity. We refer to this approach as *multicast testing.*

Multicast testing: In this approach, the following algorithm is implemented by all the processes after they finish issuing the requests. Process with rank zero who is the root, posts a multicast *ping* message to the new hardware multicast group and waits for Acks from all the other processes. If the routing has been done, the message is received by all the processes and these processes soon post the Acks to the root. On the other hand, if routing is not complete then the message may not arrive at some of the processes. These processes block waiting for the *ping* message. Meanwhile, the root retransmits the *ping* message after a certain time-out interval. This process repeats until everyone has received the *ping* message.

4.3 Lazy Approach

Although the Basic design is good for its simplicity, it is blocking in nature. The application has to wait for the multicast management entity to process the requests and update the routing tables. Until then, all the processes block in the *multicast testing.* Depending on the size of the cluster and the multicast group this can take a considerable amount of time. Instead of doing the *multicast testing* in an eager fashion within the communicator creation call, we do this in a lazy manner by calling this routine every time a collective call is made. We do this until the *multicast testing* phase is over. We accomplish this by making the *multicast testing* as a non-blocking routine.

Asynchronous return: The new *multicast testing* is implemented in the following manner. The root process posts the *ping* message and checks for the arrival of the Acks from the rest of the processes. It does not block for the Acks to arrive. In the subsequent collective calls to this routine, it repeatedly checks for the progress of the Acks. It reposts the *ping* message only if the timeout is exceeded. The root keeps an estimate of the time elapsed by recording the time-stamps in the communicator object. The remaining processes behave in a similar fashion. They check for the *ping* messages in a non-blocking fashion and post the Acks soon after discovering the *ping* message.

Point-to-Point fall back: One important issue requiring detailed attention is the progress of the collective communication call before the communicator is ready for hardware multicast. In our approach, all the collective communication traffic is transmitted via point-to-point messaging until the root discovers that the routing has been done.

This approach overcomes the drawbacks of the Basic design. Due to the asynchronous nature of the *multicast testing* routine, overlap of computation as well as communication is easily achievable.

4.4 Multicast Group Pool Based Design

Though the Lazy approach can effectively hide the overhead of hardware multicast group construction in the MPI application, it still has some drawbacks. The benefits of hardware multicast in an application is reduced if the set-up time of the multicast groups is high and the collective communication follows the setting up of these communicators. Using our earlier design, the communication traffic falls back to point-to-point

if the multicast groups are not set up. But, this does not improve the performance of the application.

Multicast Group Pool: We overcome the drawback mentioned above using a *complementary* approach of setting up communicators explained as follows. The basic idea in this design is to have a certain pre-defined pool of multicast groups already constructed. These groups contain all the processes to begin with. In the Communicator construction routine, instead of participating nodes joining the multicast group, the non-participating nodes leave a multicast group chosen from the pool. There are several advantages of using this approach. First of all, since the multicast groups are already set-up the routing tables in the fabric are in place. So, when the application calls communicator creation function we can use the multicast group directly and we avoid the overhead of the *multicast testing* phase. This approach considerably improves the utility of the hardware multicast groups in an application. Secondly, the multicast pool can be maintained easily as most of the overhead is due to the multicast management entity and can be done in the background. We now explain the steps involved in this design.

When a call to the communicator creation is made, first a multicast group is chosen from the available list of multicast groups already constructed. If this pool is empty we fall back to the Lazy approach explained in the previous section. Once an available multicast group is obtained, the non-participating processes issue leave requests to the management entity. The list of non-participating process can be easily obtained by subtracting the set of the processes involved in the communicator from the global set involving all the processes. This global set is the MPI_GROUP_WORLD process group in MPI. Once a multicast group is consumed from the pool, it is immediately replenished by making all the processes issue requests for group construction. We also need to check for *multicast testing* before including the group in the pool. However, this check is done in the background by the application. The initial pool can be either constructed by the management entity or by the MPI application in the initialization phase. We have taken the latter approach in our implementation.

5 Performance Evaluation

Each node in our experimental testbed has dual Intel Xeon 2.66 GHz processors, 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. They are equipped with MT23108 InfiniBand HCAs with PCI-X interfaces. An InfiniScale MTS14400 switch is used to connect all the nodes. OpenSM, version 1.7.0, is the multicast management entity used in our tests.

5.1 Basic Hardware Group Setup Latencies

OpenSM has two parameters which affect the performance of multicast group creation. These are:1) timeout which is the time for transaction timeouts in milliseconds and 2) maxMADs which is the number of MADs that can be outstanding on the wire at any given point of time. We measure *multicast testing* to tune these parameters as this reflects the time taken by OpenSM to configure routing tables. Figure 3 shows these results. From these we have chosen 10 ms for timeout and the number of outstanding MADs is set to maximum for OpenSM to deliver best performance.

