Efficient and Scalable All-to-All Personalized Exchange for InfiniBand-based Clusters *

Sayantan Sur Hyun-Wook Jin Dhabaleswar K Panda

Computer and Information Science,
The Ohio State University,
2015 Neil Avenue,
Columbus, OH 43210
{surs,jinhy,panda}@cis.ohio-state.edu

Abstract

The All to All Personalized Exchange is the most dense collective communication function offered by the MPI specification. The operation involves every process sending a different message to all other participating processes. This collective operation is essential for many parallel scientific applications. With increasing system and message sizes, it becomes challenging to offer a fast, scalable and efficient implementation of this operation. Infiniband is an emerging modern interconnect. It offers very low latency, high bandwidth and one-sided operations like RDMA write. Its advanced features like RDMA write gather allow us to implement traditional Hypercube algorithms much more efficiently than in the past. Our aim in this paper is to evaluate traditional personalized exchange algorithms in the context of InfiniBand and propose efficient and scalable implementations of them. Performance evaluation of our design and implementation reveals that it is able to reduce the All-to-All communication time by upto a factor of 3 and perform upto 77% better on clusters with thousands of nodes.

1 Introduction

Cluster based computing systems are becoming popular for a wide range of scientific applications, owing to their cost-effectiveness. These systems are typically built from commodity PCs connected with high speed Local Area Networks (LANs) or System Area Networks (SANs). A majority of these scientific applications are written on top of the Message Passing Interface (MPI) [14, 4]. MPI provides both *point-to-point* and *collective* communication functions. Many parallel applications use collective communication functions to communicate either with all participating processes or a subset of them. Many applications, such as IS and FT in the NAS Parallel Benchmark suite [2], almost use collective operations exclusively for communication. Thus, providing high performance and scalable collective communication support is critical for cluster based systems. One of the most important collectives is `MPI_Alltoall`. In `MPI_Alltoall`, every process sends distinct data to each of the other processes. As the number of processes participating in the `MPI_Alltoall` or the message sizes becomes larger, the All-to-All communication cost rapidly increases. Thus, It is a challenge to provide an efficient implementation of this collective. In this paper we focus on the design and implmentation issues regarding `MPI_Alltoall` for InfiniBand-based Clusters.

The current implementation[17] of the MPI collective functions, including `MPI_Alltoall` is, based on MPI level point-to-point operations. Thus, the collective operations are abstracted onto point-to-point primitives. This approach cannot fully utilize the advanced features offered by contemporary interconnects. There are several modern network interconnects that provide very low latency (less that 10μs) and high bandwidth (in the order of Gbps) for cluster based systems. Two of the leading products are Myrinet [3] and Quadrics [13]. Recently InfiniBand [1] has entered the high performance computing market. InfiniBand offers a new model of data transport based on memory semantics. This operation is called *Remote Direct*

*This research is supported in part by Department of Energy's grant #DE-FC02-01ER25506 and National Science Foundation's grants #CCR-0204429 and #CCR-0311542

Memory Access (RDMA). It allows transfer of data directly between user level buffers on remote nodes without the active participation of the receiver. This is a one-sided operation that does not incur a software overhead at the remote side. InfiniBand also allows Gather/Scatter RDMA operations.

In addition, the current MPI implementation[17] of `MPI_Alltoall` considers only the message size to decide the algorithm being used. However, we note that the total time taken for the All-to-All communication depends on two parameters - the message size and the number of processes participating in the exchange.

In this paper, we aim to provide answers to the following questions :

1. Can we implement the All-to-All communication in the most efficient manner, so as to fully utilize the benefits of RDMA, Gather/Scatter RDMA and other features of InfiniBand?
2. Can we come up with a scalable set of algorithms that perform the best for any given message and system size?

This paper shows that replacing the point-to-point communication calls in the collective operations with faster lower-level primitives can provide significant performance gains. Our approach makes it possible to reduce the number of data copies, number of memory registration operations and other software overheads. We propose a Hypercube RDMA Gather based scheme (HRWG) for optimizing the All-to-All communication time for small messages and a Direct Eager scheme (DE) for achieving better performance for larger messages. In addition, we provide performance analysis and estimate the performance of our design and implementation on large scale clusters.

The designs for `MPI_Alltoall` were implemented and integrated into the MVAPICH[11] implementation of MPI over InfiniBand. MVAPICH is an implementation of Abstract Device Interface (ADI) for MPICH[5]. MVAPICH is derived from MVICH[10] from Lawrence Berkeley National Laboratory. Compared to the current `MPI_Alltoall` implementation, our new designs can improve the latency by a factor of upto 3.07. In addition, our performance analysis and estimation show a benefit of upto 77% for an All-to-All exchange in a 1k node cluster.

The rest of the paper is organized as follows: In section 2, we provide an overview of the InfiniBand architecture. In section 3, we provide background on `MPI_Alltoall` and existing algorithms. In section 4, we describe the limitations of implementing collective operations using MPI level point-to-point functions. In section 5, we propose our designs for RDMA based `MPI_Alltoall`. We evaluate our designs using experiments and analytical models in section 6. Conclusions and future research directions are presented in Section 7.

