Efficient Collective Communication using RDMA and Multicast Operations for InfiniBand-Based Clusters

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Sushmitha P. Kini, B.E.

* * * * *

The Ohio State University

2003

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Advisor

Prof. Gerald Baumgartner

Approved by

Advisor

Department of Computer and Information Science

© Copyright by

Sushmitha P. Kini

2003

ABSTRACT

Collective Communication operations form a vital part of most high performance computing applications. The traditional implementations of these operations in the Message Passing Interface (MPI) implementations are based on the point-to-point messaging operations. Modern interconnects like InfiniBand Architecture (IBA) provide high performance primitives like Remote Direct Memory Access (RDMA) and hardware-based multicast for communication. Exploiting these features of InfiniBand to efficiently implement the collective operations is a challenge in itself.

In this thesis, a discussion of the issues involved in the design of a collective communication library using the features available in the IBA networks is provided. The design alternatives available, the feasible algorithms, implementation issues and the performance benefits achieved using the new implementations are also discussed.

The new Barrier and Allreduce implementations give considerable performance improvements over the traditional implementation based on the point-to-point message passing model. The barrier implementations improve the performance up to 1.29 times on a 16-node cluster. The allreduce implementations provide a factor of improvement of up to 2.06.

This is the first attempt to characterize the multicast performance in IBA and to demonstrate the benefits achieved by combining it with RDMA operations for efficient implementations of collective operations.

I dedicate this work to Amma and Appa

ACKNOWLEDGMENTS

I would like to express my sincere appreciation and gratitude to my advisor Dr. Dhabaleswar K. Panda for his constant guidance and motivation. His valuable support has gone a long way in the completion of this work.

I am grateful to Dr. Gerald Baumgartner for consenting to serve on my Master's examination committee.

For generously sharing their thoughts, ideas and comments, I would like to acknowledge Dr. Pete Wyckoff, Jiuxing Liu, Jiesheng Wu, Darius Buntinas, Rinku Gupta, Balasubramanian Chandrasekaran and Pavan Balaji.

I would like to thank Kevin Deierling, Jeff Kirk and Ezra Silvera from Mellanox Technologies for their support with the InfiniBand hardware and software. I would also like to thank Ben Eiref, Robert Starmer, and Loren Boden from Topspin Communications for all their efforts in providing us access to their 16 node InfiniBand cluster.

Special thanks to my roommates (the grils) and friends at the Ohio State University, who made my stay here a memorable experience.

I am also indebted to the Students Safety Service of the Ohio State University, without whose services, the late night sessions in school would not have been possible.

My heartfelt thanks to my family and friends for their support, love and affection.

VITA

March 7, 1979	Born - Coimbatore, India
1996 - 2000	B.E. Computer Science and Engineering
June 2000 - July 2001	Software Engineer, Novell Software Dev. (I) Ltd., Bangalore
Sep 2002 - Present	Graduate Research Associate, Ohio State University.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in High Performance Computing: Prof. D.K. Panda

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the past decade, the high performance computing community has seen a shift from mainframe and host-centric computing to a parallel and network-centric computing approach. Network interconnects that offer very low latency and high bandwidth have been emerging to complement the increasing computational power of commodity PCs. This trend has enabled the deployment of high-end production scientific computing environments, built using clusters of commodity PCs and high speed interconnects. The high performance-to-price ratio of these commodity-off-the-shelf (COTS) clusters has been the most important factor that has accelerated this trend.

Parallel applications developed for these computing environments, by definition require co-operation between the processors to solve a task and this requires some means of communication. Thus, the performance of such applications depends to a great extent on the inter-processor communication provided by the communication subsystem. The networking interconnects, the communication protocols, and the messaging middleware form some of the vital components of the communication subsystem.

In this chapter we provide a brief overview of the networking interconnects, protocols and messaging middleware.

1.1 Interconnect Technologies

Some of leading products in the network interconnects market include Myrinet [2], Giganet cLAN [8], Gigabit Ethernet [4], Quadrics [19] and InfiniBand Architecture (IBA) [9]. Some of these interconnects provide very low latencies (even less than 10 μs) and very high bandwidth (of the order of Gbps). These interconnects provide memory-protected user-level access to the network interface, thereby allowing data transfer operations to be performed without kernel intervention. Thus the interconnect no longer is the bottleneck in the critical path of the communication. Some of the interconnects like IBA provide hardware-based support for Quality of Service(QoS) and for multicast operations. These provisions at the hardware level open avenues to develop higher-level layers using a novel approach.

1.2 Communication Protocols

The Gbps speeds offered by the network interconnects has shifted the onus of reducing the communication latencies from the networking hardware to the software messaging layers. Kernel-based protocols like TCP/IP are not capable of effectively utilizing the performance provided by the underlying network. Consequently, various OS-bypass protocols like AM [27], LAPI [5], EMP [18], VIA [3] and IBA were developed to remove the kernel from the critical path and to thereby reduce end-to-end latencies. They reduce the protocol processing overheads like copying data into intermediate buffers, kernel context switches, etc. Some protocols also offer the option of offloading some of the processing to the NICs, therefore leading to further reduction in the communication latencies. IBA was the result of the effort taken by the industry to consolidate the benefits provided by the user-level protocols and to standardize the specifications.

1.3 Message Passing Programming Model

High performance applications are written using a variety of programming models. The programming models are chosen based on the target system architecture, which could be shared memory processors (SMPs), massively parallel processors (MPPs) or distributed memory network of workstations (NOWs). For the NOWs with distributed memory, the Message Passing programming model has been one of the most efficient and easy-to-use approaches. Message Passing Interface (MPI) [16] is a popular standard for the message passing programming model.

In this model, as shown in Figure 1.1, processes do not share an address space. The processes are named and data transfer is done using explicit communication through send and receive calls. Apart from the send-receive communication calls, the programming model can also include synchronous and asynchronous operations, group communication, and aggregate operations.

These message passing middleware libraries now need to be designed to derive the maximum benefit possible from the high performance communication protocols. A number of factors like the copying of messages between buffers, handling the posting of descriptors, buffer alignment, etc. affect the message passing performance. Ideally, the middleware should be a light-weight substrate that provides the applications with performance very similar to that of the underlying protocols.

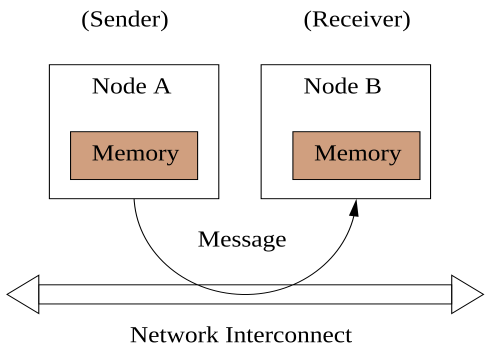


Figure 1.1: Message Passing Programming Model

1.4 Collective Communication

The message passing programming models specify operations for both point-to-point messaging and for group or collective communication. Unlike point-to-point messaging where data transfer happens between two processes (called the sender and receiver), collective communication involves data transfer or synchronization between a group of processes.

Collective communication operations are important to parallel and distributed applications for data distribution, global processing of distributed data, and process synchronization. High performance parallel applications are often involved in operations that require the co-operation of a group of processes. Collective operations are used in these applications whenever there is a need for global communication interleaved with stages of local computation.

Some of the commonly used collective operations in parallel applications include:

1. *Barrier* - This is a synchronization operation involving no transfer of data.

2. *Broadcast* - A data transfer operation that is used to send the data from one process to all the processes in the group.

3. *Gather* - An operation to collect data from all processes in the group and transfer it to a single process.

4. *Reduce* - Used to perform some operation, say summation or calculating the minimum, on data from all the processes and to transfer the result of the operation to a single process.

There are many other variations of these operations available. For example, in the gather and reduce operations, the collected data may be broadcast to all the processes. The collective communication libraries are tailored to meet the diverse requirements of the applications. Most of these operations are supported by the Message Passing Interface (MPI) standard.

The fast improving performance of the underlying interconnects has led to the shift in communication bottleneck from the network fabric to the software layer at the sending and receiving ends. Hence it is vital that the software developers of message passing libraries make the best use of the primitives offered by the interconnects and implement the messaging layers with minimal overheads.

1.5 Problem Statement

Developers of message passing libraries that support collective communication, have to consider not only the various algorithms available for the operations, but also the support from the underlying protocols for efficient implementations of these operations.

In the earlier generation MPP and SMP systems, collective operations were achieved by using special hardware support. The current generation clusters typically use software based collective operations built using the point-to-point communication operations.

Most user level protocols provide a send/receive or channel semantics based model of communication, which requires explicit function calls at the sender and receiver ends. Recent technologies like VIA and IBA also offer a different model based on memory semantics. They allow transfer of data directly between user level buffers on remote nodes without the active participation of either the sender or the receiver. This method of operation is called Remote Direct Memory Access (RDMA). RDMA allows a process to directly access a remote process' user buffer without the remote process making an explicit function call.

Another attractive feature in the IBA networks is the support for hardware-based multicast. This primitive is provided under the Unreliable Datagram (UD) transport mode. IBA allows processes to attach to a multicast group and then the message sent to the group will be delivered to all the processes in the group. This means that a single descriptor needs to be posted in order to perform a collective operation.

Given these powerful and efficient features in IBA we are faced with the interesting question of whether these remote memory data transfer primitives and multicast support in IBA clusters can be made use of for optimizing the performance of collective operations. The performance characteristics of IBA networks and the additional features lead us to reevaluate the efficacy of the current implementation of collective operations.

As a part of this research, we explore the idea of developing collective communication libraries using the efficient low-level data transfer primitives. Replacing the point-to-point communication calls in the collective operations with faster lower-level operations can provide significant performance gains. Performance improvement is possible due to various reasons, some of which are:

1. *Reducing Data Copies* - The number of data copies between buffers in the messaging layer can be reduced by avoiding point-to-point messaging protocols. Also for multi-stage algorithms, data can be directly transferred from intermediate buffers, and copies to and from the user buffers can be avoided.

2. *Avoiding Tag Matching* - The software overhead of matching tags of messages can be avoided by assigning specific buffers for messages and using RDMA calls to write to them.

3. *Eliminating Unexpected Message Handling overhead* - If the message arrives before a receive call is made, it is placed in a temporary buffer. The receive function has to then search the "unexpected" queue of buffers for the message and copy it into the user buffer when found. All this processing adds considerable overhead to the basic send-receive latency. Using RDMA writes, such extra copies of messages can be avoided.

4. *Reducing Number of Posted Descriptors* - The hardware multicast feature allows a single descriptor to be posted at the sender end to send data to multiple receivers. This primitive fits in well with the semantics of collective operations and hence can be utilized to our advantage.

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

1. *Can we optimize the MPI collective operations by using algorithms that leverage the RDMA primitives in IBA instead of algorithms that use the existing MPI point-to-point operations?*

2. *Can the multicast primitives in IBA be used to implement scalable collective communication operations?*

1.6 Our Approach

In this thesis we discuss the various design issues and challenges faced in the development of a collective communication library using RDMA and Multicast primitives for IBA-based clusters. We discuss the issues like buffer management, buffer reuse mechanisms, data reception and reliability for Unreliable Datagram messages. As a proof of concept, we have designed, implemented and evaluated two popular collective operations. We have considered the Barrier operation, which is a synchronization operation between all the processes in the group. This operation involves no transfer of data and is thus a good starting point for understanding the basic issues involved in RDMA and Multicast based collective operations. For this operation, we also discuss three different implementations. Two of these are pure RDMA based solutions and the other uses both RDMA and Multicast support.

We also look at the Allreduce operation, which involves a global reduction and broadcast of data. We discuss the design issues for two different implementations of this algorithm.

The rest of this thesis is organized as follows. In Chapter 2 we provide an overview of the InfiniBand Architecture and the various features it provides. We also take a look at the MVAPICH [17] implementation of the MPI standard for IBA-based clusters.

Chapter 3 delves into motivation for the work and the core issues of implementing the collective communication library. In Chapter 4 we go into details of the algorithms, design and performance evaluation of the Barrier operation. Chapter 5 discusses the solutions offered for the Allreduce operation. We conclude and discuss the areas for future work in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter we provide an overview of InfiniBand Architecture and the set of features that can be utilized for the efficient implementation of point-to-point and collective message passing operations. We also provide a brief overview of the MVAPICH message passing library.

2.1 InfiniBand Architecture

InfiniBand Architecture [9] defines a System Area Network (SAN) for connecting multiple independent processor platforms, I/O platforms and I/O devices. This industry standard uses scalable switched serial links to design clusters and servers offering high bandwidth and low latency. IBA makes use of kernel-bypass techniques to offer zero-processor copy data transfers between user level processes on remote nodes. Figure 2.1 shows the different components of an IBA SAN.

In an InfiniBand network, nodes are connected to the IBA fabric using Channel Adapters (CA). Host Channel Adapters (HCA) are installed into the processing nodes and initiate communication within the fabric. Target Channel Adapters (TCA) connect I/O nodes to the fabric.

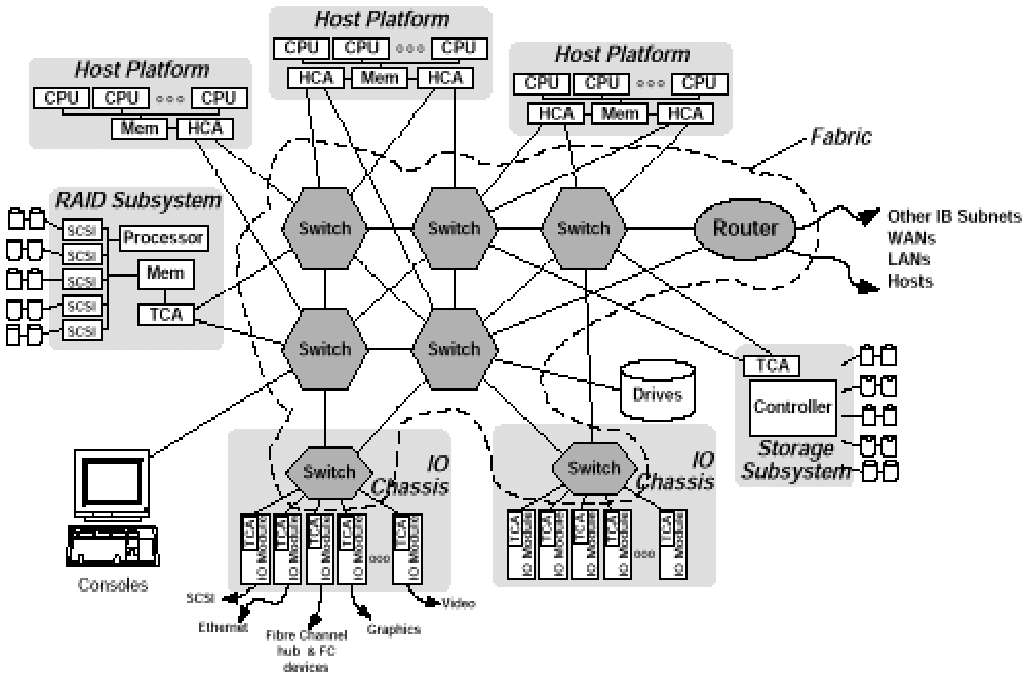


Figure 2.1: IBA System Area Network (Courtesy IBATA)

The InfiniBand specification classifies the channel adapters into two categories: Host Channel Adapters (HCA) and Target Channel Adapters (TCA). HCAs are present in servers or even desktop machines and provide an interface that is used to integrate the InfiniBand with the operating system. TCAs are present on I/O devices such as a RAID subsystem. Each channel adapter may have one or more ports. A channel adapter with more than one port, may be connected to multiple switch ports. This allows for multiple paths between a source and a destination, resulting in performance and reliability benefits.

