LRUM: Local Reliability Protocol for Unreliable Hardware Multicast

Hoang-Vu Dang

University of Illinois at Urbana-Champaign
Champaign, IL
hdang8@illinois.edu

Richard Graham

Mellanox Technologies
Knoxville, TN
richardg@mellanox.com

Brian Smith

Mellanox technologies
Knoxville, TN
brians@mellanox.com

Gilad Shainer

Mellanox technologies
Sunnyvale, CA
shainer@mellanox.com

ABSTRACT

This paper describes two new Message Passing Interface (MPI) broadcast algorithms whose performance is essentially independent of communicator size. These are based on using the InfiniBand unreliable datagram (UD) hardware multicast capabilities, with a latency which is very close to that of the MPI ping-pong point-to-point latency between the root and the furthest away process in the communicator. These algorithms rely on a new scale-independent local reliability protocol that guarantees destination buffer availability under load imbalance. Performance is compared to that of HPC-X/Open MPI, MVAPICH and IntelMPI. The new algorithms provide the best available latency across the board. At 128 processes the new algorithms are 2.3 times better at four megabytes, 5% better at four kilobytes, and provide comparable performance at eight byte broadcasts when compared to the next best broadcast implementation. The new algorithms also demonstrate the lowest streaming latency and highest broadcast throughput.

KEYWORDS

ACM Reference Format:

Hoang-Vu Dang, Brian Smith, Richard Graham, and Gilad Shainer. 2018. LRUM: Local Reliability Protocol for Unreliable Hardware Multicast. In *HPC Asia 2018: International Conference on High Performance Computing in Asia-Pacific Region, January 28–31, 2018, Chiyoda, Tokyo, Japan*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3149457.3149467>

1 INTRODUCTION

Parallel applications often require some of their data to be replicated across the full or partial portion of the application processes or as a notification mechanism to a predefined set of processes. For example, a single process within the parallel job may read input data and send it on to other processes within the job. A single process within an allreduce operation may obtain the full result, and distribute it to the rest of the processes. The High Performance

Linpack benchmark sends data down a column of processes, and some Graph500 implementations send the discovered vertices at a given stage of the Breadth-First-Search to other processes in the job.

The broadcast operation reliably sends data to several other processes. This motif is in wide enough use that is a predefined primitive in several communication standards and parallel aware languages, as part of their support for collective communication. These include but are not limited to the Message Passing Interface (MPI) [3], the OpenSHMEM specification [4], Unified Parallel C [6] and CoArray Fortran [1]. This paper focuses on MPI broadcast operation, however the algorithms are applicable to other communications libraries and APIs as well.

Both scalability and performance of the broadcast operation are important for large systems. Scalable and well performing implementations, based on user-level point-to-point communication to distribute the data, scale logarithmically. Pipelined tree-based distributions are in use, performing a scatter operation followed by an all-gather operation. These algorithms perform well, especially for larger messages however, they still require multiple sends of the same data which multicast methods avoid.

However, network hardware that supports a *multicast* feature provides an opportunity for algorithms that scale as $O(1)$. Such a feature provides an attractive primitive on which to build support for a broadcast operation. The InfiniBand specification defines an unreliable UD multicast operation, which replicates data to all members of a multicast group in $O(1)$ time. Mellanox's production InfiniBand hardware supports this capability in its switch hardware with the latency of the operation being similar to that of the point-to-point latency from the source of the data to the furthest away Host Channel Adapter (HCA) in the group. The operation is unreliable, with the reasons for dropping such data being corrupt data packets or unavailable receive buffers. Mellanox's bit error rate is very low, on the order of 10^{-15} , and as such the rate of dropped packets is very low, even with EDR network links which transmit at 10^{11} bits/second. With this low error rate it is highly desirable to use the UD multicast capabilities, if one can design an algorithm whose reliability protocol overheads are low, and which does not cause the loss of the $O(1)$ characteristic of the algorithm.

Algorithms and implementations atop Mellanox's hardware multicast feature have been explored before [9, 10], but they have some performance issues and are not well adopted due to the lack of a

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HPC Asia 2018, January 28–31, 2018, Chiyoda, Tokyo, Japan

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5372-4/18/01.

<https://doi.org/10.1145/3149457.3149467>

systematic reliability protocol. MVAPICH2, a highly optimized MPI implementation atop InfiniBand Verbs implemented a multicast-based algorithm [10] but does not recommend it by default. This is mainly because the overheads of their reliability protocol are too high for common cases such as small messages or a relatively small number of hosts where traditional algorithms using point-to-point are better.

In this paper, we describe a multicast broadcast with an improved reliability algorithm that takes $O(1)$ steps when there is no packet loss, and logarithmic steps in the worst case, when data loss occurs. The algorithm relies on a reliability tree for error-detection and a buffer management scheme for flow-control and recovery. This results in scalable algorithms in both time and memory use. Furthermore, the algorithm is highly generic and flexible, and can be tuned to adapt to packet drop rate. The new of MPI_Bcast algorithms take into account message size and buffer availability, and outperforms the generic Intel MPI implementation by up to 40x and the vendor's implementation by up to 2.3x.

2 BACKGROUND

InfiniBand Verbs (IBverbs) [2] is a standard user's API interface for modern High-Performance Network Interface Cards (NIC). IBverbs is known for its Remote Direct Memory Access (RDMA) interface which allows the network device to bypass the OS and deliver data directly to user buffers. For this reason, IBverbs is adopted by many Network Interface Cards for HPC clusters. Mellanox has implemented the interface for their NICs, such as its ConnectX-5 cards.

Data transfer and delivery in IBverbs is based on a queuing model. A message is sent to a destination queue using a data structure enclosing a queue identifier (QID). To receive the message, the host having the queue is required to post a receive buffer to the queue so that the data can be delivered. When the message is received, a completion event is generated and enqueued to the *completion queue* (CQ). The QID is a virtual number created to represent a connection *end-point*, attached to a CQ and exchanged between end-points before the communications take place.