2 InfiniBand Architecture Overview

The InfiniBand Architecture[1] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes, and TCAs sit on I/O nodes. 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 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 the Work Queue Requests (WQR), or descriptors, and submitted to the work queue. the completion of WQRs is reported through Completion Queues (CQs). Once a work queue element is finished, a completion entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished. InfiniBand also supports different classes of transport services. In current products, Reliable Connection (RC) service and Unreliable Datagram (UD) services 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. Write Gather and Read Scatter are supported in RDMA operations. RDMA write operation can gather multiple data segments together and write all data into a contiguous buffer at the receiver end. Gather/scatter features are very useful to transfer non-contiguous data. The Gather/Scatter facility not only reduces the startup costs, but also increases network utilization. RDMA Write with Immediate data is also

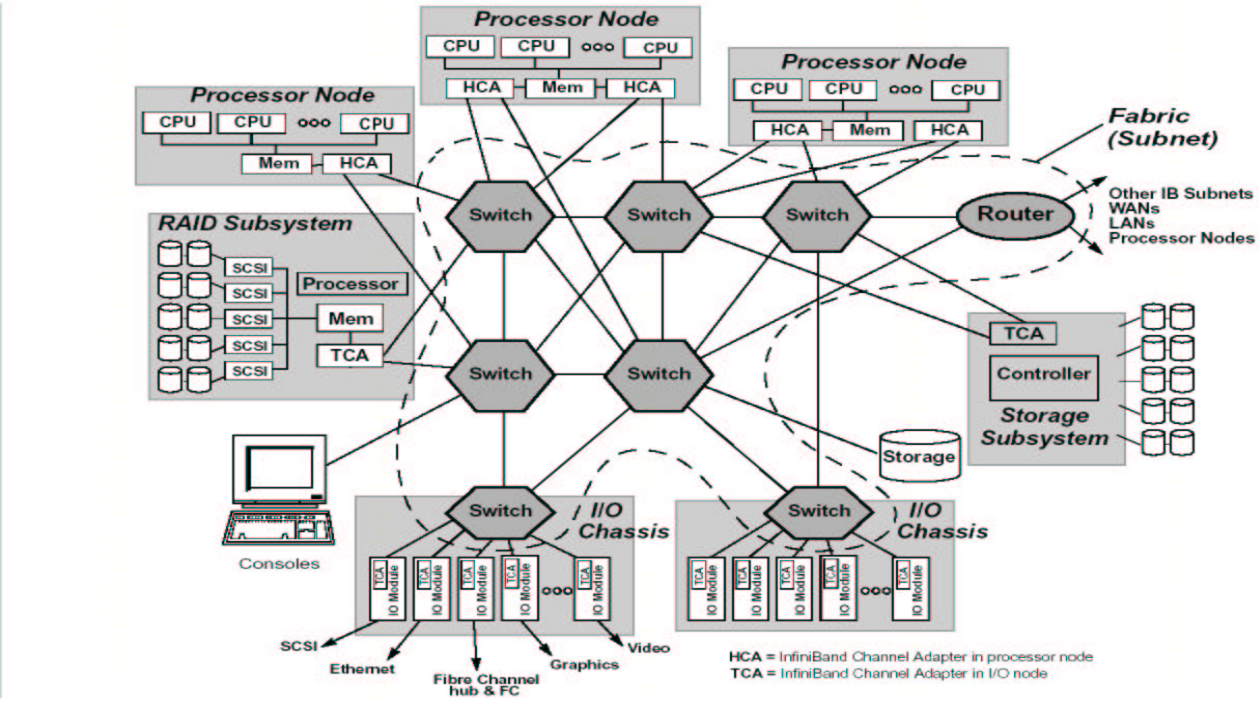

Figure 1. InfiniBand Architecture overview (courtesy IBTA)

supported. With Immediate data, a RDMA Write operation consumes a receive descriptor and then can generate a completion entry to notify the remote node of the completion of the RDMA Write operation.

3 MPI_Alltoall Overview and Existing Algorithms

In this section, we provide a brief overview of the `MPI_Alltoall` collective function. We also decribe the existing algorithms used in implementing `MPI_Alltoall` and their cost models.

3.1 MPI_Alltoall Overview

MPI supports both point-to-point and collective communication functions. `MPI_Alltoall` is a commonly used collective for achieving a complete exchange of data among all participating processes.

`MPI_Alltoall` is a blocking operation. The call does not return until the communication buffer can be reused. `MPI_Alltoall` is used when all the processes have a fixed length of message to send to each of the other processes. The jth block of data sent from process i is received by process j and placed in the ith block of the receive buffer. The process is explained by Figure 2.

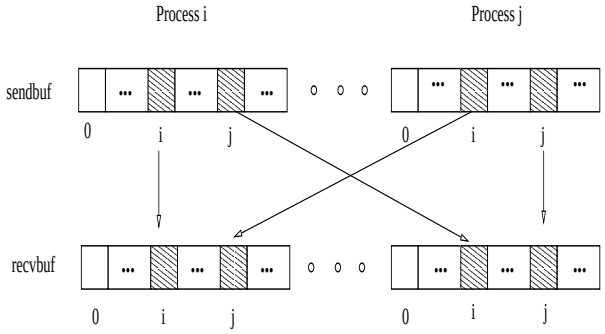

Figure 2. MPI_Alltoall

3.2 Existing Algorithms

In this section we provide an overview of existing algorithms and their cost models.

3.2.1 Hypercube / Combining Algorithm

A hypercube is a multidimensional mesh of nodes with exactly two nodes in each dimension. A d-dimensional hypercube consists of $p = 2^d$ nodes. Figure 3 shows a 3 dimensional hypercube.