IBA defines a semantic interface called Verbs to configure, manage and operate a HCA. VAPI is the Verbs implementation provided by Mellanox Technologies [14] for the HCAs. It supports two kinds of communication semantics: channel semantics and memory semantics. In channel semantics, send/receive operations are used for communication. In this model, each send descriptor needs to be associated with a receive descriptor on the remote node. If a send descriptor is posted without a corresponding receive descriptor on the remote node, the message will be dropped. In memory semantics, remote direct memory access operations (RDMA write and RDMA read) are used. Here the initiator of the operation specifies both the local and remote virtual addresses. There is no need of posting a descriptor on the remote end.

In order to communicate, each process creates a Queue Pair (QP) which consists of a Send Queue and a Receive Queue. The transport service needed has to be specified when the QPs are created. Each Queue Pair is also associated with a Completion Queue (CQ) and a CQ can be associated with many QPs. Communication requests are initiated by posting Work Queue Requests (or descriptors) to the work queues.

The HCA executes these work requests in the order that they are placed in the work queue. When the HCA completes a request it places a Completion Queue Entry (CQE) in the Completion Queue. The processes can then poll on the CQs to check for completion of the requests. Figure 2.2 shows the IBA communication architecture. User buffers used for transferring data must be registered first before they can be used for communication. This allows the pinning down of the virtual memory region in order to prevent it from being swapped out when the HCA is accessing it.

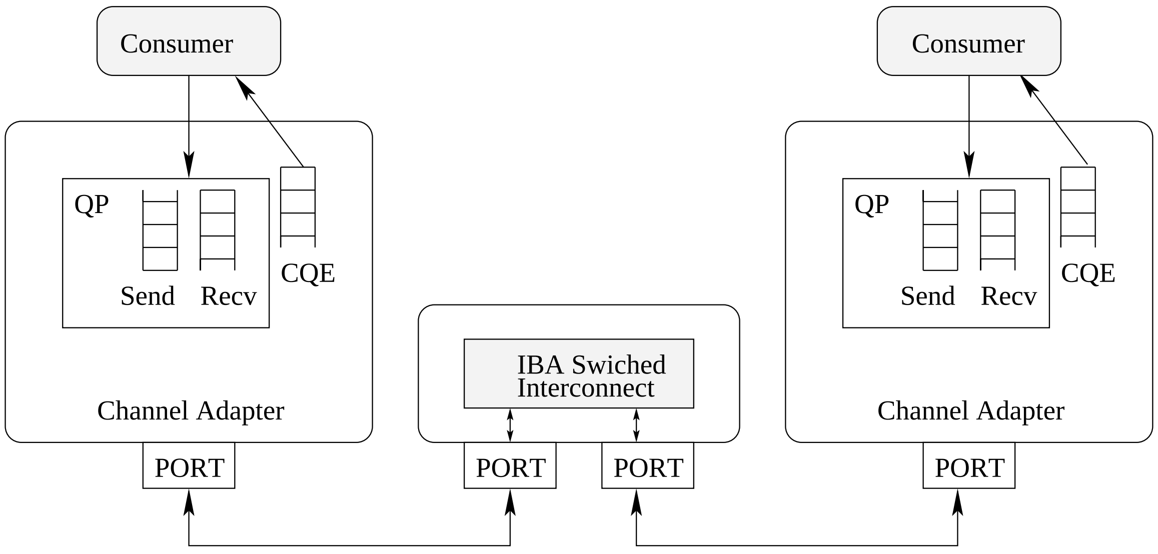


Figure 2.2: IBA Communication Architecture

The IBA transport mechanisms provide multiple classes of communication services. A connected service requires each consumer to create a QP for each consumer with which it wishes to communicate. The unreliable datagram service allows the consumer of the QP to communicate with any unreliable datagram QP on any node.

When a QP is created, it is configured to provide one of these classes of transport services:

1. Reliable Connection (acknowledged - connection oriented)
2. Reliable Datagram (acknowledged - multiplexed)
3. Unreliable Connection (unacknowledged - connection oriented)
4. Unreliable Datagram (unacknowledged - connectionless)
5. Raw Datagram (unacknowledged - connectionless).

The VAPI implementation currently supports only RC and UD transport services.

Some of the features of IBA that are of interest in the message passing context are described below.

RDMA Read

This is a memory semantic operation that allows a process to read a virtually contiguous buffer on a remote node and write to a local memory buffer. RDMA service is available only with the RC transport service type. It is possible to read data from a contiguous buffer at the remote process and scatter it into multiple buffer segments at the local process.

RDMA Write

This memory semantic operation allows a process to write to a virtually contiguous buffer on a remote node. This is a one-sided operation that does not incur a software overhead at the remote side. However, for the RDMA Write with Immediate operation, a receive descriptor will be needed at the receiver end. There is also a gather list available to send data from non-contiguous local buffers. Figure 2.3 shows the difference in latencies between the Send/Receive and RDMA write operations as measured on Cluster 1, which is described in Section 4.4. We see that the performance of RDMA write is much better than that of a Send/Receive operation and hence it would be beneficial to use this primitive for our purposes.

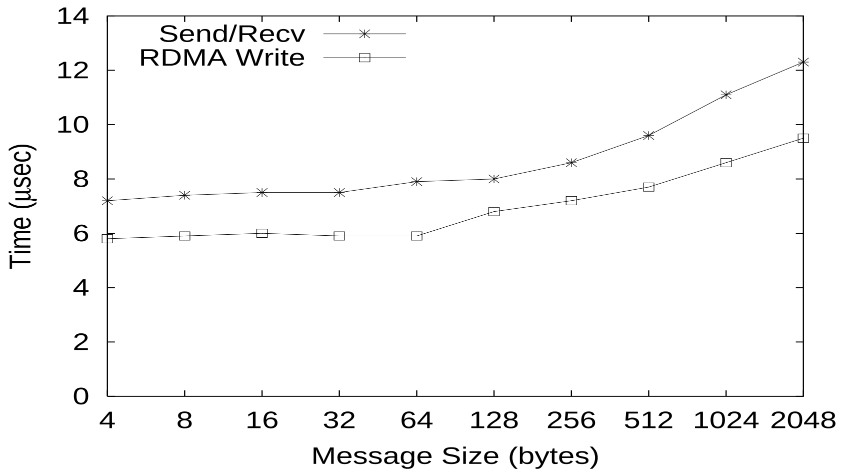


Figure 2.3: Comparison of RDMA write latency with Send/Recv latency at the VAPI layer (Cluster 1)

Multicast

Multicast is the ability to send a single message to a specific address and have it delivered to multiple processes which may be on different end nodes. This feature is implemented as a combination of the capabilities of the IBA HCA, switch and software layers by replicating the multicast message and sending it to all the designated receivers. Performance evaluations of this multicast primitive on Cluster 1 show that it takes about 9.6μs to send a 1-byte message to 1 node and 9.8μs to send the message to 7 nodes. This implies that the operation is very scalable and can be used effectively to design scalable collective operations. The multicast facility is available only with the UD service type. The UD service is connectionless and unacknowledged. It allows the consumer of the QP to communicate with any UD QP on any node, and thus greatly improves the scalability of IBA. Current version of VAPI supports a single multicast group that includes all the nodes in the subnet. We have made use of this "broadcast" primitive in our implementations. When the later versions of VAPI provide support for attaching to different multicast groups it will be possible to design new algorithms where messages are sent only to a select group of processes.

2.1.1 Message Passing Interface

Message Passing Interface (MPI) [16] is a standard library specification for the message passing programming model for parallel applications. Portability and ease of use make MPI the most popular interface for developing high performance applications.

The MPI specification defines a wide variety of operations for both point-to-point and collective communication, all scoped to a user-specified group of processes.

MPI defines a concept called the communicator, which is a data structure that contains the information about the processes in the group and the context of communication. The identification of processes in an application is done by means of logical ranks that are assigned to all the processes in the communicator (or group).

On most current generation systems it is possible to overlap communication with computation in order to improve performance. The MPI standard provides the ability to utilize this factor in the means of asynchronous or non-blocking operations. A synchronous operation blocks a process till the operation completes. An asynchronous operation is non-blocking and only initiates the operation. The caller could discover completion by polling, by software interrupt, or by waiting explicitly for completion later. There are blocking and non-blocking counterparts for the point-to-point messaging functions.

The MPI standard, as mentioned in Chapter 1, provides a variety of collective communication operations. All collective operations are currently defined to be blocking in nature.

2.1.2 MVAPICH - MPI over InfiniBand Architecture

MPICH [6] from Argonne National Laboratory is a popular open source implementation of the MPI standard. At the core of the MPICH design is a small set of functions that form the Abstract Device Interface (ADI) [23]. The ADI contains the protocol and network dependent code. The MPI functions are built using the functions provided by the ADI. The ADI allows easy porting of MPICH to various interconnect technologies. MVICH [10] from Lawrence Berkeley National Laboratory is one such implementation of ADI2 for VIA based clusters. MVAPICH [11] is the implementation of ADI2 for the VAPI interface of the InfiniHost HCAs and is derived from MVICH.

In the original MPICH design, the collective operations are implemented using the MPI point-to-point operations above the ADI. A collective operation like Barrier is implemented using the MPI_Send and MPI_Recv functions. The MPI_Send and

MPI_Recv functions are in turn implemented by using the device specific functions present in the ADI.

The collective communication library design, implementation and evaluations in this thesis have been done using the MVAPICH 0.8.5 [17] code base.

2.2 Related Work

The benefits of using RDMA for point-to-point message passing operations for IBA clusters has been described in [12]. The methods and issues involved in implementing point-to-point operations over one-sided communication protocols in LAPI are presented in [1]. However using these optimized point-to-point operations does not eliminate the data copy, buffering and tag matching overheads. A lot of research has taken place in the past to design and develop optimal algorithms for collective operations on various networks using point-to-point primitives, but not much work has been done on selection of the communication primitives themselves.

RDMA based design of collective operations for VIA based clusters [20, 21] has been studied earlier. Combining remote memory and intra-node shared memory for efficient collective operations on IBM SP has been presented in [26]. In [7] the performance of collective operations using optimized algorithms over SCI, a shared memory based interconnect, is discussed. None of these papers focus on taking advantage of the novel mechanisms in IBA to develop efficient collective operations.

2.2.1 Summary

In this chapter we provided an overview of the features and architecture of InfiniBand based clusters. We also provided an overview of the Message Passing Interface standard. In the next chapter we introduce some of the key issues to be considered in the design of a library based on RDMA and Multicast features of IBA.

CHAPTER 3

DESIGN ISSUES

The remote memory access and multicast features available in IBA clusters provide us with a wide range of options to implement the collective operations. The RDMA primitives help create a logical shared memory illusion across the processes on different nodes with distributed memory. The hardware based multicast primitive allows data to be collectively transferred to all the processes in the application in an efficient and scalable manner. This makes us consider the viability of implementing the collective operations using these low-level calls at the ADI layer instead of the traditional method of implementing them above it.

In this chapter we discuss the motivation for this work and the design issues that were considered as a part of the development of the library.

3.1 RDMA based design

As mentioned earlier, the RDMA primitives provide us with the ability to construct a logical shared memory space between processes. This allows us to apply simple algorithms for collective operations. Let us consider the case of a Barrier operation. A barrier is a synchronization point operation wherein each process blocks until all the processes in the application have reached the barrier. So the only information that needs to be shared between the processes is whether they have reached the logical synchronization point. Once they get this information about all the processes, they can continue with their computation.

In shared memory systems, a barrier operation can be done in a simple manner. Each process can update a well-known shared location by incrementing it. This updation needs to be done in an atomic operation. All the processes wait for this memory location to contain value equal to that of the number of processes in the application. Once this happens, all the processes exit the barrier, since they have the information that all the other processes have reached the barrier.

An alternative approach would be to have an array of bytes (of length equal to the number of nodes in the barrier) allocated at a well-known shared memory location. Each process on arriving at the barrier, toggles the initial value of the array element that corresponds to its rank. The processes then check to see if all the other elements of the array have been toggled, and wait until they are. This means that every process has arrived at the barrier and the barrier can be exited.

As seen in the previous examples, shared memory provides the advantage of implementing simple and efficient barriers. In clusters with distributed memory the collective operations are implemented using algorithms that use the MPI point-to-point communication calls. When an operation like barrier is executed, the nodes make explicit send and receive calls. The receive operation is generally an expensive operation since it involves posting a descriptor for the message. Also, if the message arrives before the receive call is made, it is placed in a temporary buffer. The receive function then has to search the unexpected queue for the message and copy the data into the user buffer. All this processing adds considerable overhead to the basic send-receive latency, thereby making the entire barrier operation slower. This is the kind of overhead that can be effectively eliminated using RDMA operations.

In order to use RDMA writes, each process allocates a set of buffers and registers them. In Figure 3.1, there are two processes P0 and P1. The addresses of the buffers are exchanged between the processes. Once the initial address and memory handle exchanges are done, all the buffers in the remote process become a part of the local process' logical address space. P0 can directly write to the buffer in P1's address space and vice versa. Therefore the algorithms used for shared memory architectures can be applied. We also see that in the case of each step of data transfer, there is no involvement of the receiver in the critical path. The receiver only needs to check the local registered buffer for the data from the remote processes, and there is no need to post a receive descriptor.

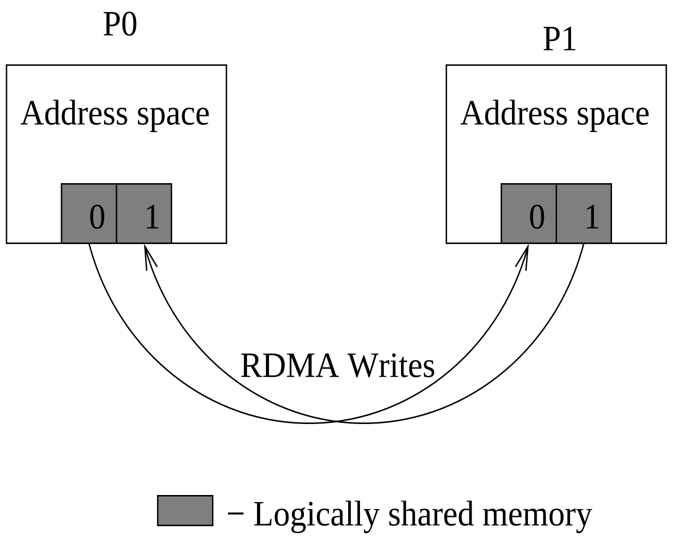


Figure 3.1: Logical Shared Memory created using RDMA operations

3.2 Advantages of using Multicast primitive

The multicast primitive in IBA, by its very definition, seems like a natural choice for implementing collective operations. Many collective operations like broadcast, allgather, allreduce, etc., require that data from one source buffer be transferred to buffers in all the processes in the group.

IBA provides support for processes to attach QPs to multicast groups. The QPs need to be of the Unreliable Datagram service type. Once a message is sent to this QP, it is delivered to all the QPs that are attached to the specified group. This primitive involves replication of the message at the switch and simultaneous delivery to all the receivers. The receivers will need to have posted receive descriptors to the receive queues.

The brute-force method of performing the broadcast-type operations would typically involve executing sequential sends to each process from the source process, in the form of a flat-tree. The implementation of such algorithms is done by using MPI_Send and MPI_Recv calls for each step of the data transfer. There also exist multi-stage algorithms that improve on this brute-force approach by reducing the total number of sends done from one process. For example, a broadcast maybe done using a binomial tree algorithm, wherein each process sends the data it has received to its children. But even in these optimized cases, there is posting of descriptors at the send queue of each parent process and at the receive queue of each child process. The posting of descriptors at each parent process can be avoided by making use of the multicast primitive. This is illustrated in Figure 3.2. Consider the broadcast of data from P0 to P1, P2, P3. In the implementation with point-to-point calls, there are two send descriptors posted by P0 and one by P2. In the implementation using the multicast primitive we see that there needs to be only one send descriptor posted by P0. Thus, the multicast feature provides us with the ability to do a broadcast with lesser overheads.

