Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support*

Jiuxing Liu Amith R Mamidala Dhabaleswar K Panda

Computer and Information Science
The Ohio State University
Columbus, OH 43210
{liuj, mamidala, panda}@cis.ohio-state.edu

Abstract

Modern high performance applications require efficient and scalable collective communication operations. Currently, most collective operations are implemented based on point-to-point operations. In this paper, we propose to use hardware multicast in InfiniBand to design fast and scalable broadcast operations in MPI. InfiniBand supports multicast with Unreliable Datagram (UD) transport service. This makes it hard to be directly used by an upper layer such as MPI. To bridge the semantic gap between MPI_Bcast and InfiniBand hardware multicast, we have designed and implemented a substrate on top of InfiniBand which provides functionalities such as reliability, in-order delivery and large message handling. By using a sliding-window based design, we improve MPI_Bcast latency by removing most of the overhead in the substrate out of the communication critical path. By using optimizations such as a new co-root based scheme *and* delayed ACK*, we can further balance and reduce the overhead. We have also addressed many detailed design issues such as buffer management, efficient handling of out-of-order and duplicate messages, timeout and retransmission, flow control and RDMA based ACK communication.*

Our performance evaluation shows that in an 8 node cluster testbed, hardware multicast based designs can improve MPI broadcast latency up to 58% and broadcast throughput up to 112%. The proposed solutions are also much more tolerant to process skew compared with the current point-to-point based implementation. We have also developed analytical model for our multicast based schemes and validated them with experimental numbers. Our analytical model shows that with the new designs, one can achieve MPI broadcast latency of small messages with 20.0μs and of one MTU size message (around 1836 bytes of data payload) with 40.0μs in a 1024 node cluster.

1 Introduction

Cluster based computing systems are becoming increasingly affordable and cost-effective for a wide range of scientific applications. These systems are typically built from commodity PCs connected with high speed Local Area Networks (LANs) or System Area Networks (SANs). In the area of parallel and high performance computing, the Message Passing Interface (MPI) [17] programming model is commonly used for writing applications. MPI provides both *point-to-point* and *collective* communication functions. Many parallel applications take advantage of collective operations. Some of the applications, such as IS and FT in the NAS Parallel Benchmark suite [12], almost use collective operations exclusively for communication. Thus, providing high performance and scalable collective communication support is critical for many cluster systems. In this paper, we focus on one of the commonly used MPI collective functions : MPI_Bcast. This operation broadcasts a message to all the other nodes in a communication group. MPI_Bcast can be used alone or as building blocks for other collective operations.

Currently, there are several network interconnects that provide low latency (less than 10μs) and high bandwidth (in the order of Gbps) for cluster based systems. Two of the leading products are Myrinet[13] and Quadrics[15]. More recently, InfiniBand [6] has entered the high performance computing market. One of the notable feature of InfiniBand is that it supports hardware multicast. Thus, a message can be efficiently delivered to multiple receivers. Although they look similar, the semantics of hardware multicast in InfiniBand do not match with those of MPI_Bcast. For example, multicast in InfiniBand is supported only in Unreliable Datagram (UD) service and does not guarantee reliable message delivery. This leads to the following questions:

1. Can we take advantage of hardware multicast in InfiniBand to provide broadcast support in MPI?
2. How can we bridge the semantic gap of InfiniBand multicast and MPI_Bcast in an efficient and scalable manner?

In this paper, we aim to provide answers to the above questions. To support MPI_Bcast, InfiniBand multicast lacks features such as reliability, in-order delivery and large

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, a grant from Sandia National Laboratory, a grant from Los Alamos National Laboratory, and National Science Foundation's grants #CCR-0204429 and #CCR-0311542.

message handling. We propose designing and using a substrate to enhance InfiniBand multicast by providing these features. This substrate is an integrated part of the MPI implementation and it exploits both multicast and point-to-point communication in InfiniBand.

Providing new features on top of InfiniBand multicast inevitably brings extra overhead. To achieve high performance and scalability, we have used two key design strategies. The first one is to remove the overhead from communication critical path so that it happens in the background. The second one is to balance and reduce this background overhead so that it is not a performance bottleneck in most cases. Based on the first strategy, we have proposed a sliding window based design which enables the root of MPI_Bcast to proceed without waiting for other nodes to send ACKs. Based on the second strategy, we have introduced a *co-root* scheme to balance background ACK traffic and various delayed ACK techniques to reduce the ACK traffic. We have also addressed many detailed design issues such as buffer management, efficient handling of out-of-order and duplicate messages, timeout and retransmission, flow control and RDMA based ACK communication.

We have implemented our designs and integrated them into our MPI implementation over InfiniBand. Compared with the current MPI_Bcast implementation, which is based solely on point-to-point communication, our new designs can improve broadcast latency up to 58% and throughput up to 112% in our 8 node testbed. With larger clusters, we expect that more benefit can be obtained from our designs because InfiniBand's hardware multicast scales very well when the number of nodes increases. We have developed analytical models to get more insight into the performance of different MPI_Bcast designs on large scale clusters. Our results show that on a 1024 node cluster, our designs can perform up to 3.86 times better than the current design and achieve MPI broadcast latency of small messages with 20.0μs and of one MTU size message (around 1836 bytes of data payload) with 40.0μs.