The device can provide transport reliability with the *Reliable connection* (RC) transport and *Dynamically Connected Transport* (DC). RC however requires each pair of connections to be established i.e. there is a separate QID for each pair of connected hosts, thus there are up to $O(N^2)$ total pairs of connections in the entire job, where N is the number of hosts for each host to be able to communicate with every other host. Each host participates in up to $O(N)$ connections. To avoid the need to establish this large number of connections, the user can choose *Unreliable datagram* (UD) or DC. UD allows sending data one chunk at a time, namely a *packet*, with the maximum size specified by the Maximum Transfer Unit (MTU) of the device to any host with a QID created independently in each host. Although this mode is more scalable than RC, UD does not guarantee reliable transmission so the user has to provide a reliability protocol to handle possible packet loss.

UD Multicast is an interesting and attractive feature exposed by IBverbs. To perform a multicast, a multicast group identifier (MGID) can be created and made known to all the hosts. A host can join this multicast group using the MGID and a QID created in UD mode

for receiving the data. Subsequently, a packet sent to this MGID will be delivered to all the hosts connected to the multicast group. Creating a multicast group is generally expensive and there is also a limited number of supported multicast groups in each system. Hence, multicast groups are usually created during initialization phases. An efficient multicast implementation replicates packets at each host very efficiently, and delivers the packet to all queues attached to the multicast group in parallel, resulting in a multicast latency comparable to that of the point-to-point latency from the root to the furthest away multicast queue.

To support this switch hardware capability, a multicast tree spanning the multicast group is defined. When data is sent to the multicast group, the switch recognizes the destination as a multicast group, and at each switch replicates the UD packet to all output ports in parallel. Thus the worst-case cost of the operation is that of a UD point-to-point operation to the furthest away group member.

Using multicast provides an opportunity to implement many collective operations which involve a one-to-all motif of communication. The obvious example is the *Broadcast* or specifically an MPI_Bcast as used by MPI. In MPI, a group of processes is often represented in a *communicator*. The default MPI_COMM_WORLD communicator represents all processes in the MPI system. The multicast group therefore, can be created, initialized, and joined by all processes in a communicator at the communicator creation time. Then when an operation is issued, the data is packetized as necessary and sent to the multicast group. The main challenge here is to detect and recover from packet losses since multicast is only available in UD mode and MPI assumes a reliable transmission.

In general, a packet loss can occur in two ways: (1) the receiver does not post sufficient buffers to the receive queue, thus when the packet arrives, there is no available space to store the data; (2) the packet is corrupted/dropped during transmission at the switch or host level due to network congestion or a transient fault. To prevent the first occurrence, a reliability protocol should ensure sufficient buffering at all time: it has to provide buffers before the data can come and further, apply some back-pressure mechanism to the parent when it knows the buffer will not be available. There is no way to prevent the second reason, thus a mechanism for detecting and recovery from such loss is required. We will go into more detail when we discuss our reliability algorithm in the next section.

3 BROADCAST ALGORITHMS

This section describes the algorithms developed to obtain broadcasts with $O(1)$ scalability characteristics. First, the common components of the algorithm are described, and then details are provided for the algorithms that have been implemented.

3.1 Key Ideas

As mentioned, the main challenge in providing a reliable broadcast algorithm using the hardware multicast capability is handling the unreliable nature of the UD protocol. Packets can be dropped due to lack of receive buffers at the destination or due to packets being corrupted in transmission. To ensure destination buffers are always available a sliding window based approach is used, with the root initiating a multicast transmission only when the window

has available space. The data corruption rate is a function of the underlying hardware's bit error rate. For Mellanox's hardware the error rate is very low, and therefore the algorithms are designed to minimize the overheads of successful completion, perhaps at the expense of a somewhat more costly recovery protocol, where data may be forwarded through several levels in the tree, providing the missing data at each level. The worst-case number of forwarding steps, when a full branch of the reliability tree did not get the data, scales as the $\log_k$ of the communicator size, where k is the tree radix.

To achieve $O(1)$ scalability for successful multicast transmission, by definition, the reliability protocol must be independent of the number of processes in the communicator. Therefore, the protocol must ensure that any coordination required be independent of communicator size, with fixed per rank costs. To achieve this scalability goal, each process in the communicator keeps copies of the multicast data in its copy of the sliding window. A tree structure is used to define parent/child relationships between ranks in the tree, with the root of the tree being the root of the broadcast operation. When a rank is missing data, it requests it from its parent, and the parent can reuse the temporary buffers once its children have acknowledged getting the data. Thus the number of messages associated with the reliability protocol is independent of the communicator size.

Missing data is detected in two ways, (1) missing sequence numbers and (2) expiring timers. Since the InfiniBand standard guarantees ordered delivery of multicast messages, the broadcast protocols assign a sequence number to each multicast packet. When a packet arrives out of order this is used as the indication that all packets between the current one and the previously received packet have been dropped. However, a gap in the sequence number cannot be used to detect the situation where the dropped packets are the last n packets sent; in such case a watch-dog timer is used. Because processes in the communicator enter the collective operation at different times and with the assumption that dropped packets are a rare event, a long timeout period is used to avoid premature data retransmission.

3.2 Reliability Algorithm

The reliability algorithm employs two principal components, a tree used to define the reliability protocol communication pattern, and a buffer management scheme used to ensure the scalable availability of missing data.

3.2.1 Reliability Management Trees. As described above, a tree is used to define a parent/child hierarchy that is used for the purpose of providing missing data. The root of the tree is the source of the data, thus ensuring that each process in the tree has a predecessor with the data. A request for the missing data may be forwarded up the tree until a node in the tree is found that can service the request. That node can then forward the data requested down the tree to the processes needing this data. We utilize a reliable transport for data traversing this tree.