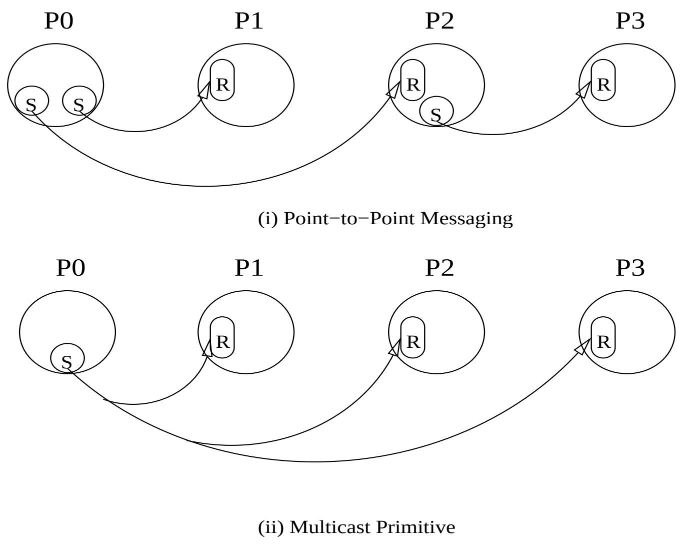


Figure 3.2: Illustration of Send and Receive Descriptors posted in the Broadcast Implementation using (i) Point-to-Point Messaging and (ii) Multicast Primitive

3.3 Design Issues

We now discuss the intrinsic issues associated with the design and implementation of the library using the RDMA Write and multicast operations. We discuss the buffer management and data reception issues to be handled when using RDMA. We also discuss the techniques to add reliability for the UD multicast. In this section, we present different alternatives for each of these issues. In the following chapters we focus on our choices and their implementations.

3.3.1 Buffer Management

IBA specification requires that all the data to be transferred be present in buffers that are registered. Implementing collective operations on top of point-to-point message passing calls leads us to rely on the internal buffer management and data transfer schemes which might not always be optimal in the collective operations context. In using RDMA, we have better control of the buffer consumption patterns. In order to use the RDMA method of data transfer, each node is required to pin some buffers and send/receive data using them. Also, the remote nodes should be aware of the local buffer address and memory handle, which means that a handshake for the address exchange should be done. The allocation and registration can be done at various stages during the life of the MPI application.

The first option is for each process to statically allocate a set of buffers for each collective operation and exchange the addresses during the initialization phase (i.e, as a part of MPI_Init). These buffers can now be used throughout the life of the application by following some buffer reuse schemes. We discuss some such schemes in the following chapters. The disadvantage of this approach is that the buffers will be allocated and pinned even if the application does not execute the collective operation.

The second option is for the buffers to be registered during the first collective operation call. This means that the buffers are allocated only when there is a need for them. Even in this case the buffers maybe reused for the subsequent collective operations. However, buffer registration is an expensive process and so the first collective operation call is likely to take more time than the ones that follow.

Another alternative is to allow dynamic allocation and registration of buffers during every collective call. This calls for an address exchange operation during each operation and can hence be very expensive.

3.3.2 Data Reception

The RDMA write operation is transparent at the receiving end since it does not require the receiver to post a descriptor for the data. Hence the receiver is not aware of the arrival of data, after it has been written by the remote process. We need a mechanism to notify the receiver of the completion of the RDMA write.

One method that can be used is to make use of the RDMA with immediate data feature. This operation consumes a descriptor at the remote end and hence deprives us of the transparency benefit of the RDMA write .

The other method is for the receiver to poll on the buffers for arrival of data. This means that when the buffers are allocated, they will need to be initialized with some special data so that the data arrival can be recognized. The processes can write the data associated with the collective operation into the remote buffers and also write the special data to indicate the arrival of data at the remote end. Each process polls for this special notification value to be written in the local buffers, and once it is available, it knows that the data for the operation has arrived as well. RDMA write does not allow for the scatter of data to non-contiguous buffers at the remote end. Hence we will need to make use of two RDMA writes, one for the actual data and the other for the notification data. Posting two RDMA writes can be an expensive task. Hence we can try to avoid the second RDMA write by piggybacking the notification value with the actual data. The sender places the notification data in the bytes that follow the actual data. In most collective operations, the receiver process is aware of the amount of data that is being transferred. Hence it can poll for the notification value at the end of the data buffer. Thus the arrival is recognized at the receiving end using a single RDMA write.

The above mentioned scheme works well if the total amount of data that is being transferred is less than the MTU of the underlying network. If not, the data will be transferred as multiple packets and the order in which the contents of the packets are written by the NIC to memory might vary. That is, just polling for the notification data does not guarantee message arrival because the data from the other packets might not have yet been written by the NIC to the memory. In this case we are forced to use two RDMA writes to indicate completion. When the second RDMA write is completed (by polling for the notification data), the receiver can be assured that the data has arrived completely.

3.3.3 Buffer Reuse

As mentioned in Section 3.3.1, it is possible to allocate a single contiguous buffer in the beginning and partition it into blocks as per the need of the collective operation. Since the sender writes to the receiver's virtual memory, it needs to be aware of when these buffers are free for use. That is, they might contain data from an earlier collective operation and it might not be safe to overwrite the data. One option that can be used to handle this issue is for the receiver to send a notification to the sender when the buffers are free for reuse. When the sender receives such a notification, it can write to the buffers in the receiver safely.

Another option would be make use of the intrinsic properties of the collective operation as an indication of the safety of buffer reuse. Consider the Allreduce operation. We can make use of a scheme where there are two sets of buffers being used. For the first operation, all processes write to buffer 1 of the remote processes. For the second operation, they write to buffer 2. When the second allreduce operation is completed, the processes are sure that all the other processes have exited from the first allreduce, and buffer 1 is free to be used. So for the next allreduce operation they can write to buffer 1. Thus the "double buffer" scheme can be used to safely reuse the buffers in a collective operation.

3.3.4 Reliability for Unreliable Datagram multicast operations

The MPI specification assumes that the underlying communication interface is reliable and that the user need not have to cope with communication failures. Since the multicast operation in IBA is unreliable, reliability has to be handled in our design.

One of the alternatives is to provide an acknowledgment (ACK) message from the processes after every multicast message is received. The sending process waits for the ACKs from all the nodes and retransmits the message if it does not receive the ACKs within a specific time period. This technique is expensive since there is a message being sent back from every process to the root, even after the barrier is logically completed.

Another alternative would be for each receiving process to maintain a timer and send a negative acknowledgment (NAK) to the sending process when it has not received a message. This NAK could be sent using the Send/Recv primitives. When the root process receives this message, it can retransmit the multicast message. However, this means that the application should make some MPI communication call in order for the root node to receive the packet and make progress.

The IBA specification allows for event handlers to be called when a completion queue entry is generated. There is the option of triggering these event handlers on the receive side only if the "solicit" flag is set in the message by the sender. This facility can be used in the NAK message. By setting the solicit flag, this message triggers the event handler at the sender process. The event handler then checks for the validity of the retransmit request and does a retransmission of the multicast message.

3.4 Summary

In this chapter we discussed the advantages of using the RDMA write and multicast primitives for the collective operations. We also discussed the issues to be handled when the library is implemented. In Chapter 4 we go into the details of the algorithms and implementations for the Barrier operation. We also present the performance evaluations performed for the different implementations.

CHAPTER 4

BARRIER

In this chapter we discuss the Barrier operation [22] which is a commonly used collective operation in parallel applications that are developed using the MPI programming model. Barriers are used for synchronizing the parallel processes and involve no transfer of data. They maybe used to separate phases of an application program. The MPI_Barrier function call is invoked by all the processes in a group. This call blocks a process until all the other members in the group have invoked it. An efficient implementation of the barrier is essential because it is a blocking call and no computation can be performed in parallel with this call. Faster barriers improve the parallel speedup of applications and helps in scalability. Therefore it is important to minimize the time spent waiting on barriers. The syntax for the barrier routine in the MPI standard is as follows:

MPI_Barrier(MPI_Comm *comm*)

where:

MPI_Comm is an MPI predefined structure for communicators.

comm is a communicator

We explain the various algorithms that can be used to perform a barrier and the implementation details. We also provide the experimental results obtained for each of the implementations.

4.1 Algorithms

In this section we present three different algorithms that can be used to implement a barrier. The aim is to leverage the RDMA and multicast features offered by IBA to the fullest extent possible. We discuss the algorithms and the number of steps that are taken in the execution of each algorithm.

In the following subsections we denote processes using symbols i, j, k and the total number of processes involved in the barrier is denoted by N. We refer to the process that has a distinguished role to play in some algorithms as the *root*. We indicate the number of the current barrier by the symbol *barrier_id*.

4.1.1 RDMA-based Pairwise Exchange (RPE)

The algorithm for the barrier operation in the MPICH distribution is called the Pairwise Exchange (PE) recursive doubling algorithm. MPICH makes use of the MPI_Send and MPI_Recv calls for the implementation of this algorithm. This is a recursive algorithm where the processes are paired up and each process does a send and receive with its partner. Each pair of processes now forms a group. The groups are now paired with each other and every process in one group does a send and receive with a process in the other group. The pairs of groups are now merged to form a bigger group. This pairing and message exchange continues until all the processes are merged into one group. This indicates the completion of the barrier.

If the number of nodes performing the barrier is a power of two, then every process in this algorithm makes $\log_2 N$ sends and receives.

If N is not a power of two, then the algorithm proceeds as follows. The largest power of two value less than N, that is N' is calculated. The processes are divided into two groups, G containing all processes of rank less than or equal to N' and G' containing the remaining processes. All the processes in G' send a message to a process in G. The nodes in G perform the pairwise exchange algorithm. Then the processes that received messages initially reply to the corresponding processes in G'. The number of steps taken in this case is $\lfloor \log_2 N \rfloor + 2$. Figure 4.1 shows the steps performed for a 5 node barrier. In this case N is 5 and N' is 4. In the first step, P4 (belonging to G') sends a message to P0 (belonging to G). Then P0, P1, P2, P3 perform the pairwise exchange algorithm in two steps. In the last step, P0 sends the barrier completion message to P4.

Now we describe how this algorithm can be performed using the RDMA Write primitive. The barrier is a collective call, and so each process keeps a static count of the current barrier number, *barrier_id*. Each process allocates an array of bytes of length N and registers it. The address of this array is exchanged among all the processes. In each step of the PE, process i writes the *barrier_id* in the i th position of the array of the partner process j. It then waits for the *barrier_id* to appear in the j th position of its own array. Since each process is directly polling on its local memory for the reception of data, and it avoids the overhead of posting descriptors and copying of data from temporary buffers, as is the case when the MPI_Recv call is used.

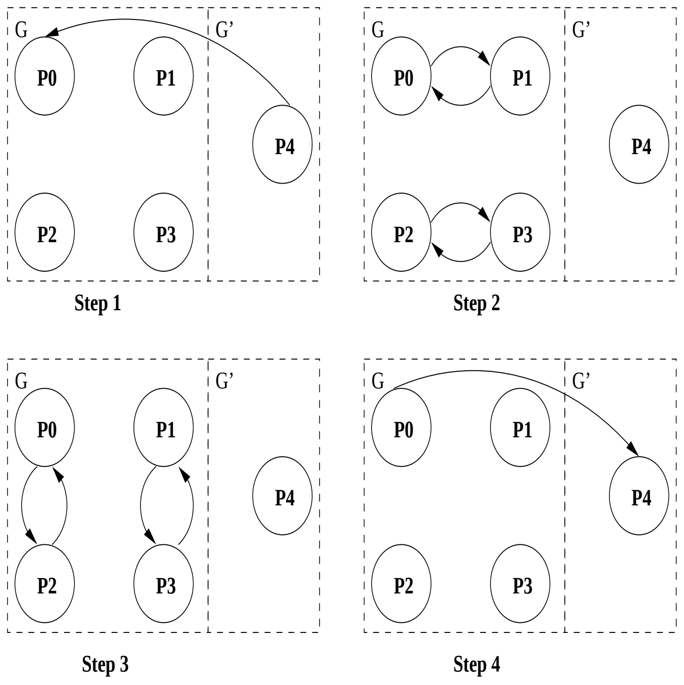


Figure 4.1: Steps in the Pairwise Exchange Algorithm for a 5-node Barrier

4.1.2 RDMA-based Dissemination (RDS)

In the Dissemination Barrier algorithm as described in [15], the synchronization is not done pairwise as in the previous algorithm. In step m, process i sends a message to process $j = (i + 2^m) mod N$. It then waits for a message from the process $k = (i + N - 2^m) mod N$. This algorithm makes $\lceil \log_2 N \rceil$ synchronization operations in the critical path regardless of whether there are power of two or non-power of two number of nodes and thus is a more symmetric pattern of synchronizations.

The barrier signaling operations using RDMA write are done exactly as in the RPE algorithm, and this algorithm only varies in the way in which the processes are grouped for communication in each step.

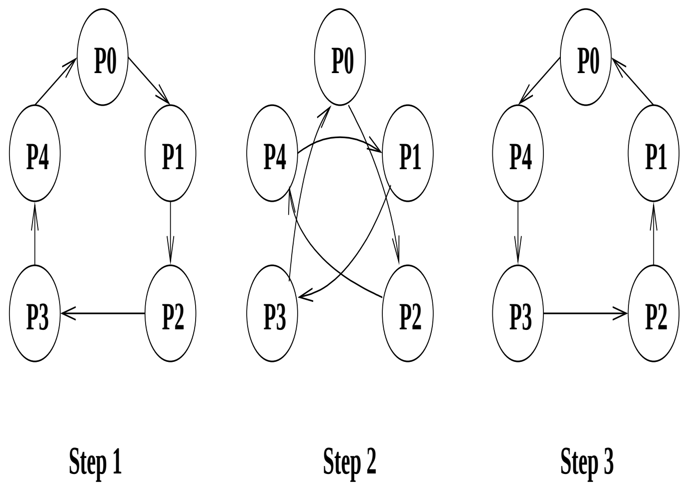


Figure 4.2: Steps in the Dissemination Algorithm for a 5-node Barrier

The Figure 4.2 shows the communication in the various steps of this algorithm for a barrier on 5 nodes. The number of steps taken in this case is 3.

4.1.3 RDMA-based Gather and Multicast (RGM)

In this scheme, based on the gather-and-broadcast algorithm [13], the barrier operation is divided into two phases. In the first phase called the gather, every process, except the *root*, on arrival at the barrier waits for a message from each child, and then sends a message to its parent. The *root* waits for the gather message from all its children and then sends a broadcast message to each of then and exits the barrier. As each process receives this message, its forwards it to its children and exits the barrier. The process of gather can be done in a hierarchical fashion by imposing a logical tree structure on the processes. The performance of this algorithm depends on the dimension of the tree. The height of the tree varies with the maximum number of children that each parent has. We call this value the fan-in value of the tree. Based on the parameters of the networking interconnect, the fan-in value can be chosen to get optimal performance.

In this two-step technique we use RDMA writes in the gather phase. The processes are arranged in a tree structure. Each process allocates an array of bytes and registers it. The address of this array is exchanged among the processes. The process on entering the barrier, polls for the *barrier_id* to appear in its array at the indices corresponding to each of its children. Once it receives messages from all its children, the process does an RDMA write to the array at its parent process.

When *root* receives all the RDMA messages, it does a hardware multicast to all the processes. The multicast message contains the *barrier_id*. This phase is a one step process, since the multicast primitive is such that the single message gets sent to all the members of the multicast group.

Let us assume that the gather phase is done with a maximum fan-in of l. The value of l is chosen to be a $(power of 2 - 1)$ value, and $l < N$. So the number of levels in the tree created in this phase will be $\lceil \log_{l+1} N \rceil$, and this is the number of hops done by the barrier signal to reach *root*. In the multicast phase just one step is taken by the *root* to signal completion of the barrier to all nodes.