The rest of the paper is organized as follows: In Section 2, we provide an overview of the InfiniBand Architecture. In Section 3, we describe the MPI_Bcast operation. Our designs of MPI_Bcast over InfiniBand are presented in Section 4 at the algorithm level. In Section 5, we discuss detailed design issues. We evaluate our designs using experiments and analytical models in Sections 6 and 7, respectively. Conclusions and future research directions are presented in Section 8.

2 InfiniBand Overview

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-process communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs connect processing nodes to the network.

The InfiniBand communication stack consists of different layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair in InfiniBand Architecture consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. The completion of WQRs is reported through Completion Queues (CQs). InfiniBand also supports different classes of transport services. In current products, Reliable Connection (RC) service and Unreliable Datagram (UD) service are supported.

InfiniBand Architecture supports both channel semantics and memory semantics. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand provides Remote Direct Memory Access (RDMA) operations, including RDMA write and RDMA read. RDMA operations are one-sided and do not incur software overhead at the remote side.

2.1 Hardware Multicast in InfiniBand

One of the notable features provided by the InfiniBand Architecture is hardware supported multicast. It provides the ability to send a single message to a specific *multicast address* and have it delivered to multiple processes which may be on different end nodes. Although the same effect can be achieved by using multiple point-to-point communication operations, hardware multicast provides the following benefits:

- Since only one send operation is needed to initiate the multicast, it greatly reduces host overhead at the sender. By reducing this overhead, multicast latency as seen by each receiver is also reduced.

- With hardware supported multicast, packets are duplicated by the switches only when necessary. Therefore, network traffic is reduced by eliminating the cases that multiple identical packets travel through the same physical link.

Figure 1 shows the latency of InfiniBand multicast on our 8-node testbed. (The details of our testbed are indicated in Section 6.) We can see that multicast latency is not sensitive to the number of destination nodes. Therefore, it provides a very scalable approach to send data to multiple destinations.

In InfiniBand, hardware multicast operation is only available under the Unreliable Datagram (UD) transport service. In UD, a connectionless communication model is used. Messages can be dropped or arrive out of order. Thus, the hardware multicast support cannot be directly integrated to upper level programming models which require stronger semantics, such as reliability and in-order delivery.

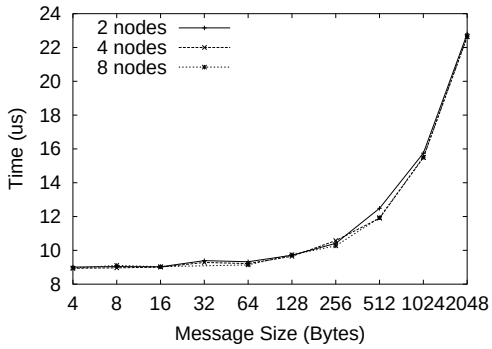

Figure 1. InfiniBand Multicast Performance

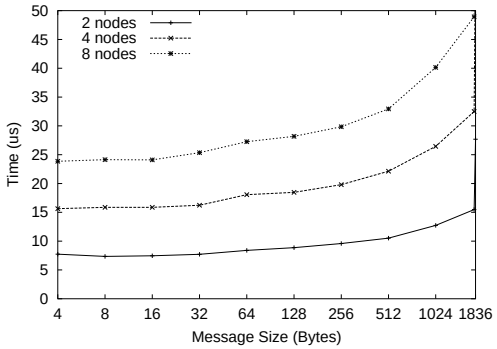

Figure 2. MPI_Bcast Latency in MVAPICH Using Point-to-Point Communication

3 MPI_Bcast Overview

MPI supports both point-to-point and collective communication functions. MPI_Bcast is a commonly used collective function. It broadcasts a message from a root process to other processes in a communication group, which is specified by an MPI communicator. In many cases, this communicator is MPI_COMM_WORLD, whose communication group includes all the processes participating in the MPI application. MPI_Bcast is a blocking operation. For a root node, the operation does not return until the communication buffer can be reused. For a receiver node, the operation returns only after the broadcast data has been delivered into the receive buffer. However, it is not necessary that the operation returns only after the broadcast is finished at the root.

In many MPI implementations, MPI_Bcast is implemented with a tree-based algorithm, which exploits point-to-point communication operations. This approach is used in our current MPI implementation over InfiniBand: MVAPICH [11, 9]. In the tree based approach, the number of hops to reach leaf nodes increases with the total number nodes (typically in a logarithmic manner). Therefore, MPI_Bcast latency also increases. In Figure 2, we show MPI_Bcast performance in MVAPICH using point-to-point communication. It can be seen that MPI_Bcast latency increases with the number of nodes.

Another drawback of tree based implementations is that if hosts are involved in intermediate nodes to forward broadcast messages, skew between different processes may significantly delay the forwarding [1]. This has adverse impact on the execution time of an application. Thus, the challenge is whether the hardware supported multicast scheme can alleviate the impact of process skew.