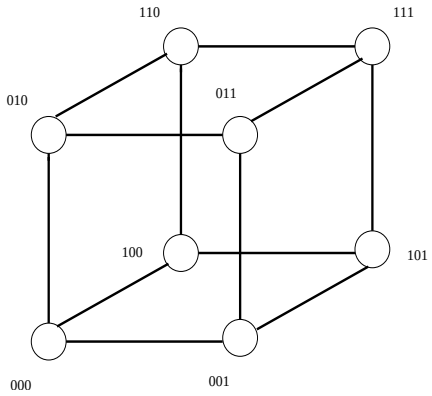

Figure 3. A 3-Dimensional Hypercube

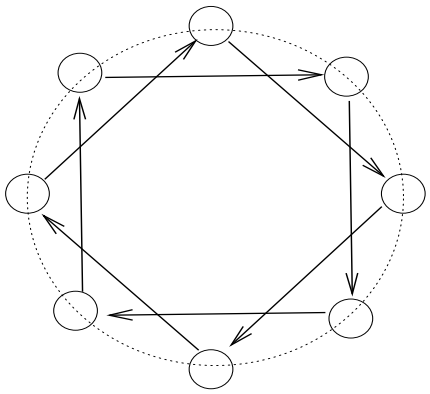

Figure 4. A Virtual Ring

The All-to-All personalized communication algorithm for a p-node hypercube with store-forward routing is an extension of the two-dimensional mesh algorithm to $\log p$ steps. Pairs of nodes exchange data in a different dimension in each step. In a p node hypercube, there are a set of $p/2$ links in the same dimension connecting two subcubes of $p/2$ nodes each. At any stage in All-to-All personalized communication, every node holds p packets of m bytes each. While communicating in a particular dimension, every node sends $p/2$ of these packets (consolidated as one message, or as multiple messages). Thus, $mp/2$ bytes of data are exchanged along the bidirectional channels in each of the $\log p$ iterations. The resulting total communication time is,

$$T_{hypercube} = (t_s + 1/2t_w mp)logp \qquad (1)$$

Where,

t_s = Message startup time
t_w = Time to transfer one byte
m = Message size in bytes
p = Number of processes

3.2.2 Direct Virtual Ring

The direct algorithm is a straightforward way to exchange messages among all the processes. It assumes that all the processes have direct links connecting them. The processes are arranged in a virtual ring. Each process then sends its message to its neighbour. To avoid all the processes from sending to a single destination, the destinations are scattered among all of them, by using a modulus operation. Process $rank$ sends its message for process $(rank + i)\%p$, $\forall i, (0 \leq i \leq p)$. Figure 4 demonstrates such a virtual ring topology.

So, at every step, each process sends m bytes of data and it does it for $(p-1)$ steps. Thus, the total time for an All-to-All exchange is,

$$T_{direct-ring} = (p-1)t_s + t_w m(p-1) \qquad (2)$$

4 Limitations of Current MPI_Alltoall

In this section we analyze in detail, the problems associated with implementing the `MPI_Alltoall` collective by using MPI level point-to-point communication functions.

4.1 Overview of the Current Implementation

The current implementation of `MPI_Alltoall` uses MPI level point to point operations. This adds some overhead because the call has to pass through many layers before the actual communication takes place. Figure 5 illustrates this.

4.2 Point-to-Point Messages

In this section, we will analyze the various cost components of point-to-point messages for the MPI implementation[11]. We will highlight the limitations of using MPI point-to-point functions for implementing collectives.

4.2.1 Small Messages

In the current implementation of MVAPICH, small point-to-point messages are transferred using RDMA. However, in order to achieve the RDMA without having to pay the on-the-fly registration cost and the address exchange cost, the messages are copied over to a pre-registered buffer[12]. The All-to-All communication for small messages is often implemented by using a

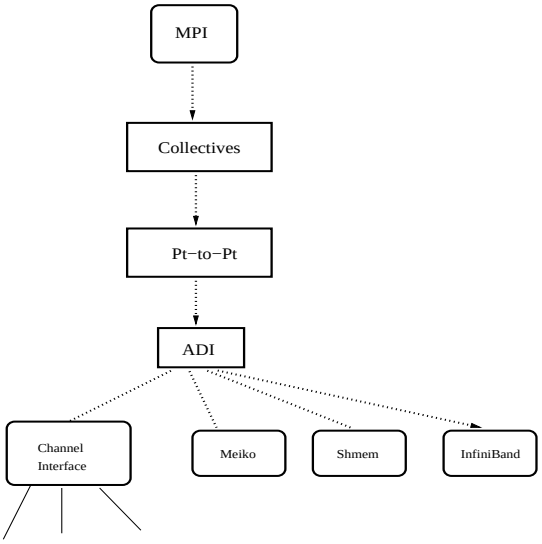

Figure 5. Implementation path for current collectives in MPICH

combining algorithm like Hypercube or Recursive Doubling. The current implementation of `MPI_Alltoall` is based on Recursive Doubling. This necessitates another copy from the user buffer to the MPI level buffer. So, in total, there are 4 copies for sending a single message across as shown in Figure 6.

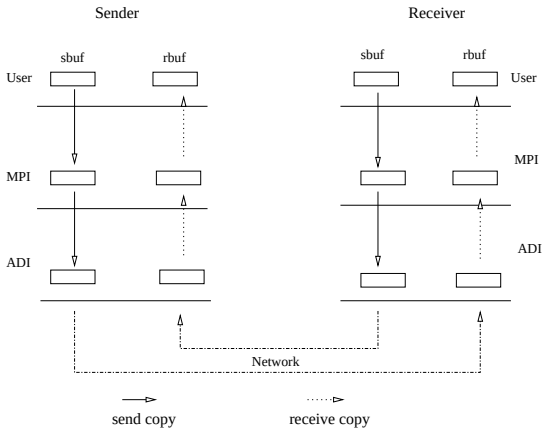

Figure 6. MPI_Alltoall Message Path for Small Messages in current MPI implementation

The copy cost rises rapidly as the message size and the system size increase. This means that combining algorithms like Hypercube or Recursive Doubling cannot be efficiently implemented using MPI level point-to-point functions.