Figure 4.3 shows how this algorithm works for a barrier on 8 processes. Here the gather is done using a tree of height 2, with the value of l as 3, in each step. Process 0 is *root*. The value for l can be chosen based on the number of nodes and the performance of the RDMA write operation. For 8 nodes, the possible fan-in values are 1, 3 and 7. If the fan-in is 1, then the tree becomes a binomial tree. There is a trade-off between the height of the tree and the overhead incurred at each parent node as the number of children increases. For example, for an 8-node barrier, if the tree has maximum fan-in of 1, the height of the tree becomes 3. The time taken for the barrier signal to reach the *root* is at least 3 times the base RDMA write latency. However, the NIC at each parent node has to process only one incoming receive. If the fan-in of the tree is 7, the signal reaches the *root* in a single step. However, the NIC at the *root* has to process 7 incoming RDMA write messages. Based on the base RDMA write latency and the time to process the message by the NIC, the optimal fan-in value can be chosen.

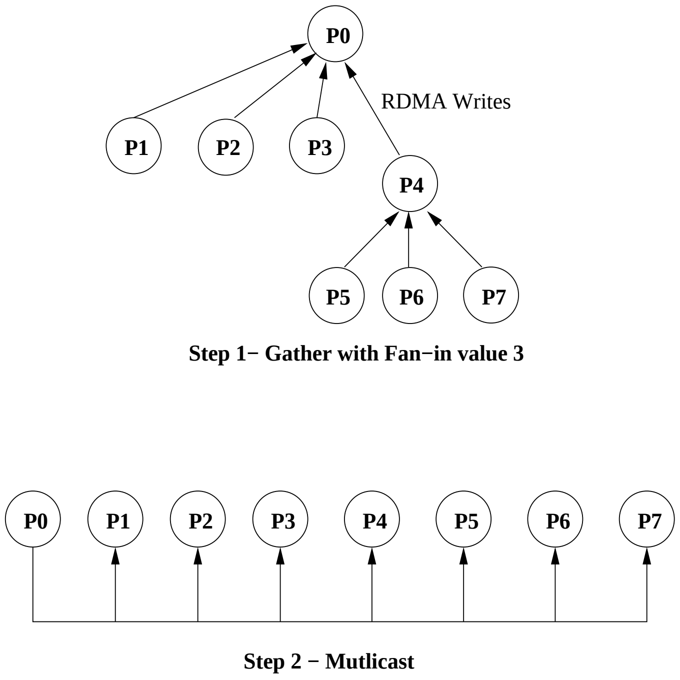


Figure 4.3: Steps in the Gather and Multicast Algorithm for an 8-node Barrier with Fan-In value of 3

4.2 Implementation Details

In this section we present the design choices made and the implementation details for the three barrier algorithms.

4.2.1 Buffer Management and Address Exchange

Since the barrier is a collective call, during the first MPI_Barrier call, all the processes allocate memory for the barrier. The number of elements in the array allocated is N. Each element in this allocated array will be written by the corresponding process using an RDMA write operation. Since every process in the communicator is identified by a *rank* the array elements can be indexed using this *rank* value.

In order to perform an RDMA write, a process needs to provide the remote memory's virtual address and the memory handle that is obtained after the registration of the memory. After the allocation of the buffers, the nodes exchange these addresses and memory handles. This address exchange happens using the send and receive primitives.

This design option seems to be the best among the ones mentioned in Chapter 3, since it ensures that the memory is registered only if the application is involved in collective operations. The overhead is also not increased since the time for address exchange will always be spent, either during the initialization phase, or in the first barrier as is done currently. Once the buffers are allocated, they can be used for all the barriers executed during the life of the process.

4.2.2 Data validation at the receiving end

There is a static count called the *barrier_id* that is maintained by each process. This value is always positive. So during the initialization we assign a negative value to all the array elements. When a process needs a message from a remote process, it polls the corresponding array element. It waits for the value to be greater than or equal to the current *barrier_id*. This is needed to handle cases with consecutive barriers. If one process is faster than the other, it will enter the second barrier before the other can exit the first one. Thus it will write the larger barrier number in the array. Since the barrier number is stored in a 1-byte datatype, we restrict the maximum value for the *barrier_id* to be 127. After every 127 barriers, the static count rolls back to 1. The processes during the poll for the arriving message are aware of this roll-back procedure. This design alternative seems to give the best performance among all those mentioned in Chapter 3.

Figure 4.4 gives a pictorial representation of this implementation using RDMA writes. Here N is 4, and the processes are called P0, P1, P2, and P3. In the first step P0 does an RDMA write of *barrier_id*, in this case 1, to index 0 of P1's array and waits for P1 to write in index 1 of its own array. In the second step it performs the same operations with P2.

4.2.3 Buffer Reuse

The barrier involves no transfer of data between the processes. It only involves a signal from the processes when they arrive at the barrier. Hence just a single data element is needed to indicate the arrival of the process at the barrier. Since the static count of the barriers is maintained, the value written to this data element changes

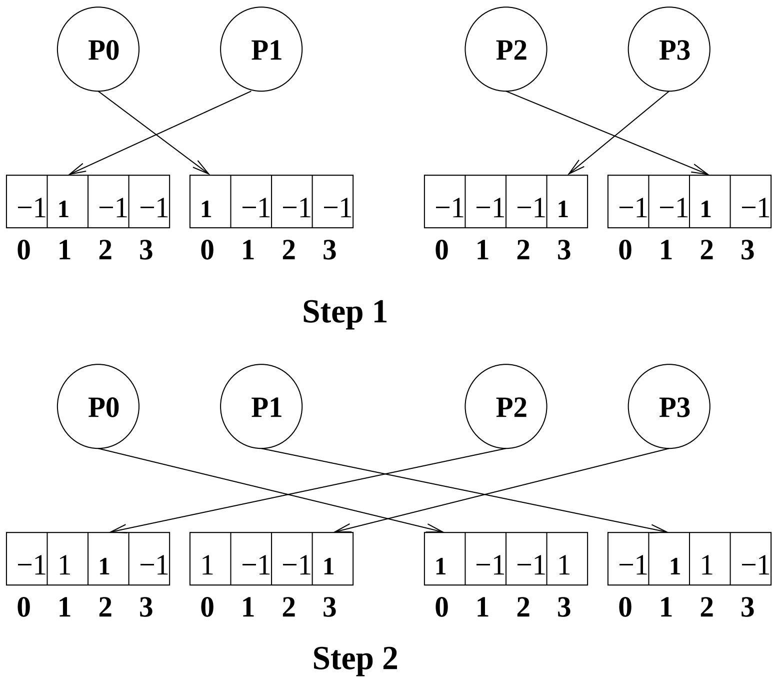


Figure 4.4: Illustration of RPE for a 4-node barrier

with every consecutive barrier. Thus the buffer can be safely reused without any need for reuse notifications to the sender process.

4.2.4 Handling UD multicast messages along with the RC messages

In the broadcast phase of the RGM algorithm we make use of the multicast feature which uses UD packets. This requires that every process create a QP for the UD service type. This is done as a part of the MPI_Init call. The address of the QPs is also exchanged among all the processes. The QP of each process is also attached to the global multicast group in order to enable it to receive messages sent to the multicast group.

In the IB specification the UD messages have the initial 40 bytes assigned for the Global Route Header (GRH). If the RC and UD QPs are associated with the same CQs, and it becomes difficult to distinguish between RC and UD messages based on the data content because we don't know where the actual data starts. Hence we create a separate CQ for the UD work request completions. During the progress check function, we poll on this new CQ also to check if any multicast messages have arrived.

4.2.5 Reliability

Once a process sends the barrier message to its parent in the gather phase, it begins to wait for the multicast message from the *root*. We impose a timeout on this phase. If a message is not received within this period, the process sends a NAK to the *root*. The NAK message is sent using the Send primitive and it contains the "solicit" flag set to true. The NAK message also contains the *barrier_id* that the process is currently waiting to complete.

When the NAK message arrives at the *root*, it triggers the registered completion event handler. The *root* then checks if the message is valid retransmit request by looking at the barrier number. It then does a retransmit of the multicast message for that barrier number.

We have seen in our clusters that the rate of dropping UD packets is very low, and hence this reliability feature is not called upon often. Also, since IBA allows us to specify service levels to QPs, we could assign high priority service levels to the UD QPs. Thus the chances of these messages getting dropped is reduced even further. We also see that in the normal scenarios where there are no packets dropped, there is no overhead imposed by the reliability component. The reliability components do not affect the performance of the critical path under the no-packet drop scenario.

4.3 Integration with MVAPICH

Developing the collective communication library using the low-level primitives requires support at the ADI level for implementing the algorithms. In the traditional implementations, the algorithms were implemented above the ADI.

In our implementation, we move the barrier algorithms to the ADI layer. Based on the compilation options, if the new implementations of the barrier are to be used, the MPI_Barrier() function makes a call to the MPID_RDMA_Barrier() function at the ADI. This function implements the three algorithms mentioned in this chapter. ADI2 does not provide any calls to perform RDMA writes. Hence we implement a function vapi_RDMA_Send() that is used to perform the RDMA writes and the MPID_RDMA_Barrier() function makes use of this call.

We also implement the function to do the UD multicast sends. The progress function DeviceCheck() is modified to poll on the CQ associated with the UD QP.

4.4 Experimental Evaluation

In this section, we present the results obtained for the implementations of the barrier algorithms and compare them with the performance of the existing implementation.

We conducted the performance evaluations on the following set of clusters.

Cluster 1: A cluster of 8 SuperMicro SUPER P4DL6 nodes, each with dual Intel Xeon 2.4GHz processors, 512MB memory, PCI-X 64-bit 133MHz bus, and connected to a Mellanox InfiniHost MT23108 DualPort 4x HCA. The nodes are connected using the Mellanox InfiniScale MT43132 eight 4x port switch. The Linux kernel version is 2.4.7-10smp. The InfiniHost SDK version is 0.1.2 and the HCA firmware version is 1.17.

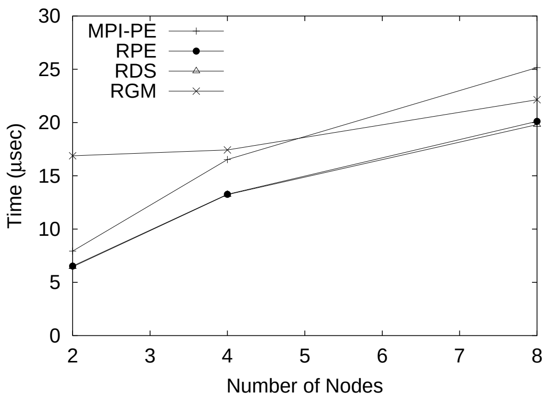
Cluster 2: A cluster of 16 Microway nodes, each with dual Intel Xeon 2.4GHz processors, 2GB memory, PCI-X 64-bit 133MHz bus, and connected to a Topspin [24] InfiniBand 4x HCA [25]. The HCAs are connected to the Topspin 360 Switched Computing System, which is a 24 port 4x InfiniBand switch with the ability to include up to 12 gateway cards in the chassis. The Linux kernel version is 2.4.18-10smp. The HCA SDK version is 0.1.2 and firmware version is 1.17.

The barrier performance was measured by executing the MPI_Barrier() function 1000 times, and the average of the time taken across all the processes was calculated.

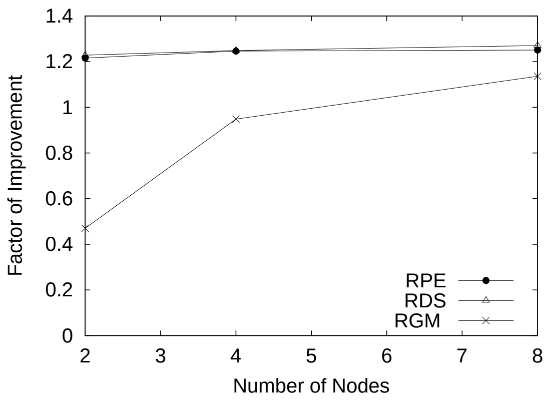
4.4.1 Comparison with MPI-PE

Figures 4.5 and 4.6 show the performance comparisons of the three barrier algorithms, RPE, RDS and RGM, with MPI-PE, the standard pairwise exchange MPICH implementation of the barrier. On the left-hand side we show the absolute values of the barrier latency and on the right-hand side we show the factors of improvement.

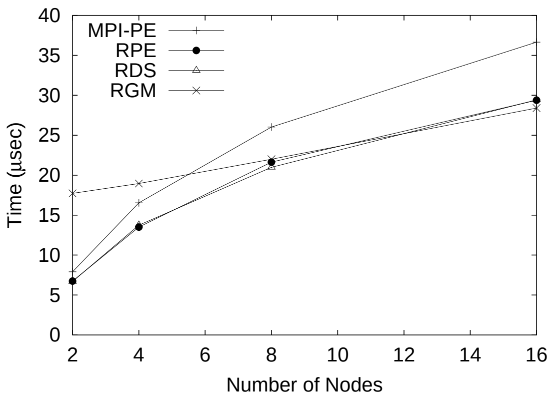
In Figure 4.5(a) and 4.5(c), we consider barriers for only the powers-of-2 group sizes. We see that RPE and RDS perform better than MPI-PE for all the cases. We also see that the latency for the 2-node barrier is very close to the base RDMA latency (about 5.8μs) and the overhead imposed by the MPI layer is just about 0.5μs. For group sizes of 2 and 4, RGM does worse because the base latency of the UD multicast operation is greater than that of a single RDMA write. The performance of RPE



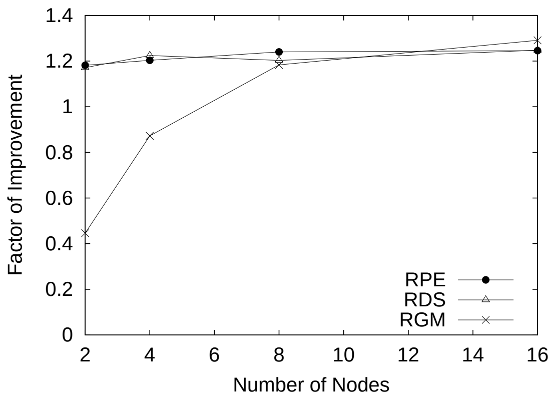
(a) **Latency (Cluster 1)**



(b) **Factor of Improvement (Cluster 1)**



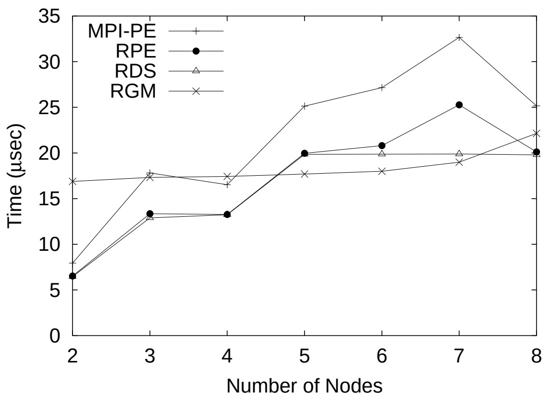
(c) **Latency (Cluster 2)**



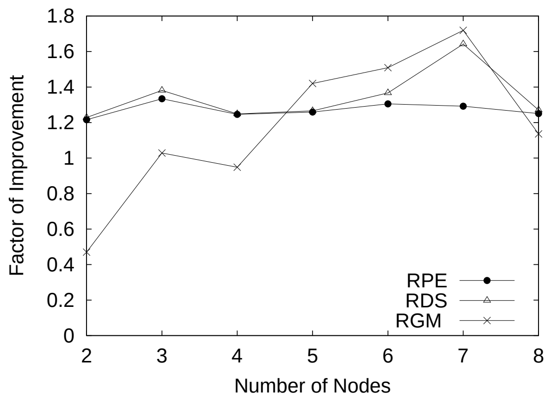
(d) **Factor of Improvement (Cluster 2)**

Figure 4.5: Comparison of MPI-PE with the proposed algorithms for power-of-2 group size on Clusters 1 and 2

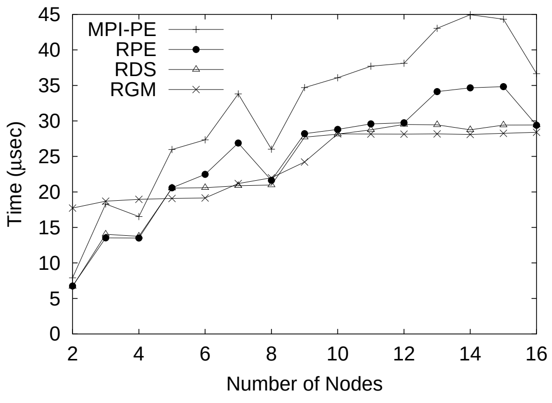
and RDS for powers-of-2 group sizes is very similar. For 8 nodes in Cluster 1, we gain as much as 1.25 factor of improvement with RPE and 1.27 with RDS, as seen in Figure 4.5(b). The benefits of the RGM implementation are seen as the number of nodes increases. The scalability of the multicast operation provides the benefit in these cases. Figure 4.5(d) shows that in Cluster 2, RGM does the best for 16 nodes



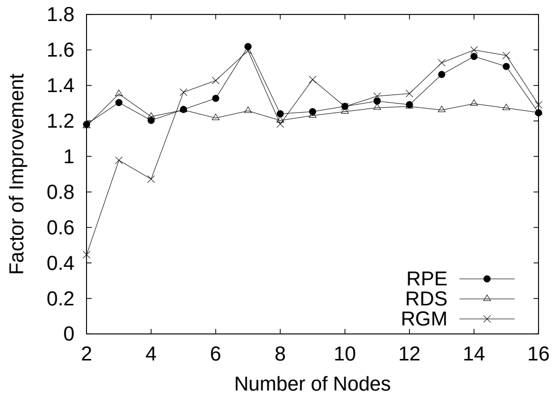
(a) **Latency (Cluster 1)**



(b) **Factor of Improvement (Cluster 1)**



(c) **Latency (Cluster 2)**



(d) **Factor of Improvement (Cluster 2)**

Figure 4.6: Comparison of MPI-PE with the proposed algorithms for all group sizes on Clusters 1 and 2

with an improvement of 1.29. This is because for larger group sizes, RGM has the benefit of the constant time multicast phase.