4 Designing MPI_Bcast with InfiniBand Multicast

In the previous section, we have seen that MPI_Bcast implementations based on point-to-point communication are not scalable with respect to the number of processes. It also indicates that point-to-point implementations are susceptible to process skew. In Figure 1, we see that InfiniBand multicast provides a more scalable way of delivering a single message to multiple destinations. However, there are several major differences between InfiniBand multicast and MPI_Bcast:

1. InfiniBand multicast does not guarantee reliability, while in MPI, communication must be reliable.

2. Since InfiniBand multicast uses connectionless UD service, there is no guarantee regarding the ordering of multicast messages. However, MPI specifies that all collective operations must be matched according to the order they are initiated.

3. In InfiniBand UD service, the size of a message cannot exceed the MTU (Maximum Transfer Unit), which is typically 2K Bytes. MPI does not limit the message size in MPI_Bcast.

In other words, there exists a semantic gap between InfiniBand multicast and MPI_Bcast. This issue must be addressed to take advantage of hardware multicast in an MPI implementation. In this paper, we propose a substrate which bridges this gap. As shown in Figure 3, this substrate sits on top of the underlying InfiniBand layer, exploiting multicast as well as other InfiniBand functionalities. It also interacts with other parts of the MPI implementation. To achieve high performance and scalability, we need to not only implement this substrate, but also do it in an efficient and scalable manner.

In designing the substrate, we need to address three issues: reliability, in-order delivery, and handling large messages. Previous study [19] has shown that data sizes in MPI collective operations are typically quite small. Therefore, we will first concentrate on efficient handling of small messages. We will deal with large messages specifically at the end of this section.

In the following, we propose several designs used in the substrate. We first describe a basic design which is easy to understand and also straightforward to implement. Then we present several new designs which deal with performance and scalability issues of the basic design. Although reliability and ordering are two different issues, they can be

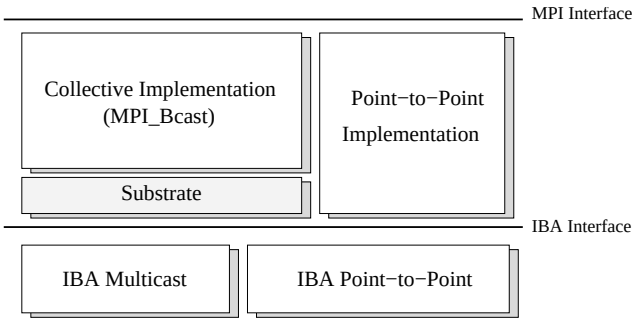

Figure 3. Bridging the Gap between InfiniBand Multicast and MPI_Bcast

addressed together. In the following discussions, we primarily focus on how to implement reliability. We have chosen ACK based approaches, in which delivery is confirmed by acknowledgments and message loss is handled by timeout/retransmission. In the proposed schemes, we also address how message ordering can be ensured for MPI_Bcast.

4.1 Basic Design

In the basic design, the root node of MPI_Bcast sends out a message using multicast and other nodes wait for this message. If the message is received, an ACK is sent back to the root node. The root blocks and waits for all ACKs to be received. If not all ACKs arrive within a certain period of time, it times out and retransmits the message using reliable point-to-point communication (using the RC service, as defined by the InfiniBand standard).

The basic design uses ACKs and timeout/retransmission to provide reliability. Two different broadcast messages from the same root are guaranteed to arrive in-order because the root node blocks for ACKs of the first message before it can send out the second one.

However, there are several major problems in this basic design. First, making the root block for all the ACKs significantly increases the overhead of the MPI_Bcast call at the root. Second, since all other nodes send back ACKs to a single root node, a hot spot is created at the root, which becomes a performance bottleneck when the total number of nodes in a system is large. This problem is also referred to as *ACK implosion* [16]. In the following subsections, we will address these problems.

4.2 Sliding-Window Based Design

Our basic design leads to poor performance because the root has to wait for all the ACKs to be received. MPI specifies that MPI_Bcast can return immediately after the buffer can be reused. Therefore, it is not necessary for the root to wait for all the ACKs to be received.

In order to alleviate this problem, we propose a solution which makes a copy of the user buffer. After the multicast operation is initiated, we can immediately return without waiting for all the ACKs to be received. To handle multiple outstanding MPI_Bcast initiated from a single root, we use a number of pre-allocated buffers at each root. These buffers are organized as a ring. A sliding-window based approach is used to manage these buffers, as shown in Figure 4. A buffer is consumed for each new MPI_Bcast operation. When all ACKs for this operation have arrived, this buffer can be freed and reused for other MPI_Bcast operations.

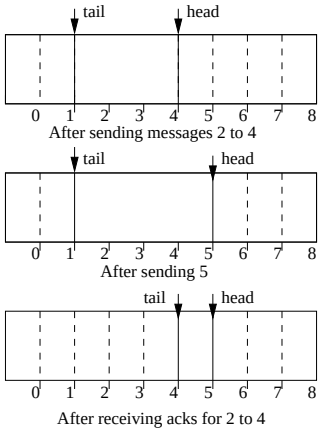

Figure 4. Sliding Window Buffer Management

Compared with the basic design, the sliding-window based design decouples ACK processing from the multicast. In other words, ACK processing is no longer done in the critical path of broadcast, but carried out in the "background". If the window size is sufficiently large such that all ACKs can arrive and be processed in time, MPI_Bcast will not block due to running out of buffers. As a result, the performance of MPI_Bcast can be significantly improved.