4.2.2 Large Messages

In the current implementation of MVAPICH, the MPI point-to-point communication for large messages undergoes a *rendezvous* protocol as shown in Figure 7. The sender registers the application buffer and sends a *rendezvous_start* message to the receiver. The receiver then tries to register the application receive buffer and if successful, sends a *rendezvous_ok* message to the sender along with the virtual address and the memory handle of the buffer. The sender then sends the data, followed by an ACK.

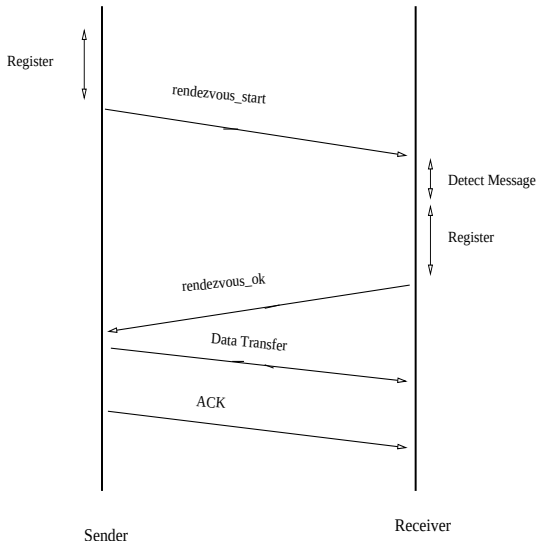

Figure 7. The Rendezvous Protocol

On current generation InfiniBand hardware, the address exchange phase costs around 10μs. The sender cannot send the message until the receiver replies back to the sender with the virtual address and memory handles. When implementing high communication density collectives like `MPI_Alltoall`, this presents a critical *bottleneck* during each step of the collective algorithm. Thus, as the system size increases, the total cost due to the *rendezvous* protocol increases linearly.

The point-to-point implementation requires individual memory registration of the sender and recieve buffer per message. In collective communication, often times the same contiguous buffer is used as the sender buffer. The individual messages are specified as offsets from the beginning of this buffer. The multiple registration operations can be coalesced into one for improving performance. However, if the collectives are implemented using MPI point-to-point functions, registration coalescing is no longer possible.

In order to optimize the registration operation, MVAPICH maintains a registration cache. Buffers which are registered once are not de-registered im-

mediately, instead they are kept in a cache. As the number of registration operations increase, the cache fills up gradually, and some cache evictions may occur. This is particularly important for long running MPI applications. Using MPI point-to-point operations for implementing collectives needlessly generates a lot of cache entries due to individual buffer registrations.

The current implementation of the `MPI_Alltoall` collective for large messages uses a Direct Virtual Ring algorithm.

4.3 Performance Limitations of the Current Implementation

Figure 8 shows the performance of the current MVAPICH implementation for `MPI_Alltoall` exchange for 4, 8 and 16 processes. We observe that with increasing system and message size the total cost of the All-to-All communication increases sharply.

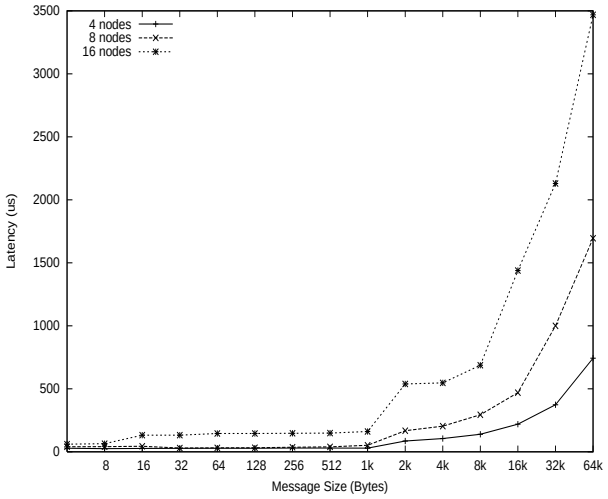

Figure 8. MPI_Alltoall Exchange Latency

5 Proposed RDMA based MPI_Alltoall Design

In this section we will take a detailed look into our designs, which fully utilize the RDMA Write and RDMA Write with gather for implementing `MPI_Alltoall`.

5.1 Overall Approach

In order to fully utilize the features provided by InfiniBand, we directly implement the collective operations on the InfiniBand Verbs Layer. Also, by directly implementing the collectives over the Verbs layer, we avoid software overheads such as copying and multiple registrations. The proposed implementation path is shown in Figure 9.

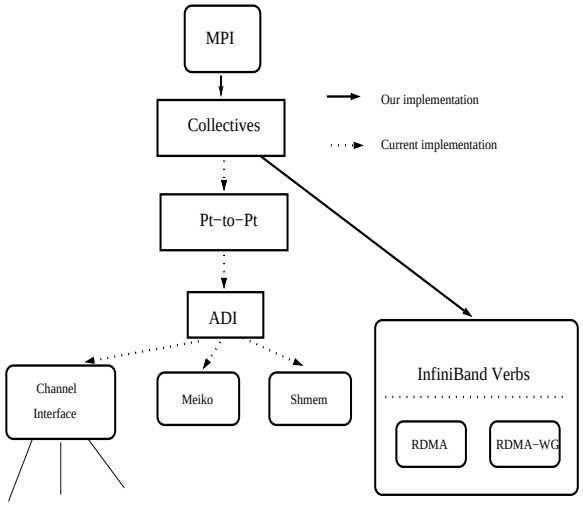

Figure 9. Proposed implementation path for current collectives

5.2 RDMA Based Design Issues

Before we can directly utilize the benefits provided by RDMA for implementing `MPI_Alltoall`, a few difficulties must be addressed[8] :

Memory Registration and Address Exchange: There can be two design choices, either we copy over the message at the beginning of the All-to-All communication to specific pre-registered buffers, or we can perform an address exchange after registering both sender and receiver buffers. The registration operation is costly and is not feasible for smaller message sizes. Moreover, on current generation InfiniBand hardware, the address exchange phase costs around 10μs. On the other hand the copy-based approach avoids on-the-fly registration and address exchange costs. However, the cost for copying large sized buffers can be prohibitively high.