Figure 4.6 illustrates the performance gains obtained for all group sizes. We see that the pairwise exchange algorithms, MPI-PE and RPE, always penalize the non-power-of-2 cases, and this is not seen in RDS and RGM. Hence on Cluster 1, RDS and RGM gain a performance improvement of up to 1.64 and 1.71 respectively. On

Cluster 2, we see that RGM performs best in most cases and the maximum factor of improvement seen is 1.59. We see that the factor of improvement for RPE is almost a constant in all cases because the benefit is obtained by the constant difference in the latency between a point-to-point send/receive operation and an RDMA-Write/poll operation.

4.4.2 Choosing the optimal fan-in values for RGM

The performance of the RGM algorithm varies with the values for maximum fan-in in the gather phase. As this value decreases, the height of the tree increases and this will increase the number of RDMA writes being done. But if this value is large, the parent node becomes a hot-spot, that could possibly cause degradation in performance. Hence we need to choose the optimal value for the fan-in. From Figure 4.7, we see that the fan-in value of 7 performs the best. Hence for all our performance evaluations we choose this value in the RGM implementation.

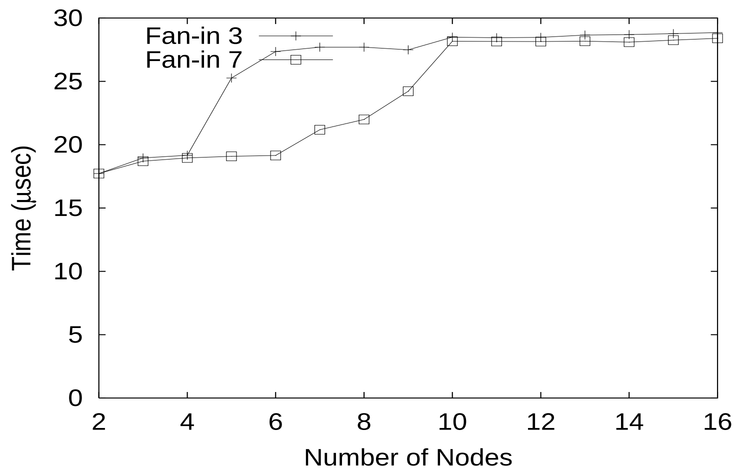


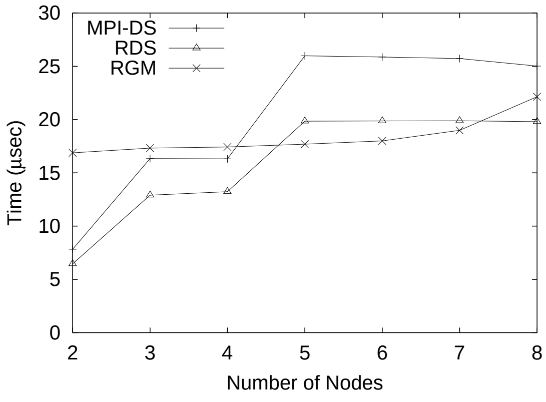
Figure 4.7: Performance of RGM algorithm for varying fan-in values on Cluster 2

4.4.3 Comparison with MPI-DS

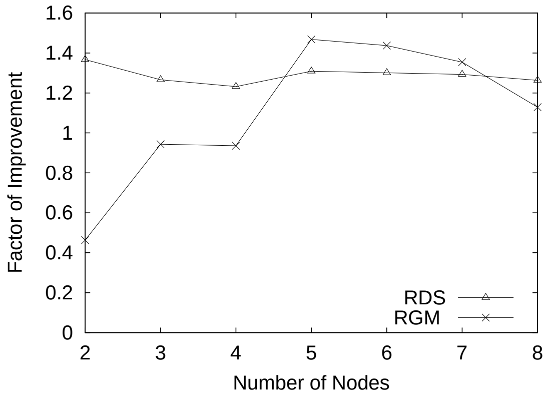
As mentioned earlier, the pairwise exchange algorithm performs worse for non-power-of-2 group sizes because of 2 extra operations. Hence in order to do a fair comparison, we implemented the Dissemination algorithm with the point-to-point MPI functions. We refer to this implementation as MPI-DS. We see that the barrier latencies of the proposed RDMA and multicast based implementations are better than that of MPI-DS too. Figure 4.8 shows the comparison of the RDS and RGM implementations with MPI-DS. It is to be noted that in spite of providing benefits to the current MPI implementation, RDS achieves up to 1.36 factor of improvement, and RGM achieves 1.46 on Cluster 1. We see an improvement of 1.32 with RDS and 1.48 with RGM on Cluster 2.

4.4.4 Summary

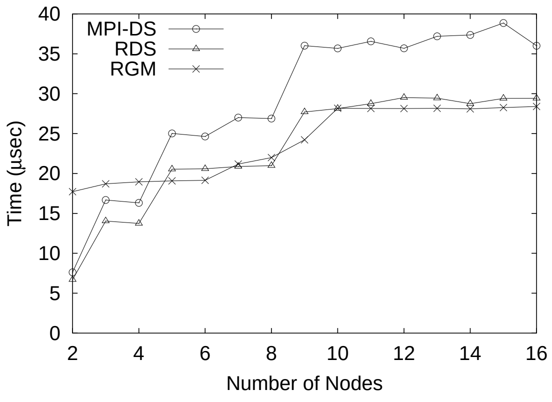
In this chapter we presented three new approaches to implement the barrier operation. We also compared the relative performances of these new implementations. We see that all three schemes outperform the existing implementation of the barrier. We also see that the RDS and RPE algorithms perform well for smaller group sizes, while RGM performs the best for large group sizes.



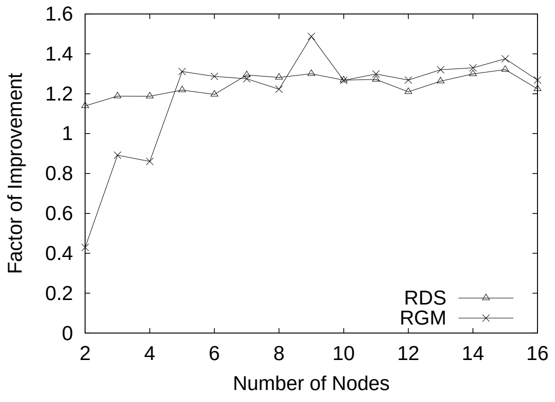
(a) **Latency (Cluster 1)**



(b) **Factor of Improvement (Cluster 1)**



(c) **Latency (Cluster 2)**



(d) **Factor of Improvement (Cluster 2)**

Figure 4.8: Comparison of MPI-DS with the proposed algorithms on Clusters 1 and 2

CHAPTER 5

ALLREDUCE

In this chapter, we focus on the algorithms, design issues, implementation, and experimental evaluations for the Allreduce collective operation. Initially, we provide an introduction to the Allreduce operation and then go into the issues of implementing this routine with the RDMA and multicast support. Since this collective operation involves transfer of data and not just synchronization, as in the case of the barrier, the buffer handling and reuse issues are more complex.

5.1 The Allreduce Collective Operation

The Allreduce is a global computation routine. It is the generalization of another computation routine called the reduce. The reduce routine performs an operation on the data from all the processes and transfers the result to a specified process. In the Allreduce routine, the results are shared with all the processes in the communicator. The operation performed by this routine is either a predefined MPI operation or a user-defined operation.

The syntax for the Allreduce operation according to the MPI standard is as follows:

MPI_Allreduce(void **sbuf*, void **rbuf*, int *count*, MPI_Datatype *stype*, MPI_Op *op*, MPI_Comm *comm*) where:

sbuf is the address of the send buffer

rbuf is the address of the receive buffer

count is the number of elements in the send buffer

stype is the datatype of the elements in the send buffer

op is the reduce operation (either predefined or user-defined)

comm is the communicator

For the Allreduce routine, each process can provide either one element or a sequence of elements. In both cases the combine operation is executed element-wise on each element of the sequence.

MPI provides 12 predefined routines for the reduce operation, the most commonly used ones being summation, minimum, maximum, product, logical and bit-wise operations. All the operations performed by the reduce are assumed to be associative. The MPI predefined functions are also commutative, but the user is allowed to define operations that are not commutative.

The implementation of the Allreduce operation can be altered based on the operation being performed and the type of the target cluster. After the execution of the Allreduce operation, the value in the result buffer at each process should be the same. In the case of heterogeneous systems, variation in floating point arithmetic (like choice of round-off mode), could lead to varying results when the results are computed on different nodes. Hence for heterogeneous clusters, this imposes the restriction that a single process calculates the final result and broadcasts it to all the processes. In the case of homogeneous clusters, we can come up with faster algorithms where the calculation of results is done in a distributed manner. In this chapter we propose two techniques, one for the heterogeneous systems, and other for homogeneous systems.

Let us consider an example of an Allreduce operation on 4 nodes. The operation being performed is MPI_MAX, which calculates the maximum value. The operation is illustrated in Figure 5.1.

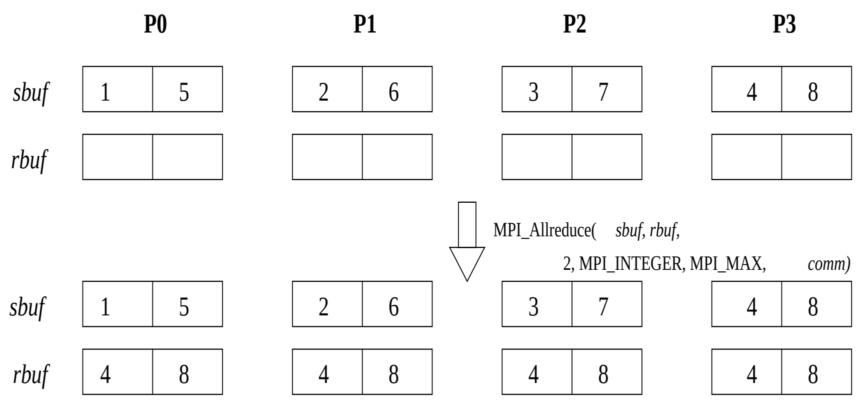


Figure 5.1: Illustration of buffer contents before and after the Allreduce operation on 4 nodes

We see that after the operation, the maximum among the values seen for each element are stored in the receive buffer of all the processes.

5.2 Allreduce Algorithms

As mentioned earlier, the types of algorithms used to implement Allreduce may differ on how the result is calculated. Some algorithms perform the calculation in a distributed manner, wherein there is no need for a final broadcast phase. Other algorithms perform the operation such that one process has the result and it broadcasts it to all the other processes. In this section we present two algorithms - one RDMA write based algorithm that does the result computation in a parallel manner, and the other RDMA and multicast based scheme which follows a reduce and broadcast pattern.

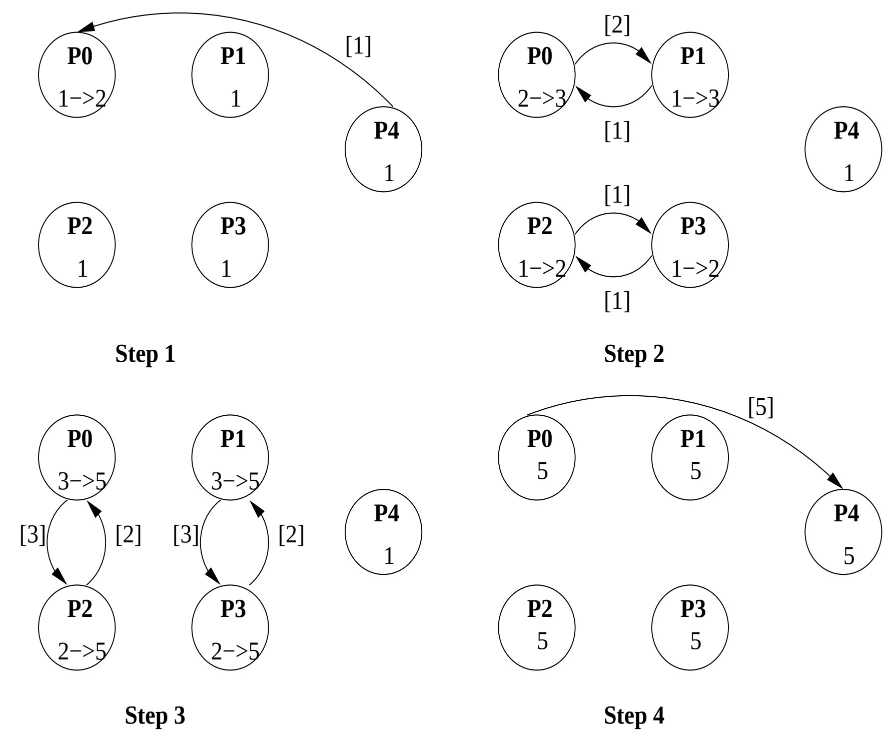
5.2.1 RDMA based Pairwise Exchange (RPE)

This algorithm is similar to the one discussed in Section 4.1.1. The results are calculated in a parallel manner at all the processes. The processes are paired up and they exchange the contents of their send buffers with each other. Once a process gets the data from its partner, it combines its own data with the data received using the specified reduce operation. In the next step it exchanges this newly calculated data with its partner. This cycle of data exchange, intermediate result calculation and grouping of processes continues until all the processes are merged into one group. At this point, every process has reduced the data from all the processes in its local buffer. Thus the Allreduce operation can complete simultaneously on all the processes.

Let us consider Figure 5.2 to understand how this algorithm works. The operation is an MPI_SUM on one integer element, and it is performed using data from 5 processes.

We see that in the first step P4 sends its data to P0 and P0 reduces this data with its own data. Then P0, P1, P2, P3 perform the pairwise exchange and reductions. Finally P0 sends the result to P4. P4 simply retains the data received as the final result and does not perform any reductions itself.