The window based design also has its drawbacks. First, the data in user buffers has to be copied to buffers in the window, which increases processing overhead. Fortunately, the typical size of MPI_Bcast is small and the copying overhead is negligible. Another problem is that it consumes more buffer than the basic design. We can control the buffer space used by changing the total window size. The third issue is that this design does not solve the ACK implosion problem. Although ACK processing is now done in the background, it still happens that all ACKs arrive at the same root node. Therefore, the root can become a performance bottleneck in this design for large scale systems.

4.3 Avoiding ACK Implosion

To solve the ACK implosion problem, we should not let all the receivers send ACKs to a single root node. The basic idea to deal with this problem is to use a hierarchical structure for ACK collection and distribute the load to a number of nodes. One solution is to use a tree based structure to collect ACKs. In this approach, all nodes form a tree structure, with the root node being the root of the tree. Intermediate nodes are responsible for collecting ACKs for its children. After all ACKs have come from its children, an intermediate node sends an ACK to its parent node. The root node

only needs to collect ACKs from its direct children instead of all other nodes.

The drawback of the tree based ACK collection is that it depends on intermediate nodes for ACK processing. Thus, ACK collection time depends on the communication progress of intermediate nodes. (A similar problem has been discussed in [1].) In a polling based MPI implementation such as MPICH, communication progress is only made within MPI function calls. Therefore, if an intermediate node is doing lengthy computation, ACK processing and forwarding could be delayed. The problem becomes even more serious when the tree has multiple levels. As a result, it is very hard to determine the timeout value for retransmission at the root. When ACK processing at intermediate nodes are delayed, the tree based ACK collection is prone to *false retransmission*, which is triggered by delayed ACKs instead of real message loss. To make matters worse, a single delayed ACK will result in the root node retransmitting the message to everyone in the same sub-tree, which can generate a lot of network traffic and increase the overhead of the root node.

To solve the ACK implosion problem and also to address problems with the tree based scheme, we propose a new ACK collection scheme called the *co-root scheme*. In this scheme, in addition to the root node, we select a subset of other nodes as *co-roots*. The remaining nodes are called *leaf nodes*. Each of the root and the co-roots is responsible for a group of leaf nodes. The basic idea is to guarantee that co-roots can get messages reliably and use them to help ACK processing. The co-root scheme is illustrated in Figure 5 and it consists of the following steps:

1. The root uses multicast to transfer the message to every other node.

2. The root does a small scale "broadcast" to all co-roots. The broadcast is done using reliable point-to-point communication. A tree based algorithm can be used, just like that in the current MPI implementation.

3. Each of the root and the co-roots collects ACKs from all other nodes in its sub-group. If timeout happens, the root or the appropriate co-root will do the retransmission.

Similar to the tree based ACK collection, the co-root scheme also uses a hierarchical structure to delegate ACK collection and processing to other nodes. They both aim to solve the ACK implosion problem. However, there are also major differences between them. The co-root scheme is a two-level hierarchy. After the message is delivered to a co-root, the co-root essentially plays the same role as the root and ACK processing for its sub-group is completely decoupled from the root. In a tree based scheme, intermediate nodes are responsible for ACK collection and forwarding, while the root is responsible for ACK collection and retransmission. The ACK processing is not completely decoupled

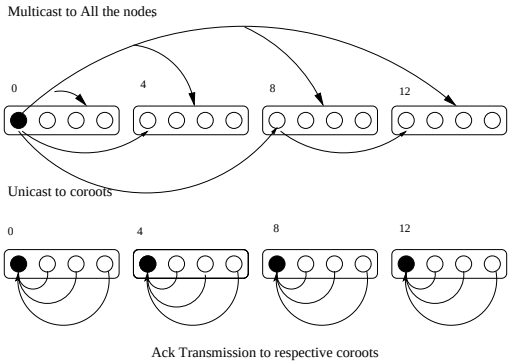

Figure 5. Co-Root Scheme

from the root because it has to handle all the retransmissions.

The co-root scheme has several advantages over a tree based scheme. Since co-roots now help with *both* ACK collection and retransmission, the load is more evenly distributed. The co-root scheme does not depend on the progress of intermediate nodes. As a result, it is easier to determine the timeout value for a given system size. The co-root scheme also results in fewer false retransmissions. (Note that false retransmission can still happen if an ACK from a leaf to its co-root is delayed.) Another advantage of the co-root scheme is that each co-root keeps information of all the leaf nodes in its sub-group. When an ACK is not received, retransmission is done only to that particular node. In a tree based scheme, the root can only track other nodes at the level of sub-trees. Therefore, retransmission must be done for all nodes in that sub-tree, which increases overhead and network traffic.

The co-root scheme also has its disadvantages. First, delivering the message reliably to every co-root introduces extra root processing overhead and network traffic. However, it should be noted that usually the co-root scheme does not increase latency of the broadcast. At any co-root, the broadcast can be completed when it receives either the multicast message or the "reliable broadcast" message. It does not have to wait for both messages. The second problem of the co-root scheme is that a copy of the message is duplicated at all co-roots. Therefore, it consumes more buffer space compared with a tree-based scheme. Another issue for co-root scheme is that we must carefully determine the number of co-roots (or the sub-group size). We address this issue (determining optimal number of co-roots) in Section 7.2.