Two different algorithms are implemented, one for short messages and the other for long messages. Both are derived from a basic algorithm that is described in two parts.

Algorithm 1: Reliability for single-packet multicast, assuming send/recv procedure for reliable point-to-point communication.

Input: p : parent process, C : set of child processes,
 r : Boolean value, true if this is a receiving process,
 pkt : multicast-received packet – $\emptyset$ if not received.

```

1 step1:
2 if  $r$  then
3   if  $pkt \neq \emptyset$  then
4      $\text{send}(\text{ACK}, \text{parent})$ 
5   end
6   else
7      $\text{send}(\text{NACK}, \text{parent})$ 
8     goto step1
9   end
10 end
11 step2:
12 for  $c \in C$  do
13    $\text{recv}(\&\text{status}, \text{child})$ 
14   if  $\text{status is NACK}$  then
15      $\text{send}(pkt, c)$ 
16   end
17 end

```

First, the two step reliability protocol for ensuring single packet reliability is described in algorithm 1. Figure 1 further illustrates the protocol with an example multicast using ten processes and a binary tree for the reliability protocol.

Line 2 – 10 in the pseudo-code describes how a process behaves, while acting in the capacity of a child in the tree:

- (1) If packet is received successfully, the process may send an ACK to its parent.
- (2) If a missing packet is detected, a NACK is sent to the parent, requesting the data. If a parent receives a NACK and does not have the needed data, the request is propagated to its parent, with the request propagation continuing up the tree until a process with the data is found. Once that parent possesses the requested data, it sends it to the requesting child.

Lines 12 – 17 describe the role of a parent in this reliability protocol. The parent sends missing data to the children in response to an explicit request for such data.

Such a scheme ensures that in the common case, where data arrives as expected, a process exchanges a fixed number of messages with its neighbors in the tree. In addition, the parent is released immediately from this child's portion of the reliability protocol, rather than waiting on acknowledgments to propagate up the full depth of the tree which would create a dependency on the size of the communicator.

The base algorithm is extended by aggregating acknowledgments to cut down on overheads associated with the algorithm. This may take place either across multiple-packets in a multi-packet message, or across messages, and is described in more detail in the next section.

Figure 1: Multicast with 10 processes from sender 0, and a reliability algorithm with a binary tree rooted at 0: red nodes are good processes and blue nodes are bad processes

Algorithm 2: Window algorithms for many-packet multicast, assuming a *reliability* procedure which makes sure all packets are sent/received (using Algorithm 1), Msend/Mrcv are the multicast operations.

Input: r : Boolean value, true if this is a receiving process.
 P : set of packets to send/receive.
 W_i, W_o : sets of buffers for storing packets.

```

1 for  $pkt \in P$  do
2   start:
3   if  $\exists w \in W_i$  then
4     if  $\neg r$  then
5       | Msend( $pkt$ )
6     end
7   else
8     | Mrcv(& $pkt$ )
9   end
10   $w = pkt$ 
11   $W_i = W_i - w$ 
12   $W_o = W_o + w$ 
13 end
14 else
15   reliability( $W_o$ )
16   swap( $W_i, W_o$ )
17   goto start
18 end
19 end

```

3.2.2 The Window Algorithm. As mentioned above, a window algorithm is used to aggregate acknowledgments, and to aid in the temporary buffer management scheme.

Algorithm 2 outlines the approach taken. Each process maintains a pool of temporary buffers into which they receive the incoming multicast data. An alternative is to receive the data directly into user buffers, however, the overheads due to synchronization were too high to offset the cost of the memory copies associated with the use of temporary buffers. W_i represents a free slot in the window, while

Figure 2: State diagram for window algorithm with $|W| = 2$ for a sender and a child receiver: *open* is when the window has available slots, *close* is when the window is full. For the sender, the state becomes closed after two multicasts, and for the receiver after receiving two packets. When the receiver's window reaches the closed state, it sends an ACK to the parent, and the parent state can now go to open. Similarly, the receiver can go to the open state when its own children have sent ACKs. Note that the condition in which the receiver sends an ACK depends solely on whether it has received successfully the two packets.

W_o represents slot that are in used. We note that each occupied slot window also represents a buffer which is in use (for send/receive).

The root will initiate a multicast operation only if there is available slot in the sliding window. This is done to make sure all the of occupied window slots are already acknowledged and thus, the children have already received the data. When a slot is consumed, all processes in the communicator move the first slot in the free set to the occupied set as shown in Line 11-12. Even the root does this, as it may not be the root for subsequent operations, and this is the means used for flow control to avoid dropping data.

When the window's free space reaches a low water-mark, the reliability protocol is run to free up the occupied slots and identify buffers that may be recycled. The available buffers are moved to the free list, with an aggregated acknowledgment used to notify the parent. The aggregation is simple, instead of sending ACKs for all packets in the window, we can send the ACK for the last received

packet. However, if a specific packet is dropped, a NACK is sent for each dropped packet.

Due to a property of our algorithm (see Section 3.2.4), once a parent gets ahead of any of its children by more than some fixed value of acknowledgments – in other words the entire window is full – the parent stops initiating more multicasts, thus applying back pressure towards the root. Figure 2 demonstrates the window algorithm for $|W| = 2$ and for a sender and a child receiver.

3.2.3 Algorithm Completion Times. When all processes receive the multicast data, the broadcast completes in a single step, with $O(1)$ time.

PROOF. Since the acknowledgment is sent by all processes except the sender at the same time in step (1) in algorithm 1, every process should receive the acknowledgment when it reaches step (2) and can retire from the operation. $\square$

Note that there is no chain of dependency as a fan-in based algorithm but only independently between each parent/child pair. This result is the most important for InfiniBand based systems, as the bit error rate is very low. Dropped packets due to transport-level data errors are very rare.