The number of steps performed for this operation is $\log_2 N$ for powers of 2 values of N. In the case of non-powers of 2 values of N, processes in set G' (described in Section 4.1.1) send the data to the processes in set G. The processes in G perform



[x] − x is the data sent using RDMA wite
x−>y − x and y are the data before and after the reduction respectively

Figure 5.2: Allreduce operation for MPI_SUM on one data element using the RPE algorithm for 5 processes

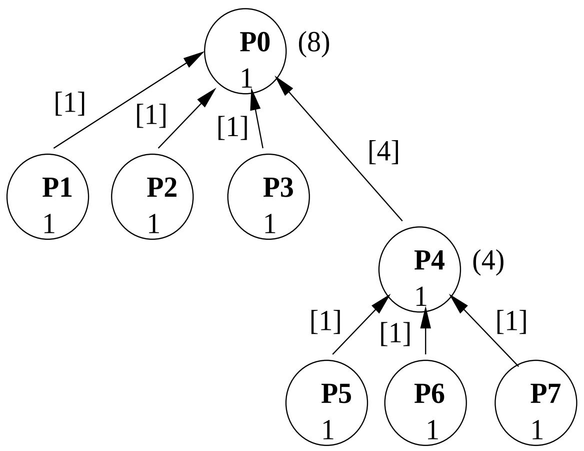
the operation and pairwise exchanges. The final result is sent back to the processes in G', which do not perform any reductions themselves. The number of steps taken in this case is $\lfloor \log_2 N \rfloor + 2$.

This algorithm can be implemented using just the RDMA write calls. In each step a process RDMA writes the data to the remote buffers. The receiving process then uses this data in the reduction operation and sends out the results using RDMA write in the subsequent step.

5.2.2 RDMA based Gather and Multicast (RGM)

In this case the Allreduce operation is implemented as a reduce followed by a broadcast. The node with rank 0, is chosen as the *root* and the reduction is done at this process. After the reduction *root* broadcasts the results to all the processes. We make use of the RDMA write operations in the reduce phase and make use of the multicast primitive for the broadcast phase.

The reduction phase involves gathering of data using a logical tree pattern of the processes. The height of the tree is decided by the maximum fan-in value (maximum number of children) at each parent node. We represent this value by l and choose it in a manner as described in Section 4.1.3. As the fan-in value increases, the height of the tree reduces, but the number of reductions to be performed at each parent node increases.



[x] − Data sent with the RDMA write
(y)− New data after the reductions

Figure 5.3: Reduce phase of the RGM algorithm

Figure 5.3 shows how the reduction happens for an 8-node Allreduce operation using a tree with l as 3. The operation performed is MPI_SUM and the data is a single integer.

Here P0 is the *root* and the final result is calculated at P0. Once it has the result, P0 sends it to all the processes using the multicast primitive.

5.3 Implementation of RPE

We now discuss the design solutions for the RPE algorithm using the RDMA write primitives. There are two protocols being used in this implementation, based on the size of the data being operated on and for each design issue we consider the two cases.

5.3.1 Buffer Management

The RPE implementation is done using two modes based on the size of the data. For data of size less than or equal to 2K we make use of the "single-transfer" scheme. For sizes between 2K-4K we make use of the "notify-buffer" scheme. In both the schemes the buffers are allocated statically. The allocation, registration and address exchange happens during the first Allreduce call. Once these buffers are allocated they are used all through the life of the application using the buffer reuse mechanisms.

Small messages

For this mode, the size of the data is less than or equal to 2K bytes. Each process allocates two contiguous buffers, each of size N*(2K+1) bytes. We refer to these buffers as the *odd* buffer and *even* buffer. Each buffer is then broken down into N blocks of 2K+1 bytes each. We need only N blocks because there are only a maximum of $N-1$ other processes that will need to write data to each process. For transferring the data in this scheme, the sender i RDMA writes to the block i of either the *odd* or *even* buffers of the receiver.

Messages between 2K-4K bytes

In this mode, each process allocates two contiguous buffers, each of size N*4K bytes. These are also called the *odd* and *even* data buffers. Each process also allocates two buffers, each of size N bytes and these are referred to as the *odd* and *even* notify buffers. These buffers are used to ensure the completion of data transfer at the receiving end. The addresses of both the data and notify buffers is exchanged during the first Allreduce call involving data of size between 2K and 4K.

5.3.2 Data reception at the receiver end

Since the data transfer happens using RDMA writes, we need schemes by which the receiver can recognize the arrival of data. We make use of the polling technique, where the receiver polls on the contents of the local buffers for a special value to be written. In order to use this scheme, all the buffers allocated are initialized to -1. The special value that is to be polled on is the *allreduce_id* which is a static count of the current Allreduce number.

Small messages

In this case, whenever data is transferred, the *allreduce_id* is appended as the last byte of the data. The data is written to the receiver in a bottom-fill manner so that the last byte of the data block contains the *allreduce_id.* The receiver is aware of the current *allreduce_id* and so it can poll on the last byte of the data buffer for the *allreduce_id* to appear. The MTU of data transfer for the IBA links is 2K bytes, and so all the data fits into a single packet. Thus the NIC transfers all the data at once.

This means that the appearance of the *allreduce_id* in the last byte of the data block is a proof of the data having arrived at the receiver's buffer.

Messages between 2K-4K bytes

In this case, the data is larger than 2K, and so the transfer of the data at the physical layer happens in terms of multiple packets. The NIC receives these packets and DMAs the data to the registered buffer. Though there is order maintained between messages on a reliable connection, there is no guarantee on the order of arrival of packets belonging to a single message. Hence polling on the last byte of data does not assure us of the completion of data transfer. Hence we make use of the property that order is maintained between messages. The sender always sends the data to the receiver's data block using RDMA write. Then the sender does a second RDMA write to the notify buffer of the receiver. This message contains the *allreduce_id* of the current operation. The receiver instead of polling on the data blocks as in case of small messages, now polls on the notify buffers for the appearance of the *allreduce_id*. Once the poll is successful, the receiver is assured that the previous message which contained the data has also arrived. Thus the receiver can proceed to operate on the data that has arrived.

5.3.3 Buffer Reuse

As mentioned in Section 5.3.1, each process allocates two sets of buffers, *odd* and *even*, for both data and notification values. This "double-buffer" scheme is used in order to ensure safe reuse of buffers without the need for any additional reuse-notification messages.

During every Allreduce operation, the sender process checks the current *allreduce_id*. If it is an odd value, it does the RDMA write to the *odd* buffer at the receiver process and to the *even* buffer otherwise. Let us see why this is ensures safety of the buffer contents. Assume that the processes execute the first Allreduce operation. In this case all the data is being written to the *odd* buffers and the reduce operations are also carried out in these buffers. During the second Allreduce operation, all the data is written to the *even* buffers. At the completion of the second Allreduce operation, every process is assured that the other processes have definitely exited the first Allreduce operation. This means that the contents of the *odd* buffers are no longer useful and maybe overwritten. Thus the scheme of using the double-buffers alternatively ensures that the data is always being written to a free data block.

Let us now consider an example of an Allreduce operation across four processes to understand the various steps in this implementation. The operation we consider here is MPI_SUM.

Small messages

The various steps performed for the Allreduce operation are as follows.

Step 1: There are four processes in the communicator and hence each process allocates two buffers of size 4*(2K+1) bytes. The buffers are registered and the addresses of the buffers are exchanged. Assume the current value of *allreduce_id* is 1. Process i copies the data from the user-specified $sbuf$ to block i of the *odd* buffer. It copies the data in a bottom-fill manner and the last byte of block i is the *allreduce_id*. Figure 5.4 illustrates this step.

Step 2: In this step each process i, RDMA writes the contents of its data block into the block i of the *odd* buffer at its partner process j. For example P0, writes to block 0 in the *odd* buffer set of P1. Each process then polls for the data to arrive at block j from the partner process j. Once the data has arrived, the process does the reduce of the data. The result is stored in block j itself. Figure 5.5 illustrates this step.

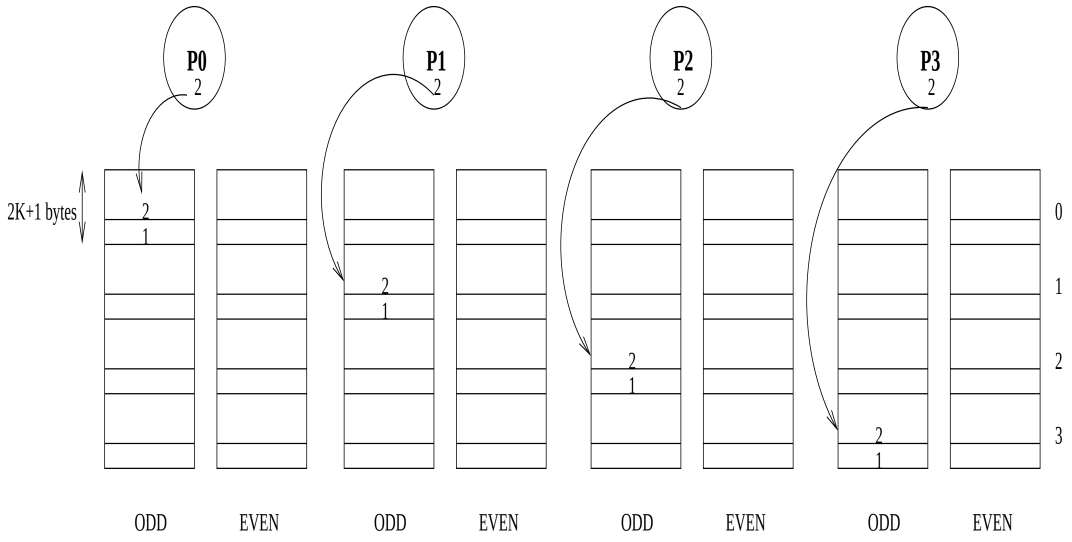


Figure 5.4: Step 1 of Allreduce using RPE for small messages - Initial buffer allocation and copy from user-buffer

Step 3: In this step, the second round of pairwise exchange of messages takes place. The processes do an RDMA write of the intermediate result calculated in the Step 2 to the partner processes of this round. For example, P0 writes the data from block 1 to the block 0 of P2. It then polls for data to arrive from P2 in block 2. Once the data arrives, it adds the data contents of block 1 and 2 and the results are stored in block 2. Figure 5.6 illustrates this step.

Step 4: This is the last step of the operation. All the reductions have been done and the data is available in the block to which the last RDMA write was done from

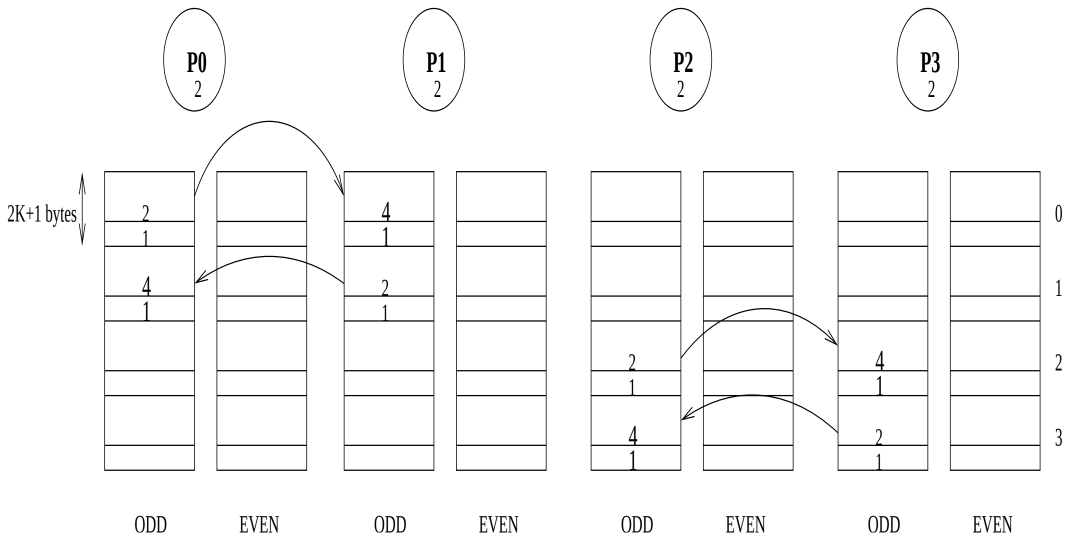


Figure 5.5: Step 2 of Allreduce using RPE for small messages - First round of pair-wise exchange and reduce

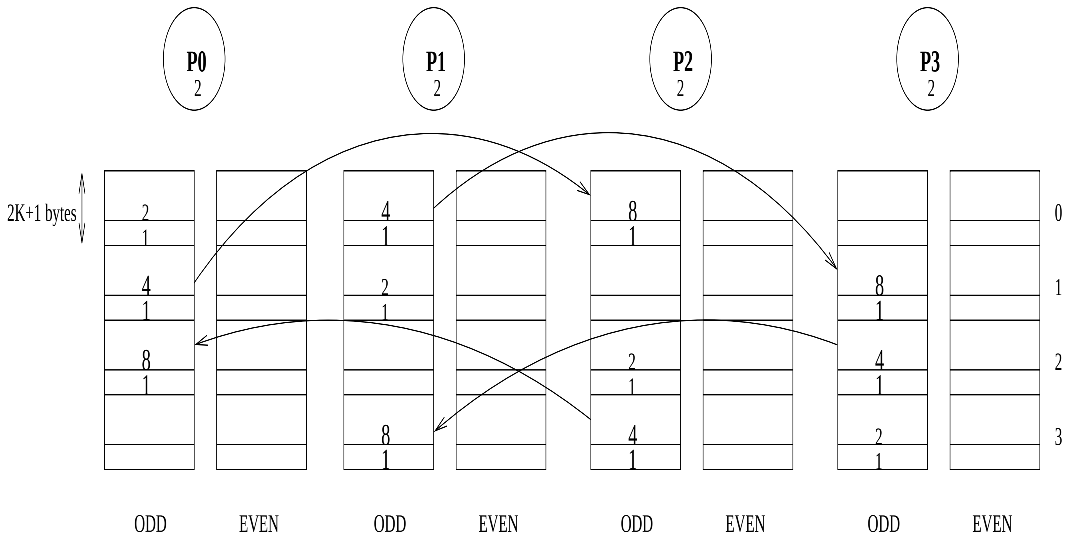


Figure 5.6: Step 3 of Allreduce using RPE for small messages - Second round pair-wise exchange and reduce

the remote process. The results are now copied by each process to the user-specified $rbuf$. This indicates the completion of the Allreduce operation. Figure 5.7 illustrates this step.

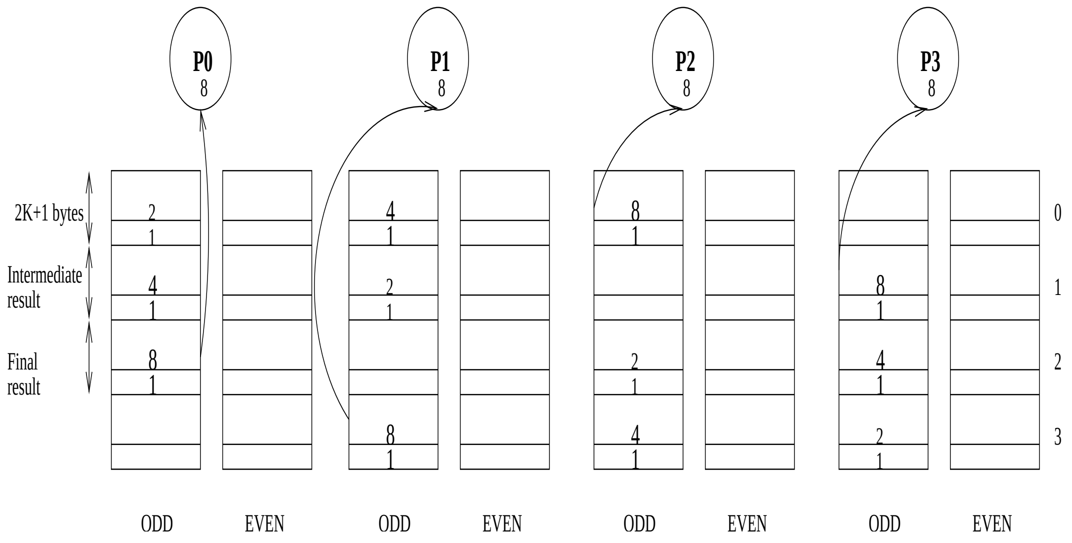


Figure 5.7: Step 4 of Allreduce using RPE for small messages - Final copy of results to user buffer

Messages between 2K-4K

The steps involved in the RDMA writes and reductions of the data are similar to the ones seen for the small messages. However, in this case the data is not written to the blocks in a bottom-fill manner. Every time the data is transferred using RDMA write, it is followed by a second RDMA write to the notify buffer. A process polls on the notify buffer instead of the data buffer to identify data arrival. Figure 5.8 illustrates the buffer allocations and RDMA writes performed in this case.