4.4 Reducing ACK Traffic

ACK implosion avoiding schemes distribute ACK processing and retransmission tasks from the root to other nodes, but they do not reduce the total number of ACK messages. To improve utilization of the network resource and to avoid possible network congestion, it is also desirable to reduce ACK traffic.

Our basic idea of reducing ACK traffic is to send ACKs in a lazy manner. We propose two schemes:

1. *Piggybacking.* In this scheme, a node attaches the ACK with other messages instead of sending it as a separate message. If there is no message sending out to the ACK destination after a certain period of time, an explicit ACK message is sent.

2. *Acknowledge every M broadcast messages.* Instead of sending an ACK for every broadcast message, we only send one ACK for every M broadcast messages. Timeout and explicit ACK messages are also used.

Both schemes can reduce the total amount of ACK traffic. The effectiveness of piggybacking is very dependent on the communication pattern of the application. In the best case, all ACKs can be attached with other messages. In the worst case, timeout happens and we have to send the ACK using an explicit message. However, even in this worst case we can still possibly reduce ACK traffic. This is because we wait for the timeout before sending out the ACK messages. Therefore, if there are multiple broadcast messages received from the same root, they can be acknowledged using a single ACK message.

The second scheme effectively reduces the ACK traffic to 1/M of the original amount if there are many back-to-back broadcasts. However, one problem with the scheme is that after every M message, the root will receive ACKs from all other nodes. This leads to similar situations as ACK implosion. To solve this problem, we introduce a technique called *skewed ACK*. In this technique, every node still acknowledges after receiving every M messages. However, they now do the ACK in a more independent way. For example, suppose there are *n* nodes in the broadcast group and every broadcast message has a sequence number B, then node i can generate an ACK based on the following condition: $B \bmod M = i \bmod M$.

We should note that schemes to avoid ACK implosion and to reduce ACK traffic are complementary. By combining both schemes, we can achieve even more benefit. For example, the schemes proposed in this subsection can be used to reduce ACK traffic in the co-root scheme proposed in the previous subsection.

4.5 Dealing with Large Messages

In previous discussions, we dealt with small broadcast messages which can fit into a single buffer. (The buffer size is no larger than InfiniBand MTU.) Large messages can be divided into small chunks and sent out using multiple buffers. The techniques we have discussed previously are still applicable in this case. However, the copying cost may be significant because the message size is large. Since large broadcast messages are relatively infrequent, an alternative way is to fall back on schemes based on point-to-point communication. The advantage of this approach is simplified design and implementation. Also the overhead of the root due to the copy can be eliminated because zero-copy point-to-point communication can be used for transferring the message.

5 Detailed Design Issues

Our MPI_Bcast implementation is based on MVAPICH [11, 9], our MPI implementation over InfiniBand. MVAPICH is derived from MPICH [3], which was developed at Argonne National Laboratory and is one of the most popular MPI implementations. MVAPICH is also derived from MVICH [8].

In this section, we discuss some of the detailed design issues in our MPI_Bcast designs. These issues include buffer management, out-of-order and duplicate message handling, timeout and retransmission, flow control and RDMA based ACK communication.

5.1 Buffer Management

To ensure reliability of MPI_Bcast, we have to store broadcast messages in buffers until we can be sure that every other node has received this message. Therefore, buffer management is an important issue in our design.

For each node, a number of pre-allocated buffers are used for storing broadcast messages sent by this node. Since InfiniBand requires communication buffers to be registered, we pre-register these buffer to save cost during communication. The buffers are organized as a ring and managed using a sliding window based algorithm. For each new broadcast, the message is copied to the buffer at the head of the window. For a buffer at the tail of the window, if we have collected all ACKs, we free this buffer by incrementing the tail pointer. For the co-root scheme, a window of buffers exist also in all co-roots and are managed in the same way.

One parameter we have to decide in buffer management is the window size. A large window size means that the application can issue a large number of back-to-back broadcast without blocking because of delayed ACKs. However, using a large window size also consumes more buffer space. This parameter is best decided by the communication pattern of applications. Currently we use a static value for window size which can be changed at compile time.

One issue we have to deal with is what to do if we run out of buffers. In the current implementation, we treat this situation in the same way as timeout. Thus, we will retransmit the message to all nodes from which the ACK has not come.

5.2 Handling Out-of-Order and Duplicate Messages

Multicast messages in InfiniBand use Unreliable Datagram transport service, which does not maintain message order. Duplicate messages can also be sent to a receiver due to false retransmission or algorithms used in co-root scheme. These situations are handled by using sequence numbers attached with each broadcast message.

Each receiver maintains a counter which specifies the sequence number of the next broadcast message it is expecting. If the sequence number of the next message is equal to the counter, the message is processed and the counter is incremented. If the sequence number is larger than the

counter, the processing is delayed and the message is put into a queue. If an arriving message is a duplicate, its sequence number is either less than the counter value or equal to the sequence number of one of the messages in the queue. In this case, the message is not processed but silently discarded.