On the other hand, if a packet is dropped we now show that: *It takes $O(h(T_b))$ steps to obtain missing data, where T_b is the sub-tree of the reliability tree, formed by processes that are missing data. $h(T_b)$ is the depth of this tree.*

PROOF. Processes that did not get the data are referred to as bad processes, and those that did, as good processes.

Consider a good process at the beginning of step (1), there must be at least one which is the root of the broadcast operation. This good process will finish step (1), and must have received either ACK or NACK from each of its children.

Suppose that some children are bad, in step (2) the process re-transmits the packet for these bad processes. Since this re-transmission is done using a reliable point-to-point protocol, a bad child becomes good after step (2). Thus it takes at most $h(T_b)$ steps to distribute the missing data. $\square$

3.2.4 Buffer management. Buffer management is generally handled at both the sender and receiver. On the receive side zero-copy algorithms may be used to avoid using temporary buffers. However, as mentioned above, the synchronization required to avoid dropped packets is too high, and also causes the algorithm to no longer be of fixed cost. Zero-copy at the send side however is beneficial, but in some instances it is still more efficient to copy the data into a temporary buffer with the data being sent from this buffer. This is discussed further in the evaluation section.

The amount of buffering is bounded as follows: *The maximum number of buffers required using the window algorithm with the reliability protocol is $h(T) \times |W|$, where $|W|$ is the window size and $h(T)$ is the tree depth.* This is equivalent to showing that there can be as much as $h(T) \times |W|$ pending single packet multicast messages.

PROOF. Suppose a process X has received $|W|$ packets from $|W|$ consecutive multicasts. It can release this window when it has received an ACK from all of its children. However it can happen that one of the children is pending due to lost packets or it has not

Figure 3: Different shape of the recovery tree with 5 nodes, rooted by 0, which has different performance characteristic (upper bound): s - number of status messages, b : buffer space, t : recovery time.

entered the multicast. Hence, the process cannot finish step (2) of the reliability protocol.

Consider the parent of X namely $p(X)$, it must have finished its own $|W|$ multicasts (since X must have sent the ACK) and may go on and perform $|W|$ more multicast before waiting for X to send ACKs. But recall that X is waiting for its children at this point to release its own previous $|W|$ multicast, hence there can be $2 \times |W|$ pending multicasts considering X and $p(X)$.

Similarly, the ancestors of X eventually become pending as long as X and its children are pending, and thus the result follows. $\square$

This fact is very useful to determine how many buffers a host has to maintain in order to always fill its QP with buffers and avoid dropping packets due to the lack of network buffering. With a tree with low depth, it is possible to achieve scalable memory management while maintaining high reliability.

3.3 Tree structure for multicast

As has been shown in the previous section, the structure of the virtual tree chosen to construct the reliability procedure affects algorithm performance. There are two main components:

- (1) *The branching factor:* the more children a process has, the more status messages that it has to wait on from its children.
- (2) *The depth:* in the case of dropped data, the depth of the tree affects how fast recovery may take place. Further, it affects the buffers space used by the window algorithm.

For example, one can choose a line-graph as the tree structure, i.e. each process has only a single child and a single parent and the number of status messages is two (a send and a receive). However, when data is dropped there could be as many as $O(N)$ the number of required window as well as a recovery time that may be of order N , where N the communicator size. Although such a structure could still be scalable from a latency perspective when dropped packet are rare, it does not scale well from a memory requirement perspective. Figure 3 illustrates the performance behavior of different tree types for 5 node trees.

A k -ary tree, on the other hand, scales better from a buffer space requirement and depth perspective, which is $O(\log N)$. However, it has a different issue which is how to handle multicasts from many different senders in a single window. In fact, a simple way is to keep track of all the possible parents, and all possible children present in each tree structure within the window. However, this can lead

to a $|W|$ number of status messages because there can be as many as $|W|$ multicast from different senders until the window is full.

The quest for finding an optimal tree structure is out of this paper’s scope. We resort to a binomial tree which gives us at worst a logarithmic number of status messages, number of buffers and recovery time. It also has a nice property that within a window, the number of parents and children is also bounded by the logarithm of the number of processes. That is, a process with rank X can only have parents/children with rank having 1-bit different from the binary representation form of X .

3.4 Algorithm Implementation

Two algorithms have been implemented, one to handle broadcast of smaller messages sizes, called the *Small Message Protocol*, and a second one for handling large messages which we call the *Large Message Protocol*.

Both protocols fragment the messages being broadcast, and use the hardware multicast to send each fragment. In both cases the data is received into a temporary buffer, and copied to the final application location. They also use the reliability protocol described above.

The algorithms differ in how they handle data at the sender. The small data algorithm always copies the data from the send buffer into a bounce buffer and sends the data from this temporary buffer. This is done to avoid the managing of a registration cache. Larger messages are sent directly from the user buffer, unless the data layout is such that it is more efficient to first pack the data into a temporary buffer and send from there rather than using the HCA’s gather capabilities. Since the data is sent from the user buffer, the sender has to wait for all of its children before returning to the user in case it has to perform recovery. Thus, large message protocol should only be more efficient when the data is fragmented into more than the window size as we shall see in the evaluation.

4 PERFORMANCE EVALUATION

The goal of this work is to develop broadcast algorithms that utilize Mellanox’s InfiniBand’s hardware capabilities that scale as $O(1)$ with respect to the size of the multicast group. The challenge is to provide such characteristics while adding minimal overhead for delivering the data to and from user buffers and providing a reliable delivery mechanism.

Since the focus for the work is on the characteristics of the network portion of the algorithm, the evaluation is done using only one process per node.