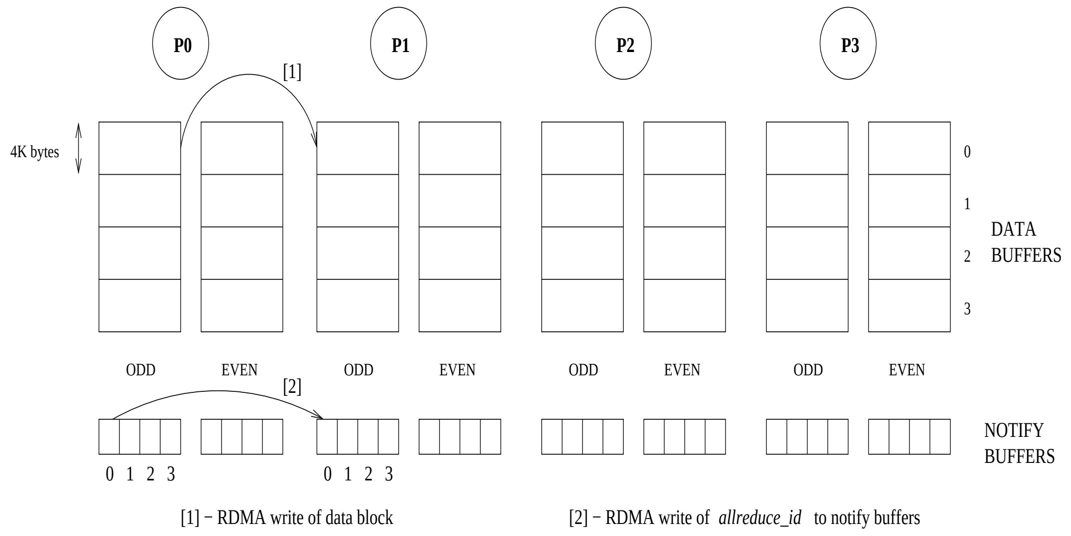

P0
P1
P2
P3
[1]
4K bytes
0
1
2
3
DATA BUFFERS
ODD
EVEN
[2]
NOTIFY BUFFERS
0 1 2 3
[1] – RDMA write of data block
[2] – RDMA write of allreduce_id to notify buffers

Figure 5.8: Buffer allocation and order of RDMA writes in RPE implementation for messages of size 2K-4K bytes

5.4 Implementation of RGM

In this section we discuss the issues involved in the implementation of the Allreduce operation using the RGM algorithm that makes use of the RDMA write and multicast operations.

The maximum size of the message that can be sent using the UD service type is equal to the MTU, which in our IBA systems is 2K bytes. Hence we consider the Allreduce operations to be performed only on data of this size less than 2K bytes.

For the implementation of the reduce operation using the RDMA writes, the buffer handling and reuse mechanisms are similar to the schemes described in the implementation of RPE algorithm for small messages. The "double-buffer" schemes are used to ensure safe reuse. The data is written to the blocks in bottom-fill manner and the data arrival is identified by polling till the last byte of the data block contains the current *allreduce_id*. The reduce takes place in the form of a tree and the *root* contains the final result for the reduction.

For the second phase of the RGM algorithm we make use of the multicast primitive. Every process creates a QP of the UD service type and attaches it to the multicast group. The *root* sends out the result of the reduce data using this QP. All the non-root processes wait for the UD multicast packet to arrive. In the following subsection we discuss how the reliability is provided for these multicast messages.

5.4.1 Reliability for the UD messages

As mentioned earlier, the buffers for the reduce operation are always used in an alternate fashion. That is, when the *odd* buffers are being used, the *even* buffers contain the data of the previous reduce operation. This property is useful because there might be a need to retransmit the results of the previous operation.

When all the processes send out their reduce messages up the tree, they start a timer to wait for the reply from *root*. If the results do not arrive within the time-out period, they send a NAK message containing the *allreduce_id* to *root*. The NAK message contains the "solicit" flag set, and hence an event-handler is invoked at the *root*. This event handler checks for the *allreduce_id* and does a retransmission from the appropriate buffers.

5.5 Experimental Evaluation

In this section we discuss the various results obtained for the different implementations of the Allreduce operation.

The time taken for the operations was based on the average time taken across all the process for 1000 iterations of MPI_Allreduce. For our results we have used the

MPI_INTEGER datatype and the reduction being performed is MPI_SUM. The size of the MPI_INTEGER datatype is 4 bytes.

We have conducted our evaluations on the two clusters mentioned in Section 4.4.

In the following subsections we refer to the existing MPICH implementation of the Allreduce operation using the gather and broadcast scheme as MPI-GB. In order to perform a fair comparison with the pair-wise exchange implementation, we implemented the PE algorithm using MPI point-to-point calls and this is referred to as MPI-PE.

5.5.1 Performance of Allreduce using RPE

In this section we just present the results for the Allreduce implementation based on the PE algorithm. We show the time taken to perform the Allreduce operation MPI_SUM on 1 to 1024 integers. We also increase the number of processes that are performing the Allreduce. Up to 512 integers, the implementation uses the "small message" mode of operation and for the 1024 integers case it uses the "notification" mode of operation. Figure 5.9 presents the results measured on Cluster 1. The time to perform an Allreduce on 1 integer between 2 nodes is just about 6.4μs, which adds about 0.5μs overhead to the base RDMA latency value. We see that time taken for the 1024 integers (4096 bytes) case is considerably high and this is because we are making use of two RDMA writes to indicate completion for messages of size 2K-4K bytes. Figure 5.10 presents the results measured on Cluster 2.

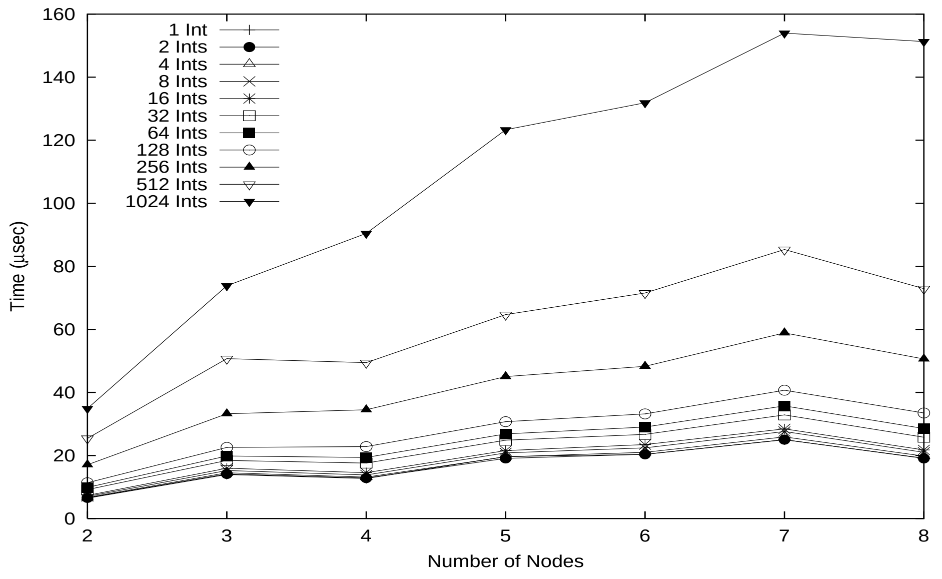


Figure 5.9: Performance of RPE on Cluster 1 for varying message sizes as the number of nodes increases

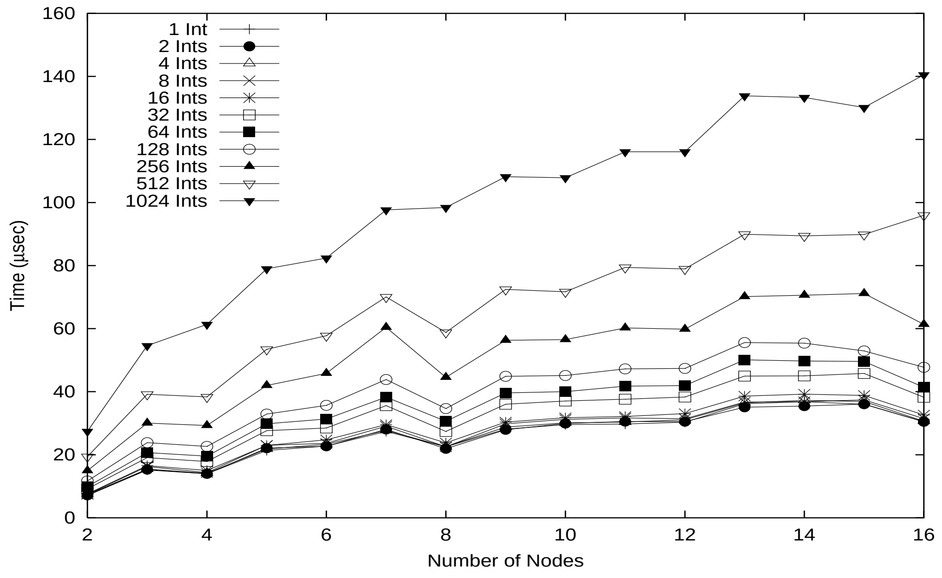


Figure 5.10: Performance of RPE on Cluster 2 for varying message sizes as the number of nodes increases

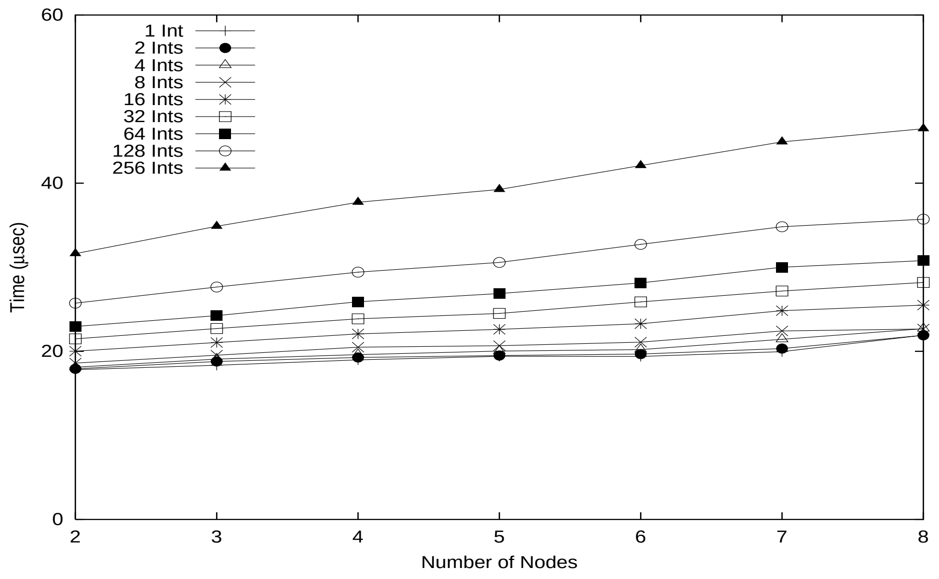


Figure 5.11: Performance of RGM on Cluster 1 for varying message sizes as the number of nodes increases

5.5.2 Performance of RGM

We now present the results of the performance of the RGM implementation on the two clusters. The fan-in value for the tree being constructed in the reduce phase is 7, since we saw that this gives the least latency in Section 4.4.2. Figure 5.11 shows the performance of the RGM implementation on cluster 1. We consider data sizes of up to 256 integers only because the maximum amount of data is limited by the maximum UD message size, which is about 2K bytes. We see that the latency increases very gradually as we increase the number of nodes. This is because the multicast phase of the Allreduce takes a constant time across all number of nodes, and the only increase is caused by the reduction phase.

5.5.3 Comparison of RPE with MPI-PE

We now compare the performance of the proposed RPE implementation based on the RDMA writes with implementation of the pair-wise exchange algorithm using the MPI point-to-point messaging calls. We consider the 8 node case for data size of 256 integers. We see that by making use of the faster low-level protocols, there is a lot of benefit gained. Figure 5.12 shows that the RPE scheme performs better for all cases. We get up to a 1.19 factor of improvement by using the RPE implementation.

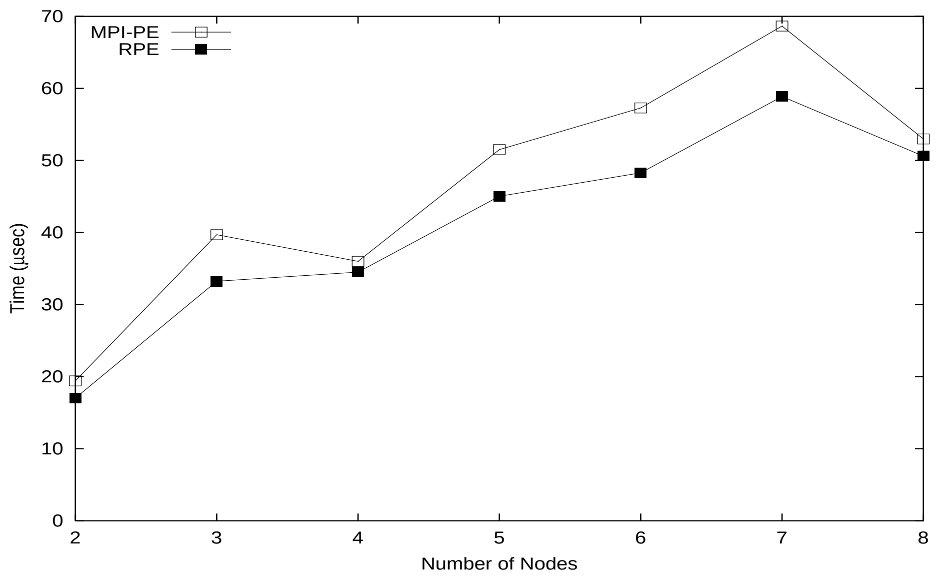


Figure 5.12: Comparison of 256-integer Allreduce performance of MPI-PE and RPE on Cluster 1

5.5.4 Comparison of RGM with MPI-GB

In this section we compare the performance of the two gather and broadcast based schemes, that is, RGM and the existing MPI-GB. Figure 5.13 clearly shows the benefit gained by using the RDMA and multicast operations for implementing the Allreduce.

The scalability of this scheme is seen from the fact that the latency for RGM increases at a much slower rate than that for MPI-GB. The factor of improvement is as high as 2.01 for the 8 node case.