Message Arrival Detection: The RDMA Write operation in InfiniBand is totally transparent to the receiver process. The only way the receiver process can make out whether data has really arrived in the buffers is by polling the contents of a specific pre-defined memory location. For achieving this, there needs to be a persistent association of buffers at the sender and receiver end. Achieving a persistent association is relatively simple when pre-registered buffers are used. All the processes can implicitly decide on start and end buffer locations. Prior to the All-to-All communication

the processes can reset the last bytes of the persistent buffers. Marking the completion of a RDMA write for direct application buffers, which are registered on-the-fly, is impossible using the earlier technique. Here, there is no unique value of the memory location that the MPI implementation can poll on. The application may choose to send any data value of its choice. Usually, in such a case, completion of a RDMA write operation can only be determined by using an explicit ack. Hence, specific buffers need to be provided for collecting acks.

5.3 Design for Small Messages: HRWG

For small messages, the message transfer startup time dominates the total cost of the operation. The message transfer startup time for the Hypercube algorithm is $t_s \log p$ and for that of the Direct Virtual Ring algorithm is $t_s(p-1)$. InfiniBand has very high bandwidth availability (850 MB/s), so the cost of transferring the data $t_w m$ for a small message is comparably less. So, our natural choice for smaller messages is the Hypercube algorithm. We implement the Hypercube algorithm using the RDMA Write Gather feature provided by InfiniBand. Hence, we call this scheme as HRWG.

Now we look closely at the startup time cost t_s. There are various costs associated with message startup based on the implementation mechanism. If we decide to have a zero copy implementation, then the address exchange phase will dominate the message startup time. Also, we would have to pay on-the-fly registration cost. This can be comparably costlier than copying the data over to a pre-registered buffer. Hence, we choose a copy-based approach, implementing the Hypercube algorithm for small message sizes. However, our copy-based mechanism has only 2 data copies bypassing the MPI-level buffer in Figure 6 instead of the 4 copies in the point-to-point based implementation.

Buffer Management: We implement a buffer management scheme for the copy-based approach. In order to detect the arrival of an incoming message, we use memory polling. In our implementation, the collective communication buffer is created during the communicator initialization time. All processes in the communicator need to exchange addresses and memory handles for remote buffers. The collective communication buffer can then be divided into several parts. This can be done implicitly by all processes. There are two divisions of the buffer for supporting back-to-back collective calls. Also, we set up persistent associations with our $\log p$ neighbours of the hypercube. Figure 10 shows how these buffers are arranged.

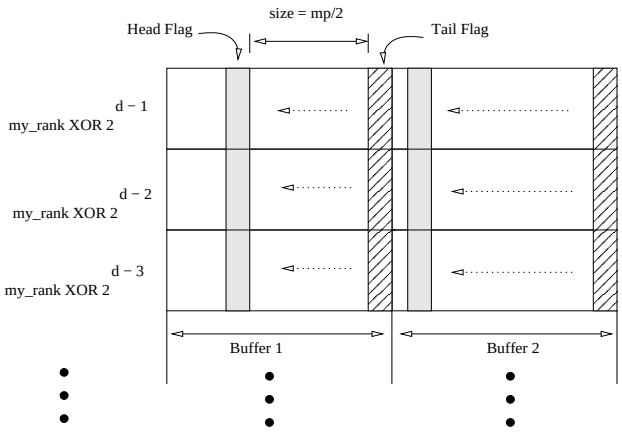

Figure 10. Buffer arrangement for Hypercube Algorithm

The collective buffer is zeroed at the beginning. At the start of one All-to-All operation, the *tail* flags of the other buffer (for all of the $\log p$ peers) are cleared. However, we need to make sure that the flag cannot be set before the data is delivered. And to do this, we need to use some knowledge about the implementation of the hardware. In our current platform, data is delivered in order (the last byte is written last). Thus, the arrival of the tail flag does ensure that the entire message has arrived.

If a process is involved in an All-to-All operation and is still waiting for its completion, another process might have entered into a successive, back-to-back All-to-All. We must guarantee that the buffer space provided for an All-to-All operation will not be over-written until it is safe to do so. We observe that providing buffering for two back-to-back All-to-Alls is sufficient to make such a gurantee.

For implementing the Hypercube algorithm, we have to keep track of the buffers in transit and forward them correctly to the next dimension across which communication will take place. We can keep track of the buffers to send, by simply observing a pattern of communication within the hypercube. At every step of communication in the hypercube, a process sends $p/2$ messages. The number of contiguous buffers from the received buffer-pool per peer is, $2^{(d-(d-i))}$, $\forall i$ in 0 to $(d-1)$. The buffer to be forwarded can be easily found out by maintaining a set of pointers at the middle of the receive buffers. The number of *middle* pointers is equal to the number of contiguous buffers to be forwarded from the persistent buffer associated with that dimension. After each iteration, the middle pointers can be moved either backwards or forwards depending on whether the rank of the destination process is greater

than the rank of the sending process and vice-versa. The amount of data to be sent per *middle* pointer is recursively halved. Figure 11 gives a detailed view of how the *middle* pointers are managed.

