Efficient Collective Communication using Remote Memory Operations on VIA-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

Rinku Gupta, B.E.

* * * * *

The Ohio State University

2002

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. P. Sadayappan

Dr. Pete Wyckoff

Approved by

Adviser
Department of Computer
and Information Science

ABSTRACT

Most high performance scientific applications require efficient support for collective communication. Point-to-point communication in current generation clusters are based on Send/Recv communication model. Collective communication operations built on top of such point-to-point message-passing operations might achieve suboptimal performance. VIA and the emerging InfiniBand support remote DMA operations. Such operations not only allow data to be moved between the nodes with low overhead, they also allow to create and provide a logical shared memory address space across the nodes. This feature demonstrates potential for designing high performance and scalable collective operations.

In this thesis, we discuss the various design issues and alternatives that may be the basis of a RDMA supported collective communication library. We discuss issues related to buffer management, data identification and validity at the receiver end. As a proof of concept, we have designed, analyzed and implemented three collective communication operations namely *1. Barrier, 2. Broadcast and 3. AllReduce.* For the RDMA AllReduce, we introduce special algorithms called as the Degree-k tree based AllReduce algorithms which when combined with the RDMA mechanism give improved performance as compared to the message passing algorithms based on the point-to-point communication model.

This new RDMA implementation of the collective operations give good performance. This new RDMA implementation is found to give a benefit of up to 30% for a 16-node barrier operation. We get a benefit of 14.4% for the RDMA Broadcast operation of data size 4608 bytes for a 16 node cluster. With RDMA AllReduce, we get a benefit of 38.13% for 16 nodes and 4 byte message and a performance benefit of 9.32% for larger messages of 4K.

I dedicate this work to my mother Uma Gupta and my father Kailash Gupta

ACKNOWLEDGMENTS

I am grateful to my advisor Dr. D. K. Panda for introducing me to the field of High Performance Computing. I thank him for his patience, understanding and invaluable guidance during the last 2 years.

I am grateful to Dr. Pete Wycoff and Prof. P. Sadayappan for agreeing to serve on my Master's examination committee.

Special thanks to Darius Buntinas, Jiesheng Wu and Igor Grobaman for spending a lot of time and effort in discussing various technical and non techincal issues and questions.

I am indebted to the OSU CIS department for awarding me teaching assistantship and Prof. Panda for supporting me as a research associate.

Last but not the least, I would like to thank my friends Pavan, Sandy, Chubs, Bala , Vijay and Suchi who made my stay at OSU an unforgettable experience.

My heartfelt thanks to all of you.

VITA

October 16, 1977	Born - Aurangabad, INDIA
1999	B.E. Computer Science
Mar 2000 - Aug 2000	Software Engineer, Mahindra British Telecom, Mumbai
Sep 2000 - Aug 2001	Graduate Teaching Associate, Ohio State University.
Sep 2001 - Present	Graduate Research Associate, Ohio State University.

PUBLICATIONS

Research Publications

Rinku Gupta, Vinod Tipparaju, Jarek Nieplocha, Dhabaleswar Panda "Efficient Barrier using Remote Memory Operations on VIA-based Clusters", 2002. *Proc. of IEEE Cluster Computing 2002* , September 2002.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

High Performance Computing : Prof. D.K. Panda

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

High Speed interconnection networks and exponentially increasing microprocessor performance have made Networks of Workstations (NOWs) an increasingly appealing alternative to mainstream supercomputing for a variety of computational needs of computation intensive applications. Commonly known as Cluster Computing systems, these collections of commodity based components offer a high performance to price ratio to the end user, attributing to it's immense success.

The communication stack in such a system is made up of several layers. Figure 1.1 describes the communication stack. At the lowest layer, lie the hardware interconnects

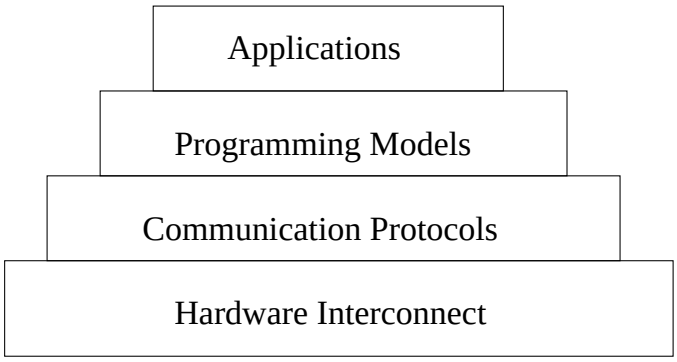


Figure 1.1: The Communication stack

and the communication protocols. On top of these communication protocols are programming models which provide ease of programming and portability to users. At the highest level are the application programs.