5.3 Timeout and Retransmission

Whenever a root issues a multicast message, it sets a timeout value for this message. For the co-root scheme, all the co-roots also set a timeout value after receiving the message from the root. When we set or check the timeout value, the current time value is obtained by reading the time stamp counter register provided by the Intel Pentium architecture. This approach has very low overhead. To check if a timeout value has been reached, we use a polling based approach. Therefore, timeout and retransmission only happen inside MPI functions calls. An alternative is to use an interrupt based method. However, this approach is not used because it brings many race conditions and does not match well with the polling based implementation of MPICH.

There are many factors which affect the timeout value, such as multicast and point-to-point latency, process skew, window size at the root (or co-roots), the number of co-roots and the system size. Currently, we use a static value which can be changed at compile time. We plan to investigate these issues in future with the availability of large-scale InfiniBand clusters.

Retransmission is always done using reliable point-to-point communication. After retransmission, the message buffer can be freed because we are now sure that the message can arrive at the receiver. In certain retransmission cases, such as those when a large number of ACKs are not received, it may be more efficient to re-issue the multicast operation. However, we decide not to use this approach because it complicates the implementation and these cases are quite rare.

5.4 Flow Control

In UD service, a multicast send operation will consume one buffer at every receiver. If there are not enough buffers posted at the receiver, incoming messages may be dropped. The purpose of flow control is to keep the root from sending if the receivers have not posted enough receive buffers.

We use a credit-based scheme for flow control. During initialization, a number of buffers are pre-posted and each node has an array of credit values for every other node which are equal to the number of pre-posted buffers. After receiving a broadcast message, a node will decrement the credit count of all nodes because a multicast operation will consume buffers at all receivers. After the message is processed and the buffer is re-posted at a receiver, the credit count for this node should be incremented at other nodes. This information is transferred using piggybacking. Both point-to-point messages and multicast messages can carry piggybacked credit information.

5.5 RDMA Based ACK communication

In our previous study [11], we have shown that RDMA operations in InfiniBand provide better performance than send/receive operations. Another advantage of RDMA is that there is no descriptor posting or management overhead at the receiver. To improve performance of ACK collection in MPI_Bcast, we have used RDMA write operations for ACK collections. To send back an ACK, a receiver issues an RDMA write to a memory location at the root (or its co-root). The root or co-root only needs to check the memory location in order to find out if an ACK has come. Since this check only involves memory read, it is very efficient.

6 Performance Evaluation

In this section, we evaluate performance of our MPI_Bcast designs based on InfiniBand multicast. We present results for several different designs and compare them with the original implementation in MVAPICH, which is a point-to-point implementation based on the binomial tree algorithm. We characterize broadcast performance using two micro-benchmarks: *latency* and *throughput*. We also show how process skew can affect different implementations. Since different ACK implosion avoiding techniques and ACK traffic reducing techniques may be combined, we can have different combinations for multicast based schemes. We have chosen only a subset of all possible combinations in the performance evaluation. All schemes used in our tests and their abbreviations are as follows:

- Original: the original implementation based on point-to-point communication.
- Basic: the basic design.
- Window: sliding-window based design without ACK implosion avoiding or ACK traffic reduction.
- Co-root2: sliding-window based with one co-root.
- Aggregate10: sliding-window based with ACK for every 10 messages.

Our experimental testbed consists of a cluster system with 8 SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with a 512K L2 cache and a 400 MHz front side bus. The machines are connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system with 2.4.7 kernel. The compilers we used were GNU GCC 2.96 and GNU FORTRAN 0.5.26.

6.1 Latency Test

We define *broadcast latency* to be the time it takes for a broadcast message to reach every receiver. Figure 6 shows the broadcast latency results for different designs. The buffer size in the multicast window is 2K bytes, which is equal to the MTU. However, because of the message header and other overhead (a portion of the buffer is used to store

descriptor.), currently we can send a payload of up to 1836 bytes in a single buffer. We can see that our new implementations based on InfiniBand multicast performs significantly better than the original broadcast implementation based on point-to-point communication. In the broadcast latency test, most of the ACK processing is carried out in the background. Thus, all the multicast based designs shown in the figure perform comparably. For small messages, multicast based designs can perform up to 58% better than the original design.

6.1.1 Large Message Latency

Figure 7 compares one of the designs (Window) with the original design. We can see that although handling large messages requires fragmentation and reassembly of messages and extra copies, the performance can still be improved by using InfiniBand multicast for message sizes up to 32K bytes. For example, the improvements are 210% for 2K byte messages and 86% for 8K byte messages.

6.2 Throughput Test

We use *broadcast throughput* to measure how fast MPI_Bcast operations can be issued and finished. In this test, a number of back-to-back MPI_Bcast operations are issued from a root node. The throughput is simply the number of broadcast operations finished divided by the total time.

Figure 8 presents the throughput results for a number of different designs. We can see that the basic design performs the worst even though it uses InfiniBand multicast. This is because it always waits for all the ACKs before initiating the next broadcast. However, if we use a sliding-window based design, we can perform significantly better than the original design. By using ACK reducing technique, the performance can be further improved because the overhead to process ACKs is reduced. We can see that Aggregate10 scheme can perform up to 112% better than the original scheme in terms of throughput.