4.1 Experimental Setup

All tests were carried out on a 128-node cluster. Each node in the cluster consists of two 16-core Intel Broadwell CPUs running at 2.60 GHz with 40 MB of L3 cache per processor. Each node has 256 GB of memory. Each node is connected by multiple InfiniBand cards, however all tests were run across ConnectX-4 network cards. All nodes are running OFED 3.4-2.1.9.0 and RHEL 7.2. HPC-X 1.9 was used for the communications libraries for HCOLL results and as the basis for the new broadcast algorithms. HPC-X is Mellanox’s optimized communications stack and includes OpenMPI 2.1.2 as the basis for MPI. Command line options were utilized to pick

Figure 4: Eight byte short and long message algorithm broadcast latency for 2 to 128 nodes

a more optimized broadcast. They are: HCOLL_BCOL=ucx_p2p, HCOLL_SBGp=p2p, HCOLL_BCOL_P2P_MCAST_BCAST_ALG=1, and HCOLL_MCAST_NP=2. Intel MPI results were obtained with the Intel MPI Library 2017, Update 3. MVAPICH results were obtained with MVAPICH2 2.3a. The nodes are connected in a fat-tree with two levels. There are eight switches at the bottom level with each switch connecting sixteen nodes.

Experiments were run with 128 nodes except as noted. All benchmark results are from the OSU Micro-benchmarks suite [5].

In several of the figures comparing LRUM data to data from other implementations, data is not shown from all other implementations. The performance of the omitted data is much worse than that of the data collected using the LRUM algorithms, and as such the scale needed to display all the data in a single figure obscures the details of the better performing algorithms. Each such figure displays the best performing algorithms in fine enough resolution that the differences between the different algorithms are readily visible, discarding the data sets that obscure such a comparison.

4.2 Algorithm Performance

We evaluate in this section the two implementations described in Section 3.4: the short and long message protocols. A binomial tree reliability tree is used.

Figures 4, 5 and 6 show the latency of the broadcast algorithm as a function of number of hosts, for eight byte, 4096 byte and 4 MB size payloads, respectively. The data reported in these figures is the maximum latency across the full communicator rather than an average. This avoids the artificially low time reported by the root, which for small data completes as soon as the local user buffer can be reused. This is done to provide a more accurate view of broadcast completion as a function of process count. What these figures show is that the short message protocol performs best for the eight byte and 4096 byte broadcasts, but for larger messages such as at 4MB the large data algorithm performs best. The algorithm names reflect this fact.

In the experiments that were run, the cross over point for these algorithms was found to be in the range of 64KB to 128 KB. At that point, the cost of memory copy outweighs doing an extra reliability protocol at the end of the broadcast so that data can be sent from the user buffer.

An increase in latency is observed between 16 and 18 nodes because the multicast tree goes from including only one switch to

Figure 5: 4096 byte short and long message algorithm broadcast latency for 2 to 128 nodes

Figure 6: 4MB short and long message algorithm broadcast latency for 2 to 128 nodes

traversing two levels in the fat tree, thus increasing the latency for the farthest nodes.

To verify that the fixed latency as a function of process is achieved, we performed a linear fit of the maximum broadcast latency. For the eight byte and 4096 byte broadcasts, the slope of the fit is zero to the the accuracy of the reporting, which is 0.000 μ -seconds. The large data aggregation does show a small slope of 0.0017 μ -seconds which we attribute that to the cost of the last reliability protocol which increases slightly due to the use of the binomial tree. Nevertheless, our goal of fixed latency algorithm is achieved.

4.3 Broadcast and Point-to-Point Latency Comparison

The goal of this evaluation is to verify that the broadcast implementation can achieve the latency as low as point-to-point communication. And thus, our claim that the cost of reliability is small can be justified.

Figure 7 compares the point-to-point and 16 process small data broadcast latency for small message. The point-to-point communication used the UD protocol, and as such also uses a software protocol to ensure reliable transmission. With no software overheads, the latency of the two operations should be similar, which is indeed what the figure shows. In some instances the latency of the point-to-point communication is higher than that of the broadcast by a tiny fraction, but it likely due to the artifact of the software implementation.

Figure 8 compares the point-to-point and large data broadcast latency for large messages, starting at 128KB.

Figure 7: Comparison of the small data broadcast latency to the point to point latency. The broadcast group size is 16, with all hosts connected to the same switch. The processes used to measure the point-to-point latency are also connected to the same switch.

Figure 8: Comparison of the large data broadcast latency to the point to point latency. The broadcast group size is 16, with all hosts connected to the same switch. The processes used to measure the point-to-point latency are also connected to the same switch.

Software timing breakdown for the broadcast operation is provided in Figures 9, 10, and 11. The communication time includes the time for posting the multicast send operation to the network and the time to poll for completions and process them. The time labeled as reliability includes the time spent waiting for acknowledgments from children and initiating any required requests for data to be resent. The miscellaneous category includes the rest of the time, and is the difference between the measured local latency and the other two categories.

For eight and 4096 byte message sizes we see that the broadcast time at the root is considerably smaller than for other processes, whereas for the four MB message, these latencies are similar. In the first two cases the root only needs to copy the data into a bounce buffer, and when the window is full, wait on acknowledgments from its children, so the latency is expected to be smaller. The four MB broadcast requires the window to be recycled multiple times plus the extra reliability protocol at the end, thus not allowing the root to exit the broadcast operation earlier than the other processes.

It is also observed that the root spends more time in the reliability portion of the algorithm waiting for children to acknowledge their window, before being able to open a new window to initiate new broadcast operations. The time spent by the other processes in the

Figure 9: Eight byte 128 node broadcast time distribution, in μ -seconds, for the root and the average of the non-root processes.

Figure 10: 4KB 128 node broadcast time distribution, in μ -seconds, for the root and the average of the non-root processes.