This chapter starts with a brief overview of the various communication protocols and programming models supported. The later part of the chapter talks about the different collective operations and follows it up with several optimization techniques possible for these operations on modern networks and protocols.

1.1 Current Technology

Together with the increasing success of Cluster Computing systems, there has been a parallel progress in the field of high speed interconnects, leading to the development of interconnects like GigaNet cLAN [14], Myrinet[3], Gigabit Ethernet [9] and Quadrics [24]. With the advent of high speed interconnects, the bottleneck in the communication has shifted from the interconnection network to the messaging software overheads at the sending and the receiving side.

Traditional kernel based protocols such as TCP/IP and UDP/IP were not able to utilize the performance offered by these high speed interconnects due to the multiple copies and kernel context switches in the critical message passing path. Thus the communication latency was high. This led the researchers to come up with alternatives to increase the communication performance delivered by the NOWs in the form of low-latency and high bandwidth user-level protocols such as FM[23] and GM [7] for Myrinet, U-Net [29] for ATM and Fast Ethernet [17] and others.

An attempt to shift the protocol processing to the user level resulted in the development of user-level protocols. In the user-level protocols, the user is given direct

protected access to the network without the intervention of the operating system. Hence, critical operations such as message sending and receiving bypass the operating system and reach the network directly. Bypassing the operating system eliminates multiple copies and kernel context switches, thereby decreasing the message latency. Protocols like EMP [25] [26] promise further increase in performance by offloading the critical protocol processing to intelligent NICs. In the past few years, several industries have taken the initiative to standardize some of these protocols, the result of which are the Virtual Interface Architecture (VIA) [13] and the InfiniBand Architecture (IBA) [1].

Due to the diversity in the API and functionality of these user-level protocols, researchers have been developing high performance programming models which offer ease and portability to the end users without compensating on the achievable performance to a great extent.

Based of the type of architecture involved, various programming models can be supported. The programming models can be chosen on the basis of whether the high performance system is a single node with many SMPs or a cluster of nodes with completely shared or distributed memories. The three most widely used programming models are:

1.1.1 Shared Memory Programming Model

The Shared Memory Programming model has a completely shared address space. Sending and receiving of data is done by local reading and writing to shared space. Such a model has to deal with various data coherency and consistency issues. Access

to data takes place through a shared communication channel, which may limit the scalability of the system.

1.1.2 Message Passing Programming Model

Communication in Message Passing Programming model is done solely on the basis of sending and receiving messages. The address space is generally completely distributed and the control of parallelism lies in the hands of the developer. Message Passing is widely portable and this programming model is standardized by the vendor community in the form of the Message Passing Interface (MPI) standard.

1.1.3 Distributed Shared Memory Programming model

The Distributed Shared Memory (DSM) model was introduced to exploit the advantages of having a shared memory system on a distributed memory system. It is able of deal with both distributed and shared memory architectures. This model provides an illusion of shared memory, generally through an interface, with the actual memory being distributed on different machines. Access to shared memory is done by mapping the shared locations to the physical locations and coherency is achieved by various synchronization primitives. A DSM can be with coherency or without coherency, depending upon the implementation. The DSM model can be implemented in various ways. Some implementations use the get and put model, wherein a process can directly read and write the data to a remote node, after specifying the remote data address at the local node. Global arrays [22], a portable DSM library uses such a mechanism.

1.2 Collective Communication Operations

High Performance Parallel Programs running on a cluster of nodes require a lot of communication between them in addition to the computation being carried out.

Point-to-Point operations involve the participation of two nodes, where one node is the sender and the other node is the receiver. Frequently, in parallel computing, there arises a need to communicate with a group of nodes at the same time. Such a communication involving a group of processes at the same time, is termed as a Collective communication operation.

The common examples of collective operations are (i) Barrier, which is a synchronization operation between all nodes, (ii) Broadcast, wherein the same data can be sent to all nodes, (iii) Reduce, where data from different nodes can be collected to perform a particular operation.