6.3 Impact of Process Skew

To measure the effect of process skew on broadcast performance, we use a test similar to that in [1]. The test consists of a loop, in which a barrier operation is performed before a broadcast. To emulate the effect of process skew, a random delay is inserted between the barrier and the broadcast for all the receiver. We then measure the average time spent in MPI_Bcast.

Figure 9 shows the impact of process skew on MPI_Bcast. Schemes Window and Original are chosen for comparison. We can see that multicast based scheme is not affected by process skew at all. In contrast, the original design relies on intermediate nodes to forward broadcast messages. Therefore, as process skew increases, the receivers spend more time in MPI_Bcast. When the average process skew is 400μs, the multicast based scheme performs 10 times better than the original design in the process skew test.

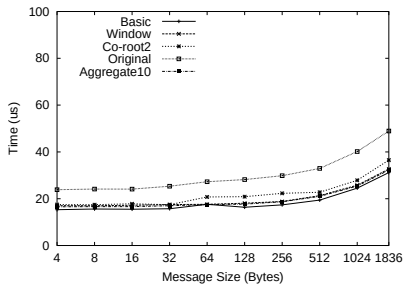

Figure 6. MPI_Bcast Latency for Small Messages (8 Nodes)

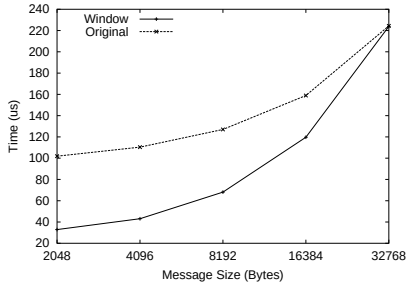

Figure 7. MPI_Bcast Latency for Large Messages (8 Nodes)

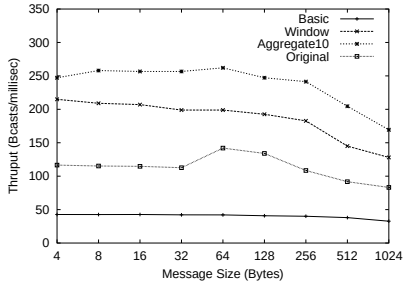

Figure 8. MPI_Bcast Throughput (8 Nodes)

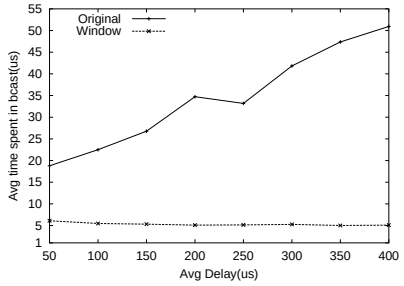

Figure 9. Impact of Process Skew on MPI_Bcast (8 Nodes)

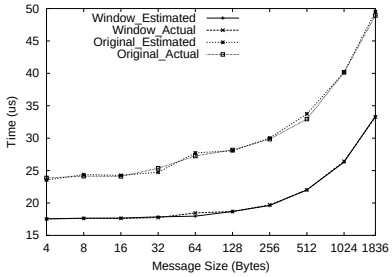

Figure 10. Estimated and Actual MPI_Bcast Latency (8 Nodes)

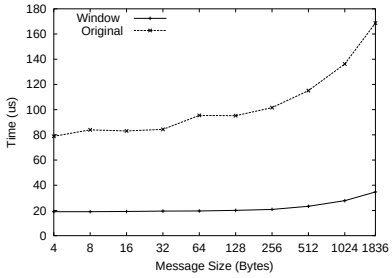

Figure 11. Estimated MPI_Bcast Latency (1024 Nodes)

7 Analytical Model

In this section, we use analytical models to characterize different MPI_Bcast implementations. Since our experiments were done in a relatively small testbed, these models help us to estimate the performance of different schemes in large scale systems.

7.1 Modeling Broadcast Latency

The broadcast operation in the original implementation happens in a binomial fashion. If we assume that the average latency of each hop is *Tpp*, the latency of the whole broadcast is given by *Tpp * log(n)*.

We now consider the latency for sliding-window based schemes. With only one root, the latency in this case is due to the cost of copying (*Tcc*) at the root, cost of posting a UD descriptor (*Tud*), latency of the hardware multicast (*Tmc*) and cost of copying (*Tcc*) at the receiver. The latency for a single root (*Tsr*) is thus given by: *Tsr = 2Tcc + Tud + Tmc*, where *Tmc* is the time taken for the multicast packet to travel from the root to the farthest node.

For the case of multiple roots, we have to consider two latencies. One of which is the multicast latency *Tsr* we have considered above. The other latency is due to the broadcast to all the co-roots using the binomial algorithm. This latency (*Trc*) is *Trc = 2*Tcc + Tud + Tpp * log (n/s)*, where s is the size of each subgroup.

For the nodes which are not the co-roots it takes *Tsr* for the message to arrive at these nodes. The latency at the co-roots (*Tcr*) is the minimum of Trc and Tsr. This is because the co-root gets the earliest of the unicast and the multicast messages destined to it.

The latency of the whole operation is thus the maximum of the *Tsr*'s for nodes other than the co-roots and *Tcr*'s for the co-roots. The latency of the operation is also determined by how these nodes are mapped to the fabric. This is because the topology is one major factor affecting the latency of hardware multicast.