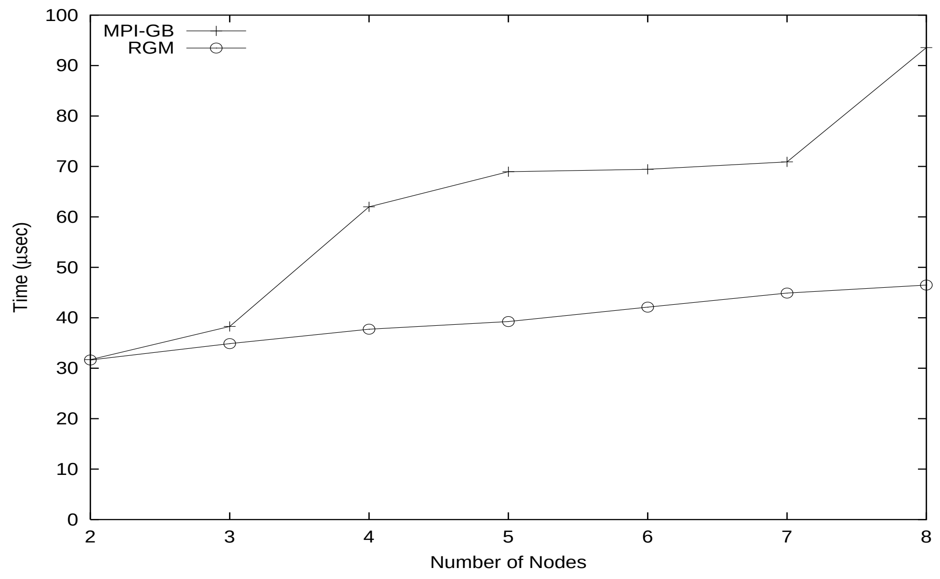


Figure 5.13: Comparison of 256-integer Allreduce performance of MPI-GB and RGM on Cluster 1

Figure 5.14 shows the comparison of RGM and MPI-GB on the 16 node cluster for a 64 integer Allreduce operation. We have chosen this data size since for larger data size we were seeing some instability on this cluster. We see that even in this case the RGM implementation gives a very good improvement of up to 2.06 times.

5.5.5 Comparison of all the Allreduce implementations

Now we consolidate the comparison of two MPI point-to-point based implementations with the two proposed implementations. As seen in Figure 5.15(a), the RDMA based schemes always perform better. The MPI-PE performs better than MPI-GB

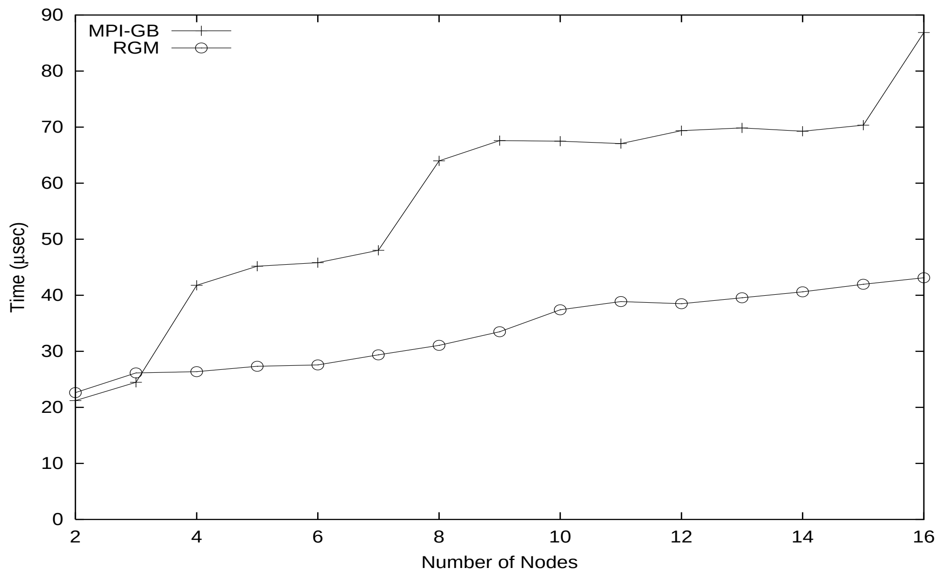
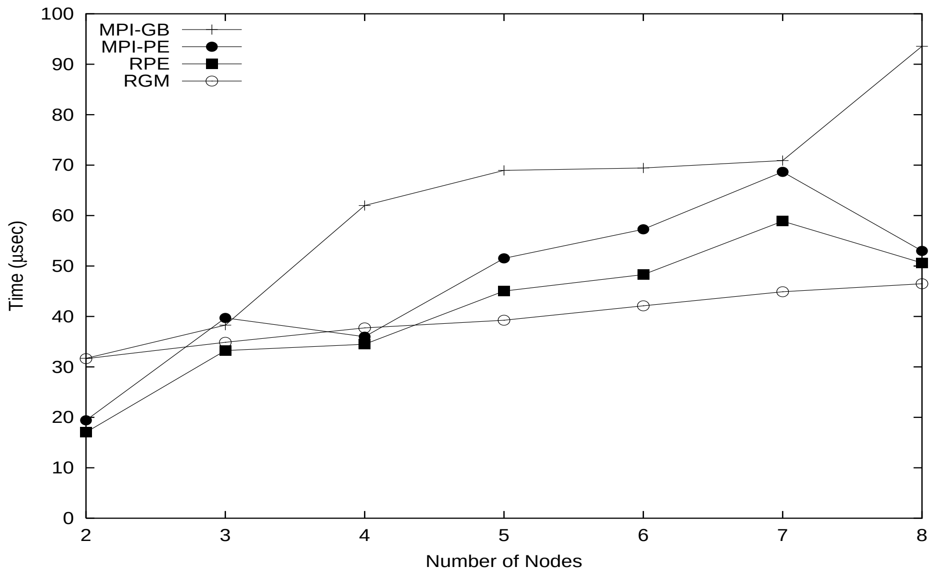


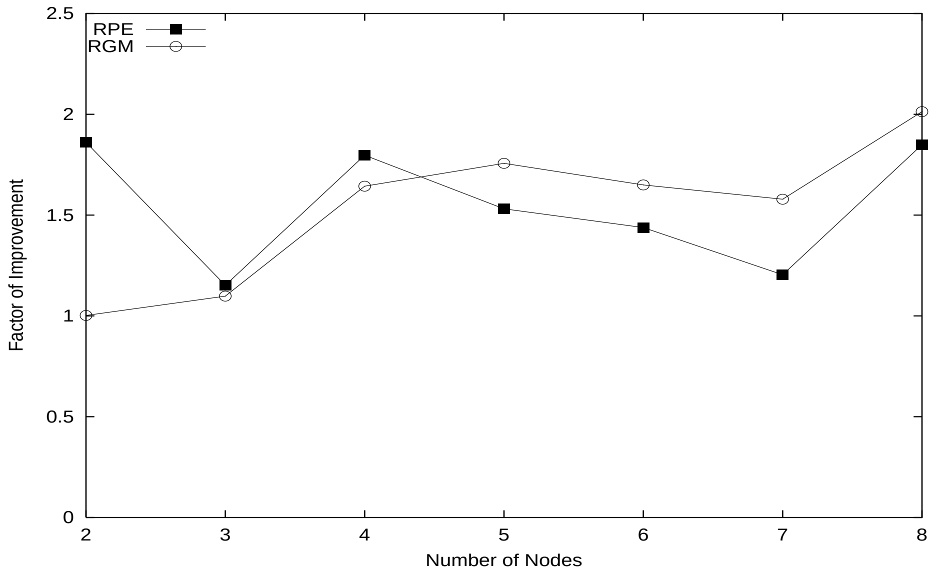
Figure 5.14: Comparison of 64-integer Allreduce performance of MPI-GB and RGM on Cluster 2

because the results are calculated in a distributed manner and there is no extra broadcast phase required. However, the pair-wise exchange algorithms cannot be used in heterogeneous clusters as mentioned earlier. We see that for small number of nodes, RGM does worse than RPE and this is because the base UD multicast latency is high and can be amortized only as the number of nodes increases. As the cluster size increases, we notice that the RGM algorithm does the best. Figure 5.15(b) shows the factor of improvement gained by these RDMA based implementations over the standard MPICH implementation, MPI-GB. We see that RGM gives a factor of improvement of about 2.01 and RPE gives a factor of improvement up to 1.86.

We also compare the performances of the implementations on Cluster 2. Figure 5.15(c) shows the absolute values of the latencies while Figure 5.15(d) gives the factors of improvement. We consider the data size to be 64 integers (256 bytes). RPE



(a) **Comparison of 256-integer Allreduce performance of RPE and RGM with MPI-GB and MPI-PE (Cluster 1)**

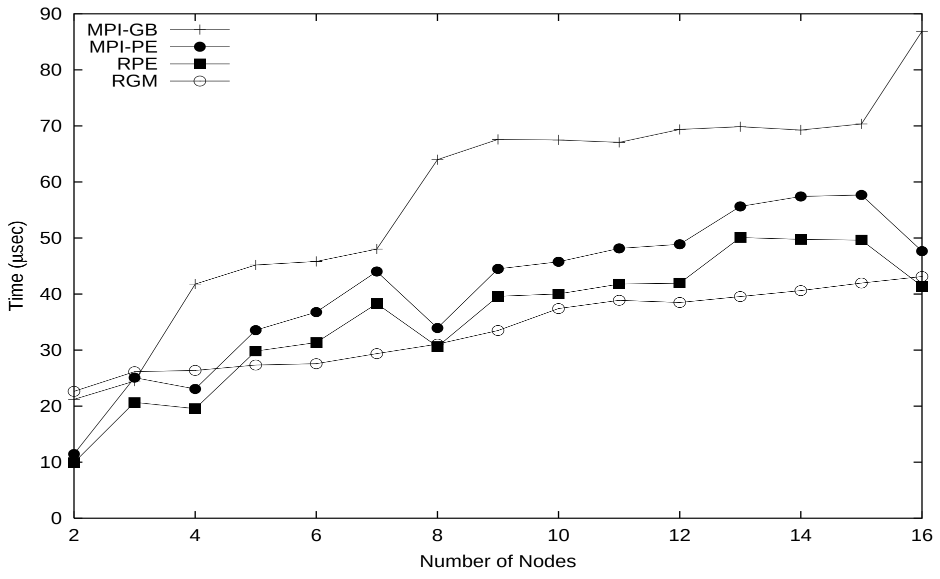


(b) **Factor of improvement of the RPE and RGM implementations over MPI-GB (Cluster 1)** improves the performance by up to 2.14 times, while RGM improves the performance 2.06 times.

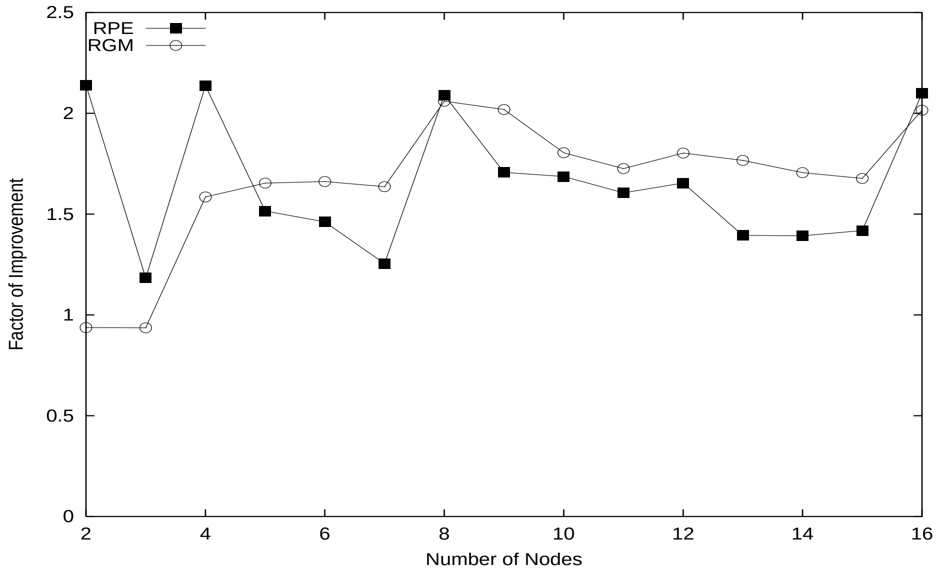
Thus we see that the proof of concept implementations of the Allreduce operation using the efficient primitives provided by IBA perform considerably better than the existing implementations.

5.6 Summary

In this chapter, we introduced the algorithms that can be used to implement the Allreduce operation and described their implementation using the features of IBA. We evaluated the proposed implementations with the existing Allreduce operation and achieved a factor of improvement of up to 2.01 using RGM and 1.86 using RPE on the 8 node cluster. For the 16 node cluster we see improvements of up to 2.14 using RPE and 2.06 using RGM.



(c) **Comparison of 64-integer Allreduce performance of RPE and RGM with MPI-GB and MPI-PE (Cluster 2)**



(d) **Factor of improvement of the RPE and RGM implementations over MPI-GB (Cluster 2)**

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this dissertation, we proposed some novel schemes of implementing the collective communication operations specified by the MPI standard. We made an attempt to leverage the fast and scalable primitives offered by the IBA software and hardware to improve the performance of the collective operations.

We have pursued the idea that collective operations can be made more efficient if designed and implemented using lower-level primitives at the ADI layer, instead of using the MPI point-to-point calls above the ADI. The idea stems from the need to have tighter control on the buffer management and to avoid unnecessary data copies in the algorithms.

The design issues that are to be handled during the development of the collective communications library, and the pros and cons of each of the design alternatives has been discussed. The various buffer management, buffer reuse and data validity issues were analyzed and the best options were chosen in the implementations. For the multicast operations, we also incorporated the reliability scheme and ensured that the reliability component does not degrade the performance of the critical path in the no packet drop scenarios.

We designed and implemented the Barrier and Allreduce collective operations as a proof of concept. With the barrier, we see a factor of improvement of 1.29 for a 16-node cluster and up to 1.71 for non-powers of two group size barriers. With the allreduce, we are achieving up to 2.06 factor of improvement over the existing implementations for the 16-node cluster.

6.1 Future Work

This thesis concentrated on the barrier and the allreduce operations. We are also working on extending these ideas to implement other collective operations like broadcast and all-to-all. There are also other features of IBA, like the RDMA read, atomic operations, etc., that can potentially provide performance improvements for collective operations. There is scope to develop new algorithms using these primitives for the collective operations.

The global allocation and management of buffers for processes belonging to different communicators is another issue that needs to be investigated, since this can have an impact on the scalability of the library. The allreduce algorithms we tested only with 4K block sizes, but this scheme can be easily extended to handle larger data sizes.

The results presented in this thesis are in the form of latencies for each of the collective operations. We are planning to run some parallel benchmarks to evaluate the performance benefits that are achievable at the application level too.

BIBLIOGRAPHY

[1] Mohammad Banikazemi, Rama K. Govindaraju, Robert Blackmore, and Dhabaleswar K. Panda. MPI-LAPI: An Efficeint Implementation of MPI for IBM RS/6000 SP Systems. *IEEE TPDS*, pages 1081–1093, October 2001.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network.

[3] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[4] H. Frazier and H. Johnson. Gigabit Ethernet: From 100 to 1000Mbps.

[5] G. H. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and Experience with LAPI: A New High Performance Communication Library for the IBM RS/6000 SP. *IPPS '98*, March 1998.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[7] Lars Paul Huse. Collective communication on dedicated clusters of workstations. In *PVM/MPI*, pages 469–476, 1999.

[8] GigaNet Incorporations. cLAN for Linux: Software Users' Guide. 2001.

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

[10] Lawrence Livermore National Laboratory. MVICH: MPI for Virtual Interface Architecture, August 2001.

[11] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit Noronha, Peter Wyckoff, and Dhabaleswar K. Panda. MPI over InfiniBand: Early Experiences. Technical Report, OSU-CISRC-10/02-TR25, January 2003.

[12] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *International Conference on Supercomputing '03*, June 2003.

[13] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers,. Technical Report MSU-CPS-94-35, 94.

[14] Mellanox Technologies. http://www.mellanox.com.

[15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM ToCS*, 9(1):21–65, 1991.

[16] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[17] Network Based Computing Lab, The Ohio State University. MPI for InfiniBand on VAPI layer, http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.

[18] P. Shivam, P. Wyckoff and D. K. Panda. EMP:Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Supercomputing '01*, November 2001.

[19] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *the Proceedings of Hot Interconnects '01*, August 2001.

[20] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters. In *International Parallel and Distributed Processing Symposium '03*, April 2003.

[21] R. Gupta, V. Tipparaju, J. Nieplocha and D. K. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *Cluster 02*, September 2002.

[22] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.

[23] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Frontiers '96*. IEEE Computer Society, Oct 1996.

[24] Topspin Communications, Inc. http://www.topspin.com/.

[25] Topspin Communications, Inc. Topspin InfiniBand Host Channel Adapter.

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

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *ISCA*, pages 256–266, 1992.