Standard collective operation algorithms simplify the higher level application programming for clusters while implementing efficient communication methods. They promote the portability of applications across different architectures and reflect conceptual grouping of processes. Collective communication operations are extensively used in scientific applications where interleaving of stages of local computations with stages of global communication is possible.

Most of the collective operations which follow a standard pattern have been supported by Message Passing Interface (MPI), the Message Passing Standard [18].

1.3 Problem Statement

Past works in the collective communication area have primarily focused on development of optimized and scalable algorithms on top of point-to-point operations

[28]. These point-to-point operations are typically supported by the send and receive model of communication. The send and receive point-to-point communication requires explicit intervention at both the sender and the receiver side. Modern user-level protocols such as VIA and IBA offer a variety of models for data transfer. Together with the traditional send and receive model, they also support the Remote Direct Memory Access (RDMA) model giving the end user an option to chose between them. The concept of Remote DMA is used for direct transfer of data between user spaces without any intervention from the receiving host. In other words, the RDMA operation is transparent to the receiver. This concept is very similar to the get and put model provided by the ARMCI [21].

Remote memory capability through RDMA operations allows the programmer to define a set of buffers across the nodes of a cluster which can be used as a logical shared address space to exchange data efficiently. This raises the following open question:

Can remote memory operations be used to support efficient communication steps for a collective communication operation?

As a part of this research, we explore the novel idea of supporting the collective communication operations using the Remote DMA capability offered by VIA. We put forward the issues and the challenges related to this and provide a proof of concept asserting the effectiveness of such a collective communication library.

1.4 Our Approach

In this thesis, we discuss the various design issues and alternatives that may be the basis of a RDMA supported collective communication library. We discuss issues related to buffer management, data identification and validity at the receiver end, address exchange mechanisms etc. As a proof of concept, we have designed, analyzed and implemented three collective communication operations. We have designed a RDMA Barrier, which is a synchronization operation. Since it involves no data transfer, the concepts and the design alternatives are simpler and provide a good understanding of the issues in RDMA Collective operations. We also discuss the Broadcast collective operation which is a data distribution operation which will serve as an example for the understanding of data intensive collective operations. In the end, we implement the All Reduce collective operation, a global reduction mechanism. We discuss the design alternatives for all these three operations. For the RDMA AllReduce, we introduce special algorithms called as the Degree-k tree based AllReduce algorithms, which when combined with the RDMA mechanism give improved performance as compared to the message passing algorithms based on the point-to-point communication model. We also provide an analytical model for deciding the optimal RDMA All Reduce algorithm for a given configuration and data size.

This new RDMA implementation is found to give a benefit of up to 30% for a 16-node barrier operation. We get a benefit of 14.4% for the RDMA Broadcast operation of data size 4608 for a 16 node cluster. With RDMA AllReduce, we get a benefit of 38.13% for 16 nodes and 4 byte message and a performance benefit of 9.32% for

larger messages of 4K. We use the MVICH-1.0 [11], the VIA implementation of the MPI Standard as the basis of comparison for our results.

The remaining part of the thesis is organized as follows. Chapter 2 provides an overview of the Virtual Interface Architecture (VIA) and the Message Passing Interface (MPI). In chapter 3, we discuss the motivation and the basic concepts behind the work. Chapter 4, 5 and 6 deal with the Barrier, Broadcast and the All-Reduce operations, respectively. We conclude the thesis and discuss possible future work in Chapter 7.

CHAPTER 2

RELATED BACKGROUND

In this chapter, a brief overview of Virtual Interface Architecture [13], a widely used user level protocol is provided. This is followed by an overview of Message Passing Interface and MVICH, a popular implementation of the MPI.

2.1 Virtual Interface Architecture

The Virtual Interface Architecture (VIA) has been standardized as a low latency and high bandwidth user-level protocol for System Area Networks(SANs). A System Area Network interconnects various nodes of a distributed computer system.

The VIA architecture mainly aims at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical path of the message. This is achieved by providing every consumer process a protected and directly accessible interface to the network named as a Virtual Interface(VI). Figure 2.1 illustrates the Virtual Interface Architecture model.