Figure 11: 4MB 128 node broadcast time distribution, in μ -seconds, for the root and the average of the non-root processes.

broadcast operation is dominated by waiting on incoming data and processing such data.

4.4 Comparison With Other Implementations

As the data in the previous section shows, the performance of the short and long message algorithms are very good, being very close to that of the hardware multicast, which reflects the hardware limits. In this section the performance of the algorithms is compared with that of other implementations. The data is compared with the multicast based HPC-X implementation [12], IntelMPI [7] and the MVAPICH [13] default and multicast based implementations.

Figure 12: Small message broadcast latency as a function of message size for 128 nodes. Latency is reported in μ -seconds.

Figure 13: Large message broadcast latency as a function of message size for 128 nodes. Latency is reported in μ -seconds.

Figure 12 shows the broadcast latency of the small message implementation, HPC-X and MVAPICH's multicast based and MVAPICH's default algorithms. The result from the IntelMPI implementation are not presented as the latency is much higher than that of the other ones, though data from smaller process indicates this may be due to improper default algorithm selection. As the data shows, the new implementation is very comparable to HPC-X, typically within a few μ -seconds of HPC-X. Starting around 2048 bytes, the new small message protocol overtakes HPC-X. The new protocol is better than the MVAPICH multicast-based implementation (typically around 40% from 1-4096 bytes) and considerably better than the default, host-based, MVAPICH implementation (typically around 80% from 1-4096 bytes). At 4096, the small message protocol shows a maximum improvement over HPC-X of $0.61\mu s$ which is a speedup of 1.18x. At this message size, it also has the maximum improvement over MVAPICH of $2.62\mu s$ or 1.77x. At 1 byte, HPC-X is better than the small message protocol by $0.19\mu s$ which is about 13%. At 64 bytes, the minimum difference between MVAPICH and the small message protocol is observed - the small message protocol is faster by $0.94\mu s$ or about 1.5x.

Figure 13 shows the broadcast latency of the large message implementation, HPC-X, MVAPICH's multicast-based and default algorithms, and Intel MPI. The new large message algorithm is the best performing implementation starting around 16KB. The other implementation change their relative order at different message sizes. The large protocol averages a 50% improvement over HPC-X, 85% over Intel, and 70-80% over the MVAPICH algorithms. The difference between the large message protocol and the other

Figure 14: Eight byte broadcast latency as a function of communicator size, for the range of 2 to 128 processes. Latency is reported in μ -seconds.

Figure 15: 4096 byte broadcast latency as a function of communicator size, for the range of 2 to 128 processes. Latency is reported in μ -seconds.

broadcast implementations scales with message size, starting at 16KB. At 16KB, the large message protocol outperforms HPC-X by $0.83\mu s$ (1.1x), Intel by $535\mu s$ (over 80x), and MVAPICH’s multicast broadcast by $0.95\mu s$ (1.4x). At 16MB, the differences are much larger - $2550\mu s$ improvement over HPC-X, or about 2.3x, $2181\mu s$ improvement over Intel MPI which is about 2.1x, and $6350\mu s$ improvement over MVAPICH’s multicast broadcast which is a 4.3x improvement.

Figure 14 shows the eight byte broadcast latency as a function of communicator size. The data displayed is for the new small message algorithm, HPC-X and the MVAPICH multicast based method. The latencies obtained using IntelMPI and host-based MVAPICH algorithms are not displayed, as their latency is considerably higher than that of these three. The HPC-X multicast best algorithm shows the best performance, being slightly better than that using the new multicast algorithm. Both these perform quite a bit better than the MVAPICH implementation.

Figure 15 shows the 4096 byte broadcast latency as a function of communicator size. The data displayed is for the new small message algorithm, HPC-X and the MVAPICH multicast based method. The latencies obtained using IntelMPI and host-based MVAPICH algorithms are not displayed, as their latency is considerably higher than that of these three. The small data algorithm shows the best performance, being better than that using the HPC-X multicast based algorithm. Both these perform quite a bit better than the MVAPICH implementation. The small data algorithm is roughly 15% faster than HPC-X at this message size.

Figure 16: 4 MB broadcast latency as a function of communicator size, for the range of 2 to 128 processes. Latency is reported in μ -seconds.

Figure 17: Small message broadcast latency when introducing random delays entering the algorithm. Communicator size is 128. Latency is reported in μ -seconds.

Figure 16 shows the 4MB broadcast latency as a function of communicator size. The data displayed is for the new large message algorithm, HPC-X and the MVAPICH multicast based method. The latencies obtained using IntelMPI and host-based MVAPICH algorithms are not displayed, as their latency is considerably higher than that of these three. The large data algorithm shows the best performance, being over a factor of two better than that using the HPC-X multicast based algorithm. Both these perform considerably better than the MVAPICH implementation.

4.5 Broadcast Latency Under Load Imbalance

Broadcast latency is measured with a random delay at each process from 0 – 100 microseconds before entering the broadcast algorithm. This benchmark is introduced to simulate application load imbalance. The intent of the experiments is to assess the algorithms’ ability to handle such imbalance.

Figure 17 shows the broadcast latency for small messages for a communicator of size 128, for the new small data algorithm, HPC-X and the default host-based MVAPICH algorithm. The IntelMPI algorithm and the MVAPICH multicast algorithms show much higher latency, and thus are not displayed in this figure. All three implementations have similar performance, dominated by the latency delays introduced. The new small data algorithm performs slightly better than the other two algorithms, with the HPC-X algorithm performing a bit better than the MVAPICH implementation. The small data algorithm and the MVAPICH algorithms have similarly

Figure 18: Large message broadcast latency when introducing random delays entering the algorithm. Communicator size is 128. Latency is reported in μ -seconds.