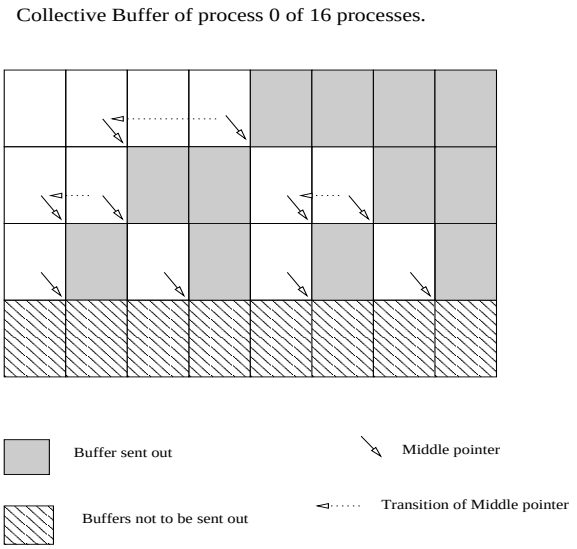

Figure 11. Managing Buffer pointers

Transfer Mechanisms: For transferring the buffers over the network, two different methods can be adopted. We can either transmit the buffer as one single RDMA or we can send the buffers individually using multiple RDMA writes. By using RDMA with gather list, we can reduce the message startup time. On the other hand by using multiple RDMAs we can achieve flexibility in the buffer management. Figure 12 shows the performance of RDMA Write with gather and multiple RDMAs. We clearly note that the startup costs associated with RDMA write gather are comparably lower than using multiple RDMA Writes Hence, we choose the RDMA with gather method for transferring small messages.

5.4 Design for Large messages: DE

For large messages, the network latency is the major factor in determining the total time taken by the All-to-All operation. Hence, we choose the Direct Virtual Ring based algorithm for implementing `MPI_Alltoall`. We implement the Direct Virtual Ring based algorithm in an eager manner. We call our scheme Direct Eager (DE).

We must consider message startup costs if we want to achieve a zero-copy implementation. In order to achieve zero copy, we have to :

- Register the user buffer
- Exchange addresses of the user buffer

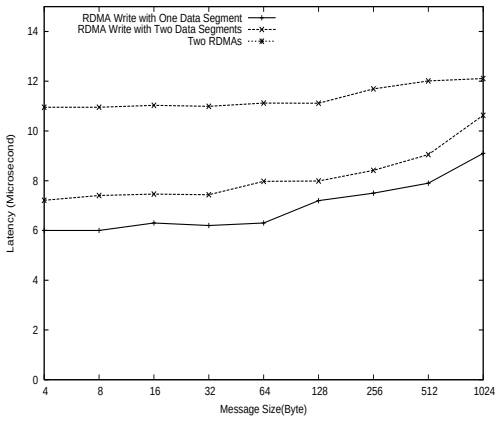

Figure 12. InfiniBand Gather and Multiple RDMA Performance

- Mark the completion with an explicit ack

In order to avoid registering parts of the buffer multiple times, as done by the MPI point-to-point based implementation, we register the entire send buffer as one single buffer. This avoids generating unnecessary entries in the registration cache and needlessly filling it up. For long running applications which do a lot of message-passing, this is critical, so as to minimize the cache miss rate. Now, we take a look at the overall latency for the All-to-All operation for the Direct Virtual ring based algorithm.

$$T_{direct-ring} = (p-1)t_{c-reg} + (p-1)mt_{w-reg} + (p-1)t_{rndz} + t_w m(p-1) \quad (3)$$

Where,

t_{c-reg} = Constant registration cost
t_{w-reg} = Time to register one page
t_{rndz} = Rendezvous exchange cost

In addition to multiple memory registrations, the *rendezvous* protocol presents a bottle-neck at each of the $(p-1)$ steps of the Direct algorithm. Each message transfer has to be preceded by interaction of both sender and receiver. This takes away the advantage of RDMA, in that it is no longer truly one sided. The entire `MPI_Alltoall` is then bottlenecked.

Instead, we adopted a new *Direct Eager* mechanism for implementing the All-to-All operation. In this new scheme, every process sends its receive buffer address to its next nearest neighbour, then the next one and so on in a ring-like manner. That is, process $rank$ sends its address to $(rank + i)\%p$. $\forall i(0 \leq i \leq p)$. This phase is totally network parallelized as the addresses and memory handles are RDMA-ed to pre-registered

buffers. Then, process $rank$ waits for address from process $(rank + p - 1 - i)\%p$. $\forall i (0 \leq i \leq p)$. We note that the time spent waiting for the first address is almost negligible since the process i sends address to j first and j sends data to i first. Hence, we name our scheme as *Direct Eager*. Figure 13 shows one step the algorithm. This is repeated for $\forall i (0 \leq i \leq p)$ steps.

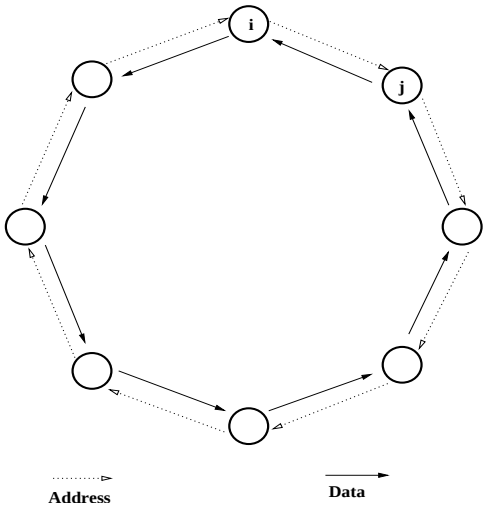

Figure 13. Direct Eager Mechanism

With this mechanism, the cost for the entire All-to-All operation is,

$$T_{direct-ring} = t_{c-reg} + (p-1)mt_{w-reg} + t_w m(p-1) \quad (4)$$

6 Performance Evaluation

In this section, we evaluate performance of our All-to-All communication. We conducted our experiments on a 16 node cluster. The cluster consists of 8 each of two different types of machines, I and II.