Each VI is a communication endpoint. Two VIs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple VIs. Each VI has a Work queue consisting of send

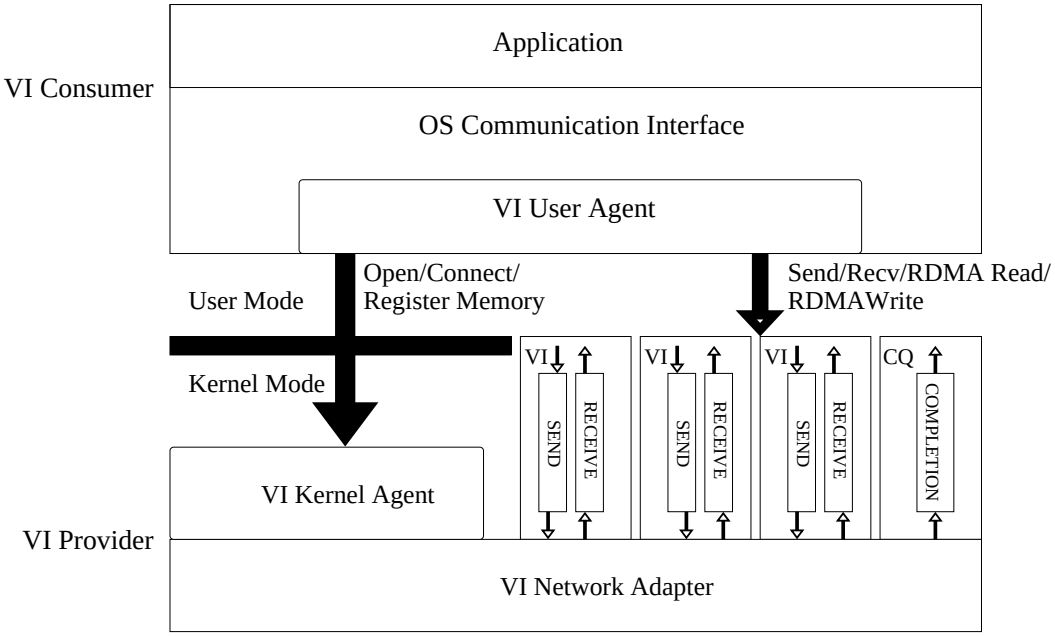


Figure 2.1: VI Architectural Model

and a receive Queue. A doorbell is also associated with each VI. Applications post requests to these queues in the form of VIA descriptors. The posting of the request is followed by ringing of the doorbell associated with the VI to inform the VI provider about the new request. Each VI can be associated with a completion queue (CQ). A completion queue can be associated with many VIs. Notification of the completed request on a VI can optionally be directed to the completion queue associated with it. Hence, an application can poll a single CQ instead of multiple work queues to check for completion of a request.

A VIA descriptor is a data structure which contains all the information needed by the VIA provider to process the request. Each VIA descriptor contains a Control Segment (CS), zero or more Data Segments (DS) and possibly an Address Segment (AS). When a request is completed, the Status field on the CS is marked complete.

Applications can check the completion of their requests by verifying this field. On completion, these descriptors can be removed from the queues and reused for further requests.

The Data segment of the descriptor contains a user buffer virtual address. The descriptor gives necessary information including the data buffer address and length. VIA requires that the memory buffers used in the data transfer be registered. This allows the VI provider to pin down the virtual memory pages in physical memory and avoid their swapping, thus allowing the network interface to directly access them without the intervention of the operating system. For each contiguous region of memory registered, the application (VI consumer) gets an opaque handle. The registered memory can be referenced by the virtual address and the associated memory handle.

The VIA specifies two types of data transfer facilities: the traditional send and receive messaging model and the Remote Direct Memory Access (RDMA) model.

In the send and receive model, each send descriptor on the local node has to be matched with a receive descriptor on the remote node. Thus there is a one-to-one correspondence between every send and receive operation. Failure to post a receive descriptor on the remote node results in the message being dropped and if the connection is a reliable connection, it might even result in the breaking of the connection.

In the RDMA model, the initiator specifies both the virtual address of the local user buffer and that of the remote user buffer. In this model, a descriptor does not have to be posted on the receiver side corresponding to every message. The exception to this case is when the RDMA Write is used in conjunction with immediate data, a receive descriptor is consumed at the receiver end.

The VIA specification does not provide different primitives for Send and RDMA. It is the VIA descriptor that distinguishes between the Send and RDMA. The Send descriptor contains the CS and DS. In case of RDMA, the VI Send descriptor also contains the AS. In the AS, the user specifies the address of the buffer at the destination node and the memory handle associated with that registered destination buffer address.