Figure 19: Broadcast streaming latency for small message sizes. Communicator size is 112 processes.

smooth behavior, with the HPC-X based algorithm having larger swings in latency. At eight bytes, the small message protocol is $1\mu s$ faster than HPC-X and about $0.9\mu s$ faster than MVAPICH. At 4096 bytes, the small message protocol is $1.5\mu s$ faster than HPC-X and about $2\mu s$ faster than MVAPICH.

Figure 18 shows the broadcast latency for large messages for a communicator of size 128, for the new large data algorithm, HPC-X, IntelMPI and the default host-based MVAPICH algorithm. The MVAPICH multicast algorithm shows much higher latency, and thus is not displayed in this figure. The new large data implementation is impacted much less than that of the other implementations. At 4MB messages, the large message protocol is significantly faster than the other algorithms - 51% faster than HPC-X, 72% faster than MVAPICH and 54% faster than Intel MPI.

4.6 Broadcast Throughput

Broadcast throughput is measured with a modified version of the OSU broadcast benchmark. Normally, the benchmark will iterate over some number of iterations for a given message size (default of 100 for large broadcasts or 1000 for small broadcasts). After every call to `MPI_Bcast`, there is a call to `MPI_Barrier`. To simulate streaming broadcasts and measure streaming latency and broadcast throughput, the barrier between every iteration of a given message size was eliminated. Note: There is still a barrier after all iterations of a given message size. Only 112 nodes of the cluster were available at the time of testing.

Figure 20: Broadcast throughput as a function of message size. Higher rates are better. Communicator size is 112 processes.

Figure 19 shows the broadcast streaming latency. Measurements are shown for Intel MPI, MVAPICH multicast, HPC-X, and the small message protocol. The message range plotted is two through 1024 byte messages. The data for one byte broadcast is omitted due to a performance anomaly in one of the implementations being used for comparison, resulting in unusually high latencies.

Figure 20 shows the broadcast throughput (in number of completed broadcasts per second) as a function of message size for Intel MPI, MVAPICH multicast, HPC-X, and the new protocols (with a 128kb short-to-large protocol transition point).

As figures 19 and 20 show, the new protocols exhibit the best performance, being better than the MVAPICH implementation by a factor of two at small message size and a factor of four at larger sizes. Performance is also better than that of HPC-X's broadcast implementation by at least a factor of two across all message sizes. The new protocols outperform IntelMPI's implementation by a factor of six to twelve.

5 RELATED WORK

Broadcast algorithms based on Mellanox's hardware multicast capabilities have been developed before. Liu et al. [10] maintains a tree such that the leaf of the trees are the majority of the processes and multicast is only used for the last level of the tree. This makes the reliability protocol simpler but less efficient since data is still transmitting with a logarithmic algorithm among the non-leaf processes.

Hoeffler et al. [9] also used a reliability protocol that is independent of the multicast group size, however they introduced arbitrary additional latency to try and avoid dropped packets. They reported about 50% dropped packets due to application load imbalance. A ring is used for the reliability protocol, with each process receiving the data being a source of such data, and the data traveling only as far as needed for reliability purposes around the ring.

The LRUM approach has several advantages over Hoeffler's approach. The buffering scheme allows for data to be sent as soon as the root initiates the broadcast operation, with the transmission slowed only when temporary buffers are unavailable. Such back pressure may be applied when one of the receivers arrives late to the operation when available buffer space is low, or when some of the receivers are slow for some reason. The algorithm itself does not lead to dropped data because of lack of receive buffers. With the

alternative case in the best case data delivery is delayed at least by the time it takes the selected receiver to notify the root it is ready for the data. In addition, the mechanism for obtaining missing data scales logarithmically in the size of the group, whereas the alternative approach is linear. In the common case where the multicast data is delivered to all the receivers, the cost of these reliability protocols are the same, since both the logarithmic and linear protocols are local, with communication only with near neighbors. It is when data is dropped that the LRUM reliability protocol performs better. Finally, the buffering scheme allows for the root to continue sending data without it being dropped due to unavailable buffers even when only part of the receivers have joined the broadcast operation.

It is interesting to note that Hoefler’s algorithm is somewhat equivalent to our protocol using a line-graph approach – except that we can avoid the short-coming of always re-transmitting the entire data over the reliable channel. The LRUM scheme also allows the depth of the tree to be changed, thus allowing for trade-offs in the cost of the ACK/NACK protocol versus the depth of the reliability tree, which determines both buffer costs as well as worst case recovery costs.

Multicast algorithms have also been explored in other contexts, surveyed in [11] and also implemented in MPICH [8] for TCP/IP. In these works, a similar sliding window protocol is used to keep packets for possible retransmission. However, in most protocols, the NACK status is either multicast to everyone in the network/group, or sent to a reliability service (which can be implemented as a dedicated process) to request lost data.

Among the multicast protocols used the closest to our reliability protocol is the Tree-Based Multicast Transport Protocol [14] which is designed for IP multicast. There are, however, several differences that make this algorithm unsuitable for use as the underlying mechanism supporting an MPI_Bcast() implementation in a high-performance-computing setting, and when used with InfiniBand hardware. This protocol created a k -ary tree of *Domain Managers* (DMs) which are service entities different from the end-points that are intended to receive the data. Each DM is typically associated with a network subnet and responsible for reliability of its members which are the actual application nodes. The reliability of the DM itself is maintained by the immediate DM parent. The use of a DM is unnecessary, and when used in an MPI implementation would require the creation of additional processes that are not part of the MPI process group. As a separate process, the DMs would require compute resources beyond those needed for the MPI processes, to avoid computing for the resources used by the MPI processes. The original method also uses multicast to send NACKs and retransmit data, but with the limit on InfiniBand multicast groups, and the association of such groups with MPI communicators, the LRUM design opts to use a single multicast group per communicator. Compared to this protocol, our algorithm is more generic such that it could handle multicasting from many different senders without the need of re-initiating the tree structure (which is noted with an overhead of $O(\log N)$ in the paper). This ability is important in our High Performance computing context, where the cost of setting up a different tree for each different root used would eliminate much of the benefit of using the hardware multicast capabilities.