Figure 10 shows the results estimated by our model and the actual measurements on our 8-node cluster. We can see that the model matches our measurements quite closely. Figure 11 shows the estimated results for the original implementation and the sliding-window based schemes in a 1024 node cluster. (In our model, the number of co-roots does not have significant impact on the results.) From the figure we can see that in a 1024 node cluster where each node has similar configuration as those in our testbed, using InfiniBand multicast can improve MPI_Bcast performance significantly. For small message, the potential latency improvement can be as high as 4.86 times. Our design can achieve MPI broadcast latency of small messages with 20.0μs and of one MTU size message (around 1836 bytes of data payload) with 40.0μs.

7.2 Determining the Number of Co-Roots

One of the important issues in the co-root scheme we proposed is to determine the number of co-roots for a given system. Since in the broadcast latency test, most of the processing overhead is in the background, different number of co-roots tend to give similar performance. Therefore, we use broadcast throughput to help us determine the optimal number of co-roots. Since in the co-root scheme, the root and the co-roots are responsible for ACK processing, they tend to be the bottleneck in the throughput test. We consider the time spent at the root in the throughput test. This is equal to *Tcc + Tud + Tum * log(n/s) + Tpack * s*, where Tpack is the ACK processing time per single node and Tum includes the copy cost and the time for posting a unicast message to one node.

Based on the model, if the number of co-roots is large, the root needs to spend a large amount of time to reliably deliver the message co-roots, and if the number of co-roots is too small, then each co-root needs to spend a large amount of time processing ACKs because the sub-group size is large. We have found that 128 co-roots are the best choice for 1024 nodes. The details can be found in [10].

8 Related Work

There have been many studies about multicast and reliable multicast in the networking area [5, 2]. A majority of the work done in this area focuses on networks based on TCP/IP protocol. Our work in this paper deals with implementing MPI_Bcast in InfiniBand. Compared with a general TCP/IP network, InfiniBand offers much higher communication performance and hardware supported multicast. Also, group membership in MPI is much more static than that in the dynamic environment of a TCP/IP network.

Recently, different collective operations in MPI have been studied on high speed interconnects such as Virtual Interface Architecture (VIA) [4], Quadrics [14], Myrinet [20] and IBM SP [18]. Compared with these interconnects, InfiniBand provides new challenges and opportunities for implementing MPI collective operations. Our previous work [7] proposed an RDMA based scheme to implement efficient barrier operations over InfiniBand. In this paper, we continue our work in this direction by presenting different broadcast designs while exploiting the hardware multicast support.

9 Conclusions and Future Work

In this paper, we described how to take advantage of hardware multicast in InfiniBand to implement MPI_Bcast operation in MPI. To support MPI_Bcast, we proposed a substrate on top of InfiniBand multicast which provides reliability, in-order delivery and handling of large messages. To improve performance of the substrate, we use sliding window based design which removes much of the processing from communication critical path. To further balance and reduce processing overhead, we proposed techniques such as the *co-root* scheme and *delayed ACK*.

Our performance evaluation on our 8-node cluster shows that our designs can improve MPI_Bcast latency up to 58% and throughput up to 112% compared with the current implementation. Our new designs also have much better tolerance to process skew. To get more understanding of the performance of MPI_Bcast in large-scale clusters, we use analytical modeling to estimate the performance of different designs. Our results show that in a 1024 node cluster, our designs can achieve MPI broadcast latency of small messages with 20.0μs.

In this work, we have focused on ACK based schemes to ensure multicast reliability. We plan to explore the use of NAK based schemes and combine them with ACK based schemes. We also intend to evaluate our designs on larger cluster to better understand possible scalability bottlenecks. Another direction we are currently pursuing is high performance and scalable implementation of other MPI collective operations such as MPI_Allreduce, MPI_Alltoall and MPI_Reduce on top of InfiniBand.

References

[1] D. Buntinas, D. K. Panda, and R. Brightwell. Application-Bypass Broadcast in MPICH over GM. In *International Symposium on Cluster Computing and the Grid (CCGRID '03)*, May 2003.

[2] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[4] R. Gupta, V. Tipparaju, J. Nieplocha, and D. K. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[5] H. Eriksson. Mbone: the multicast backbone. *Communications of the ACM*, August 1994.

[6] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[7] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *Euro PVM/MPI Conference*, Venice, Italy, September 2003.

[8] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html, August 2001.

[9] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing 2003 (SC '03)*, November 2003.

[10] J. Liu, A. R. Mamidala, and D. K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. Technical Report, OSU-CISRC-10/03-TR57, October 2003.

[11] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[12] NASA. NAS Parallel Benchmarks.

[13] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[14] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, February 2002.

[15] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002.

[16] S. Pingali, D. Towsley, and J. F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 221–230, New York, NY, USA, 1994. ACM Press.

[17] 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.

[18] V. Tipparaju, J. Nieplocha, D.K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[19] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *IPDPS*, April 2002.

[20] W. Yu, D. Buntinas, and D. K. Panda. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2. In *Int'l Conference on Parallel Processing, (ICPP 2003)*, Kaohsiung, Taiwan, October 2003.