- I : SuperMicro SUPER P4DL6 node. Dual Intel Xeon 2.4 GHz processors, 512 KB L2 cache, 512 MB memory, PCI-X 64-bit 133 MHz bus.
- II : SuperMicro SUPER X5DL8-GG node. Dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, 1 GB memory, PCI-X 64-bit 133 MHz bus.

All the machines had Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected using the Mellanox InfiniScale 24 port switch MTS 2400. The Linux kernel version used was 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. It is to be noted that performance numbers for 4 and 8 nodes are on machines II.

The All-to-All latency was obtained by executing `MPI_Alltoall` 1000 times with the same buffer being used for communication. The average of the latencies from all the nodes was calculated. The calls to `MPI_Alltoall` were synchronized in each iteration using `MPI_Barrier`.

6.1 Evaluation for Small Messages

We implemented the Hypercube algorithm (HRWG) for performing the All-to-All communication for small messages. Figure 14 compares the performance of the current implementation and the proposed scheme. We observe that the proposed scheme (HRWG) has a 3.07 factor of improvement over the current implementation. The current implementation is limited to doing a point-to-point based communication due to the rise in copy-cost as a result of increase in the number of processes. Figure 15 shows the scalability of our implementation. We note that with increasing system size, it is indeed better to have a combining algorithm than a naive point-to-point based implementation.

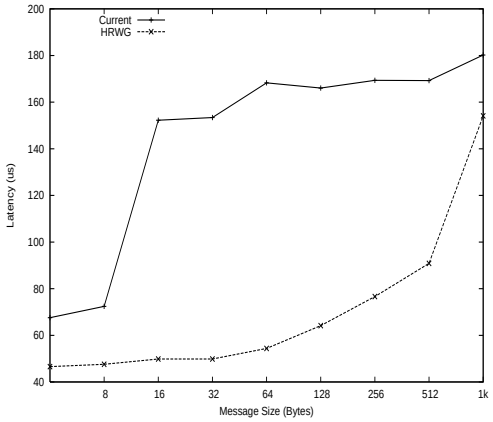

Figure 14. Small Messages on 16 Nodes

6.2 Evaluation for Larger Messages

For medium and large messages we implement the Direct Virtual Ring based algorithm with the Direct Eager mechanism. For larger messages the current implementation falls back on a Pair-by-Pair Exchange algorithm implemented on `MPI_Sendrecv`. We observe that this algorithm is designed mainly with older generation networks where sending large messages indiscriminately in the fabric would lead to congestion. The current generation InfiniBand switches are entirely non-blocking and provide cross-bar connectivity. Thus, it is no longer essential for us to fall back on Pair-by-Pair Exchange algorithm. The Direct Ea-

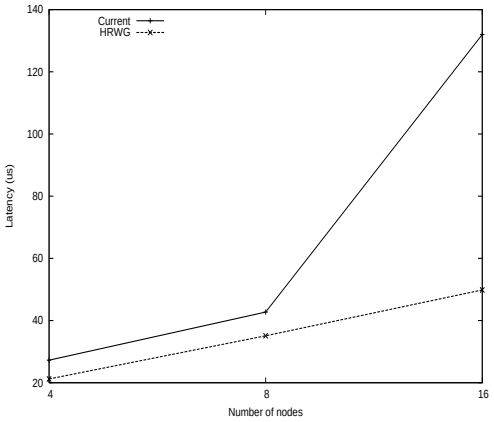

Figure 15. Small Messages (32 Bytes) Scalability

ger mechanism performs better than the current implementation. Figures 16 and 17 show the performance of our implementation compared to the current one. We note that DE performs better for medium sized messages. This is mainly because the total cost for *rendezvous* is comparable to that of the message transfer latency. Figures 18 and 19 show the scalability of our designs. We note that the difference between the current implementation and the proposed designs is infact growing as the number of processes increases.

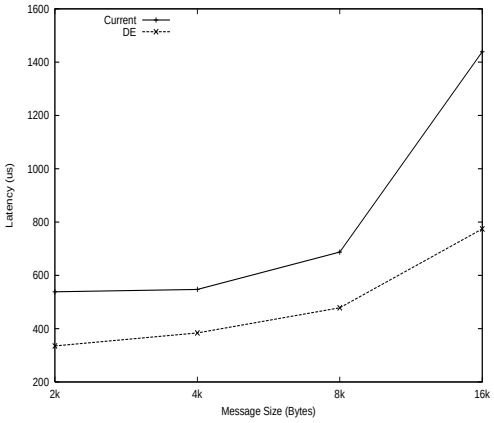

Figure 16. Medium Messages on 16 Nodes

6.3 Performance Extrapolation for Large Messages

In this section, we try to extrapolate the performance of our *Direct Eager* mechanism to find out how much performance improvement we can expect over larger scale clusters.

We note from Equations (3) and (4), the *Direct Eager* mechanism avoids adding a cost of rendezvous that