6 DISCUSSION AND CONCLUSIONS

The goal of the work described in this paper is to develop broadcast algorithms with scalability similar to that of the underlying hardware multicast capabilities supported in the Mellanox InfiniBand switches. In addition, the intent has been to also have these algorithms have a worst case latency similar to that of the point-to-point latency between the root and the process on the furthest away node from the root which reflects the performance of the underlying hardware multicast support. The primary challenge to meet these goals is that the hardware multicast support provides an unreliable data delivery mechanism, whereas the broadcast algorithm is required to be reliable. Therefore, the challenge is to develop reliability algorithms that do not depend on the size of the multicast group, and which add a minimal amount of overhead on top of the data transport time.

Avoiding a reliability protocol with a communicator size dependence requires an acknowledgment scheme with behavior that is independent of system size. The overheads added on top of the hardware multicast costs include the costs associated with the reliability protocol. On top of the costs associated with the ACK/NACK protocol, the protocol should minimize the chance of reaching a state where receive buffers are not available when data arrives at a receiver, therefore causing data retransmission. In addition the costs associated with using the underlying UD transport, which does not support RDMA operations, pose a challenge to minimize the costs of delivering the data to the user buffer at the target processes.

As section 4 shows both goals have been achieved, with the latency of the broadcast operations being similar to that of the point-to-point latency, and the small (eight byte) and medium (4096 byte) latencies being essentially constant for the measurements made up to 128 nodes, and that of the large data (four MB) having a weak dependency on node count (slope of 0.0017 μ -sec/process). In addition, broadcast throughput is extremely good, beating other implementations by at least a factor of two.

A key to achieving such scalability is assuming data is delivered correctly most of the time because of the low bit error rate on the underlying InfiniBand hardware and having a slightly more costly re-transmit protocol. The reliability protocol used is local, with acknowledgments being sent only to the parent, once the reliability window can be recycled. Local buffers holding the data copies are retained only until acknowledgments are received from all children. This allows for local knowledge only, independent of system size, to free the local window, and in a situation where broadcast operations are being issued at a high rate by all processes in the group, allows for continuous flow of data. In the event of back pressure resulting from children completing the broadcast operations slowly, and therefore the parent not initiating any new broadcast operations until the children complete their operations, the buffering scheme described in section 3 ensures sufficient buffer to avoid dropped packets. These properties are proven and shown in the evaluation section.

Finally, the performance of the new algorithms under load imbalance situations, which are common application use-case scenarios, was shown to be better than that of the other implementations tested. This is the case for performance as well as for algorithm stability.

The retransmission protocol is optimized for the low hardware bit error rate. A request for missing data being propagated up the reliability tree to the first node that has the data, rather than to the root of the operation, with the data being sent reliably down the tree. The protocol is designed this way under the assumption that such requests are rare. It avoids having the incast problem associated with all a single process serving as the agent reliability agent. The use of the reliability tree where requests for data are propagated up the tree only up to the first tree node that has the needed data, provides a logarithmically scaling scalable means to obtain missing data. This approach has the advantage of having an essentially fixed low-cost for the common case where no data is dropped, and a single request up the tree is sufficient to have the data be sent down the tree as far as is needed only once, satisfying all requests for such data. In addition, the comparison with other broadcast implementations show that this implementation is better than most other implementation tested. The one exception is the multicast based implementation used in HPC-X, for messages smaller than 512 bytes - where the two implementations are in a virtual tie.

Current work has focused on achieving MPI_Bcast() performance and scalability that is commensurate with that underlying hardware. This covers the network portion of the platform, which is the portion of the algorithm that grows with communicator size. Future work aims to includes to add the local-host portion of the algorithm, while maintaining the constant algorithm cost for a fixed number of local MPI processes.

REFERENCES

- [1] [n. d.]. Co-Array Fortran. “<http://caf.rice.edu/>”. ([n. d.]).
- [2] [n. d.]. Infiniband Verbs. “<http://www.infinibanda.org/>”. ([n. d.]).
- [3] [n. d.]. MPI Standard. “<http://www.mpi-forum.org/>”. ([n. d.]).
- [4] [n. d.]. OpenSHMEM. “<http://www.openshmem.org/>”. ([n. d.]).
- [5] [n. d.]. OSU Micro-benchmarks. “<http://mvapich.cse.ohio-state.edu/benchmarks/>”. ([n. d.]).
- [6] [n. d.]. UPC. “<http://upc.lbl.gov/>”. ([n. d.]).
- [7] Intel Corporation. [n. d.]. Intel MPI. “<https://software.intel.com/en-us/intel-mpi-library>”. ([n. d.]).
- [8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.
- [9] Torsten Hoefler, Christian Siebert, and Wolfgang Rehm. 2007. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 1–8.
- [10] Jiuxing Liu, Amith R Mamidala, and Dhabaleswar K Panda. 2004. Fast and scalable MPI-level broadcast using InfiniBand’s hardware multicast support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 10.
- [11] Katia Obraczka. 1998. Multicast transport protocols: a survey and taxonomy. *IEEE Communications magazine* 36, 1 (1998), 94–102.
- [12] Mellanox Technology. [n. d.]. HPC-X. “http://www.mellanox.com/page/products_dyn?product_family=189&mtag=hpc-x”. ([n. d.]).
- [13] The Ohio State University. [n. d.]. MVAPICH2. “<http://mvapich.cse.ohio-state.edu/>”. ([n. d.]).
- [14] Rajendra Yavatkar, James Griffioen, and Madhu Sudan. 1995. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of the third ACM international conference on Multimedia*. ACM, 333–344.