There are two types of RDMA operations: RDMA Write and RDMA Read. In the RDMA Write operation, the initiator specifies both the virtual address of the locally registered source user buffer and that of the remote destination user buffer. In the RDMA Read operation, the initiator specifies the source of the data transfer at the remote and the destination of the data transfer within a locally registered contiguous memory location. In both cases, the initiator should know the remote address and should have the memory handle for that address beforehand. Also, VIA does not support scatter of data, hence the destination buffer in the case of RDMA Write and RDMA Read has to be contiguously registered buffer. The RDMA Write is a required feature of the VIA specification whereas the RDMA Read operation is optional. Hence, the work done in this thesis exploits only the RDMA Write feature of the VIA.

Since the introduction of VIA, many software and hardware implementations of VIA have become available. The Berkeley VIA [6], Firm VIA [2], M-VIA [15], Server Net VIA [27], GigaNet VIA [14] are among these implementations. In this thesis, we use GigaNet VIA, a hardware implementation of VIA for experimental evaluation.

2.2 Message Passing Interface

Message Passing Interface [18] is the most popular and widely used standard library specification for developing message passing high performance applications. It provides portability and ease of use for the parallel programs written using the distributed memory programming model.

The MPI standard specifies a rich set of functions for point-to-point communication and collective communication, all scoped to a user specified group of processes.

MPI provides abstractions for processes at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient and efficient way.

A key concept in MPI is that of a communicator, which provides a safe message-passing context for the multiple layers of software within an application that may need to perform message passing. For example, messages from a support library will not interfere with the other messages in the application, provided the support library uses a separate communicator. Communicators, which house group and communication context (scope) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

Within a communicator, point-to-point and collective operations are also independent. An application can post several non blocking receive operations and then call a barrier collective operation. Messages used to complete the barrier operation will

be processed independently from the posted receive operations. Most implementations of MPI simply use an additional *hidden* collective communicator to distinguish between peer communication and collective communication.

MPI version 1.0 and 1.1 [19] define all point-to-point operations as two sided operation where every send has a corresponding receive. However the future MPI version 2.0 [20] promises one-sided operations like *Remote Memory Read* and *Remote Memory Write* which shall allow the sender to specify the source and destination buffers.

The MPI Standard as mentioned provides a variety of collective communication operations. All collective operations are built upon the point-to-point communication operations. Since point-to-point communication operations are two sided, hence currently all collective operations are built on the the send and receive Message passing primitives.

2.2.1 MVICH - A MPI Implementation

MPICH [10], which combines portability and efficient code sharing with high performance is one of the most popular implementation of the Message Passing Interface 1.0 standard.

The MPICH implementation is a layered implementation as shown in 2.2. One of the lower layers is called as the *Abstract device interface (ADI)* contains the device or underlying protocol dependent code. All the MPI functions are implemented in terms of the macros and functions that make up the *ADI* and are hence portable. MPICH, thus contains many implementations of the *ADI*. The *Channel interface* is the lowest level portable implementation of the *ADI*. MPICH can be ported to any

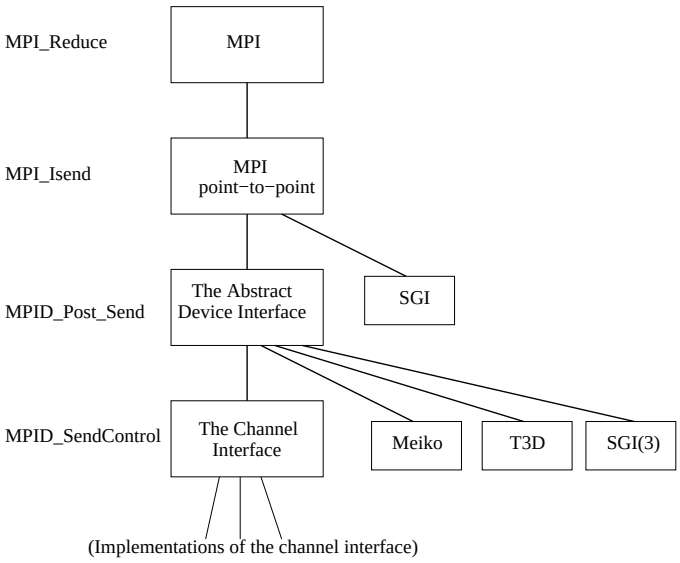


Figure 2.2: Upper Layers of MPICH [10]

new platform simply by implementing the device dependent functions at the *channel interface* level.