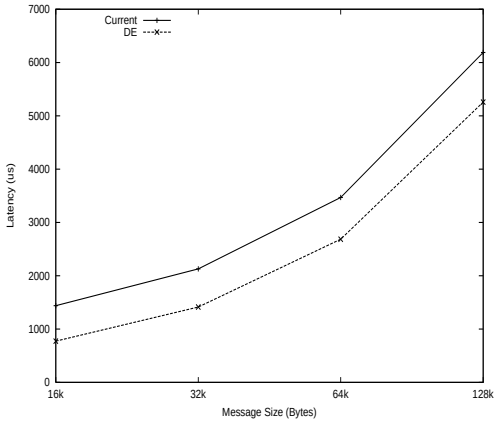

Figure 17. Large Messages on 16 Nodes

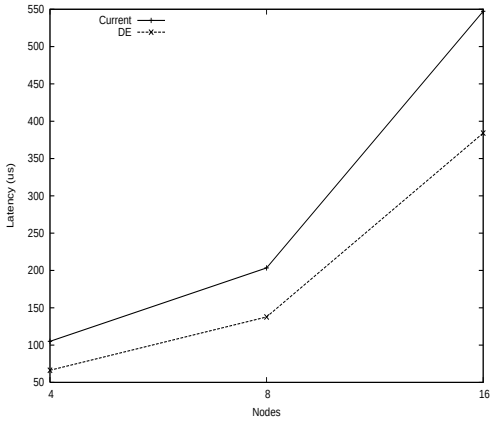

Figure 18. Large Messages (4k) Scalability

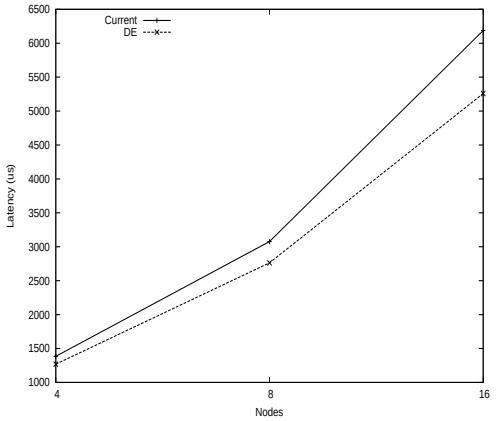

Figure 19. Large Messages (128k) Scalability

is linear according to the number of processes. We expect that this will show performance improvement when the All-to-All communication happens over a large cluster. In order to evaluate the benefits clearly, we assume 100% buffer re-use. That is, we try to eliminate the effects of the MVAPICH cache from the cost model. Including the cache will not lead to degradation of our implementation, since we use lesser (actually only one) cache entry per All-to-All communication as compared to $(p-1)$ for current implementation. Using our extrapolation, we determine that a performance benefit of 77% can be obtained for an All-to-All communication of 4k message size among 1k nodes. The Figure 20 shows this extrapolation graph obtained from equations (3) and (4).

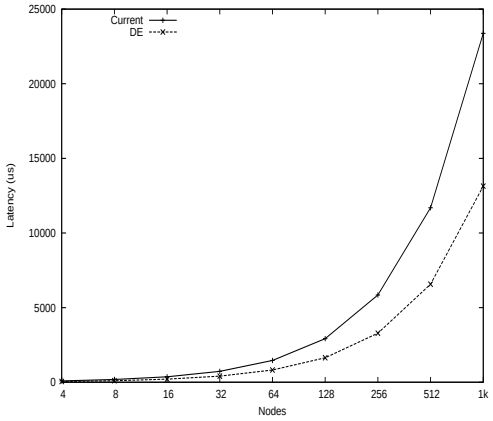

Figure 20. Performance for 4k message among 1k processes

7 Related Work

Traditionally, study of algorithms for supporting efficient All-to-All communication has been an active area of research [7, 9]. Researchers have focused on a variety of algorithms, mainly based on the physical topology and architecture of highly parallel supercomputers [18, 16]. Clusters of workstations present a very different challenge for providing scalable and efficient All-to-All communication. Recent research in this direction suggests that some algorithms perform better than others for a given message size than others and vice-versa [15, 6, 17].

8 Conclusions and Future Work

In this paper, we presented new designs to take advantage of the advanced features offered by InfiniBand in order to achieve scalable and efficient implementation of the `MPI_Alltoall` collective. We proposed that the implementation of collectives be done directly on the InfiniBand Verbs Interface rather than using MPI level point-to-point functions. We evaluated in detail why MPI point-to-point calls are a hindrance to achieving good performance from collective operations.

We detailed our design challenges and proposed two different schemes for small and large messages, HRWG and DE respectively. Our performance evaluation on our 16 node cluster shows that we can get an improvement of upto a factor of 3.07. We studied the analytical models of our implementation, and our investigation shows that for a 1k node cluster, we can get a performance improvement of upto 77%.

We plan to continue our work in this direction. In this work we have focused on fully utilizing the RDMA and RDMA Write Gather for efficiently implementing `MPI_Alltoall`. We plan to explore the use of InfiniBand Hardware Multicast and Atomic operations for furthur improving our designs. Also, we plan to have an efficient `MPI_Alltoall` implementation for SMP machines. We intend to evaluate our designs on a larger scale cluster.

References

[1] InfniBand Trade Association. Infiniband architecture specification. Release 1.0, October 2000.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. volume 5, pages 63–73, Fall 1991.

[3] 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. February 1995.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high performance, portable implementation of the mpi message passing interface. In *Parallel Computing*, 1996.

[5] W. Gropp, E. Lusk, and A. Skjellum. MPICH-a portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[6] M. Jacunski, P. Sadayappan, and D. K. Panda. All-to-all broadcast on switch based clusters of workstations. In *IPPS/SPDP*, 1999.

[7] S. Lennart Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9), September 1989.

[8] 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 MPI/PVM*, 2003.

[9] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing : Design and Analysis of Algorithms*. Addison Wesley / Benjamin Cummings, 1993.

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

[11] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html, January 2004.

[12] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *International Conference on Supercomputing*, 2003.

[13] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network: High performance clustering technology. *IEEE Micro*, 2002.

[14] 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.

[15] N. S. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan. Hybrid algorithms for complete exchange in 2-d meshes. *IEEE Transactions on Computers*, 12(12), December 2001.

[16] R. Thakur and A. Choudhary. All-to-all communication on meshes with wormhole routing. In *IPPS*, 1994.

[17] R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. In *Euro PVM/MPI conference*, 2003.

[18] Y. Yang and J. Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *HPCA*, 99.