Figure 4 indicates the results of the basic multicast group operations like create, join and leave. We also present the *multicast testing* time for varying number of nodes.

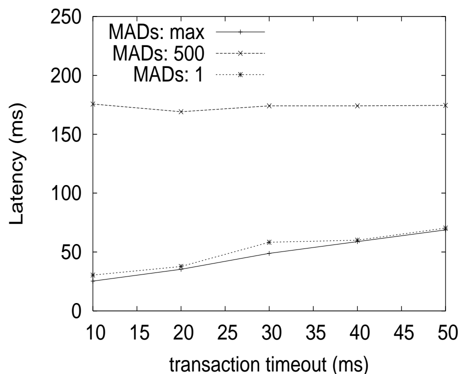

Fig. 3. Tuning of Multicast Testing

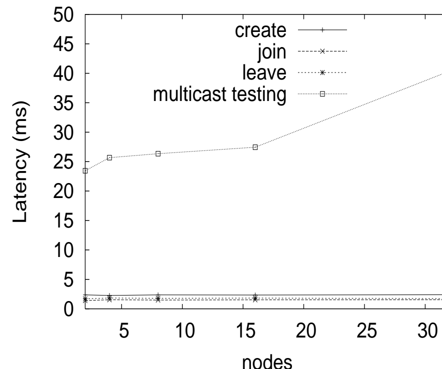

Fig. 4. Overhead of Basic Multicast Group Operations

As the figure indicates, *multicast testing* overhead is very high compared to the latencies of issuing create, join or leave requests. This is because as explained in the previous sections, after the requests are issued the management entity has to compute the spanning tree and update routing information of the switches in the fabric.

5.2 Effective Latency of Suggested Schemes

To compare the different schemes suggested in the paper we have measured the effective latency which is the latency of MPI_Bcast operation together with the communicator creation time. We have chosen the size of the message to be 1024 bytes in all our tests. The benchmark is constructed by calling communicator creation followed by the communication calls as many as the number of iterations specified. This is done for communicator sizes of 16 and 32 respectively. We have evaluated all the schemes for communicator sizes of 16 and 32.

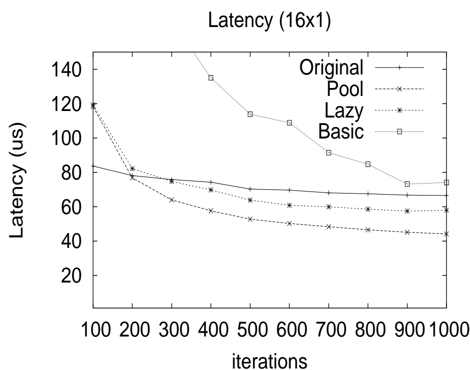

Fig. 5. Effective Latency with Collectives:16 processes

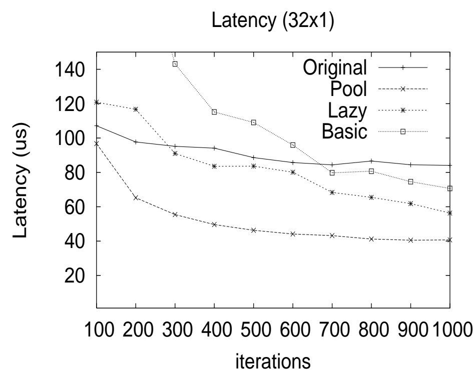

Fig. 6. Effective Latency with Collectives:32 processes

In Figure 5 we measure the effective latencies for varying number of iterations for all the three schemes: Basic, Lazy and Pool. We have also taken the traditional point-to-point collectives as the reference. We refer to this as the original design in the figures. As shown in the figure, the Pool based design outperforms all the rest. This is

because *multicast testing* phase can be fully overlapped with the communicator creator operations. The multicast group is immediately available. For the Lazy approach, we see the benefits of hardware multicast with the increasing number of iterations. This is because of the increasing percentage of communication using hardware multicast rather than point-to-point. The basic design performs poorly compared to all the designs. This is because of the very high overhead associated the *multicast testing* which is not overlapped with communication. Figure 6 shows the same trend for communicator size of 32. Note that the latencies of Pool and Lazy are almost the same for 16 and 32 for higher number of iterations. This is due to the scalability of hardware multicast.

To understand the overlap with computation we have introduced some computation between the communicator creation and the communication in the benchmark used for the above experiments. Figures 7 and 8 show the trend with increasing computation for sizes 16 and 32 respectively. The Lazy approach due to its asynchronous nature can overlap communicator creation with computation where as the Basic cannot. The Pool based design on the other hand can immediately take the benefits of hardware multicast. However, the initial latencies for size 32 are higher than for size 16 due to the increased overhead of creating larger hardware multicast group. As the Figure 8 indicate the Pool based design and the Lazy approaches improve the effective latency by a factor of 4.9 and 3.8 respectively.

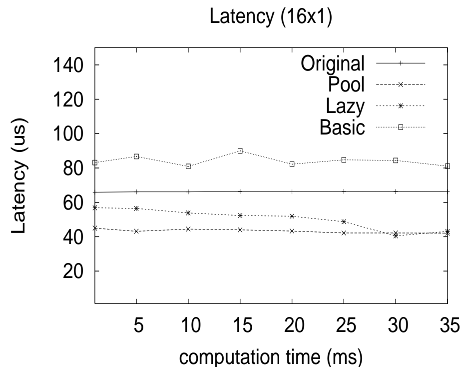

Fig. 7. Effective Latency with Computation and Collectives:16 processes

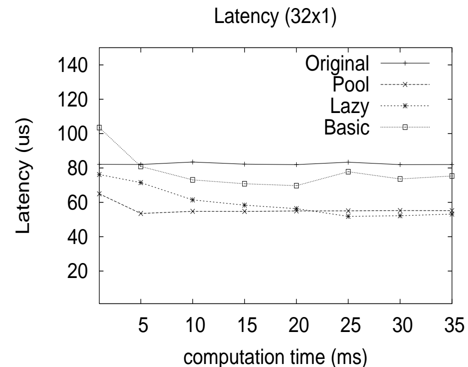

Fig. 8. Effective Latency with Computation and Collectives:32 processes

6 Related Work

Lot of research has been done in the area of subnet management over IBA. [3] [4] [9] study the various aspects of subnet management with respect to subnet discovery, routing and setting up of forwarding tables. The authors have shown the benefits of their scheme using a simulation of the subnet manager. In [5], the authors have implemented MPI collective operations using IP multicast. They have implemented their techniques over Fast Ethernet. In [11], the authors proposed different designs for constructing IP multicast groups. In [1], collectives have been implemented using hardware multicast and NACK-based schemes. Our work differs from these as we provide dynamic schemes of hardware multicast group construction in the context of InfiniBand and we overlap these with the application progress.

7 Conclusions and Future Work

In this paper, we propose efficient schemes of dynamically constructing communicators with hardware multicast support in the context of InfiniBand. The basic idea behind all the schemes is to overlap the group construction with the progress of the application. We have designed efficient schemes like the Multicast Pool and the Lazy approaches to move most of the overhead of multicast group creation to the background, i.e., not in the critical path of MPI application execution. We have evaluated these schemes together with our basic scheme and found out that the Multicast Pool performs the best of all the three followed by the Lazy scheme. Multicast Pool and Lazy schemes improve the Effective Latency by a factor of 4.9 and 3.8 respectively. In our future work, we would like to evaluate the impact of these schemes on a range of MPI applications with and without dynamic process creation.

8 Acknowledgements

We would like to thank Eitan Zahavi, Dror Goldenberg and Eitan Rabin from Mellanox for providing helpful comments.

References

1. Multicast collectives. http://vmi.ncsa.uiuc.edu.
2. Mvapich: Mpi over infiniband project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.
3. A. Bermudez, R. Casado, F. J. Quiles, T. M. Pinkston, and J. Duato. Evaluation of a subnet management mechanism for infiniband networks. In *Proceedings of ICPP*, 2003.
4. A. Bermudez, R. Casado, F. J. Quiles, T. M. Pinkston, and J. Duato. On the infiniband subnet discovery process. In *Proceedings of Cluster Computing*, 2003.
5. Y. O. Carrasco H. A. Chen and A. W. Apon. MPI Collective Operations over IP Multicast. In *Workshop PC-NOW 2000*, 2000.
6. InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2. http://www.infinibandta.org, October 2004.
7. J. Liu, A. R. Mamidala, and D. K. panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Proceedings of IPDPS*, 2004.
8. A. R. Mamidala, J. Liu, and D. K. panda. Efficient Barrier and Allreduce InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms . In *Proceedings of Cluster Computing*, 2004.
9. J. C. Sancho, A Robles, and J. Duato. Effective strategy to compute forwarding tables for infiniband networks. In *Proceedings of ICPP*, 2001.
10. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.
11. A. Faraj X. Yuan, S. Daniels and A. Karwande. Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast. In *The 6th International Conference on Computer Science and Informatics, Durham, NC*, March 8-14 2002.