MVICH[11], is one such variation of MPICH where the *ADI* is modified to provide support on VIA platforms.

The layered structure of MPICH shows that the MPI collective operations are built on top of the MPI point-to-point operations. Hence a collective communication operation like *MPI_Broadcast()* will call other MPI point-to-point primitives like *MPI_Send()* and *MPI_Recv()* to send and receive messages. *MPI_Send()* will internally call the VIA specific primitives like *VipPostSend()* and *VipSendDone()* or *VipSendWait()* to send the data onto the network. The VIA primitives will be available only in and below the *ADI* layer of the MPICH hierarchy.

In this thesis, we modify the mvich-1.0 implementation of MPICH over VIA to support collective communication with RDMA.

2.3 Summary

In this chapter, we provided a brief overview of the Virtual Interface Architecture, the Message Passing Interface and its implementation. Chapter 3 will provide the basic concepts and the motivation behind the RDMA based collective communication library.

CHAPTER 3

RDMA BASED COLLECTIVE COMMUNICATION LIBRARY - CONCEPTS AND DESIGN ISSUES

VIA and the emerging InfiniBand architecture support remote DMA operations, which allow the data to be moved between the user space of the communicating nodes with low overhead. This concept can be used to create and provide a logical shared memory address space across the nodes. Many efficient collective communication algorithms have been developed that are based on the message passing paradigm. But the idea of providing a logical shared memory address space using the concept of RDMA on a distributed collection of nodes which has no shared memory has not been explored in the past.

This chapter talks about the basic motivation for this work. It also discusses the design issues involved in trying to provide an RDMA based Collective communication library.

3.1 Basic Concept

In a shared memory system, collective algorithms are simple and easy to implement. For example, let us consider the Barrier collective operation. A *Barrier* is a

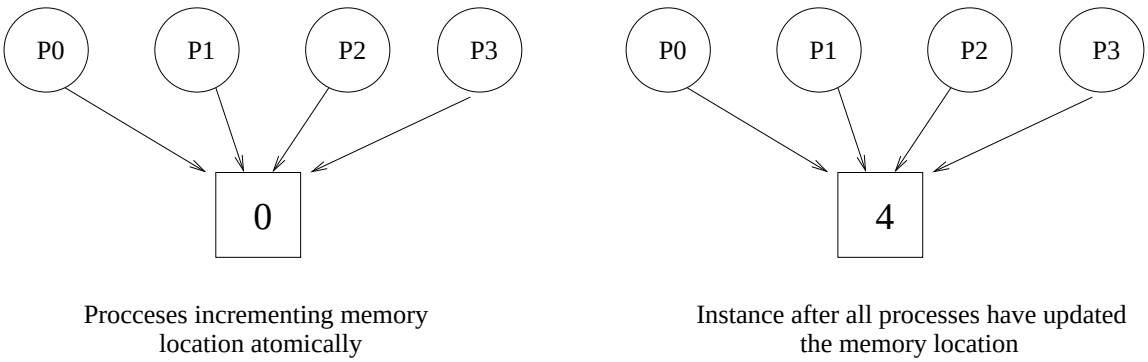


Figure 3.1: Illustration of a simple barrier scheme using one shared memory location

synchronization operation. It essentially establishes a logical point, which all nodes must attain before they can continue with their computation or communication.

In a shared memory system, a barrier operation can be done very easily. A specific memory location can be reserved for the barrier operation and initialized to '0'. When processes reach the barrier, they simply increment the value at the location (in an atomic manner) and then wait for the value to be updated by all the other processes. This concept is illustrated in Figure 3.1 with four processes (P0, P1, P2 and P3) and a single memory location initialized to '0'. Since all the processes know the number of other processes involved in the barrier (in this case a total of 4 processes are involved), when the value at the memory location reaches the value '4', each process knows that the *Barrier* has been attained and it can then proceed.

Another approach is to have a section of memory (with multiple locations) be reserved for the barrier and initialized to '0'. Every process writes a '1' to a specified location in this memory region when it reaches the barrier. Next, the process reads from other memory locations to see if other nodes have reached the barrier. This concept is illustrated in Figure 3.2 with four processes (P0, P1, P2 and P3) and

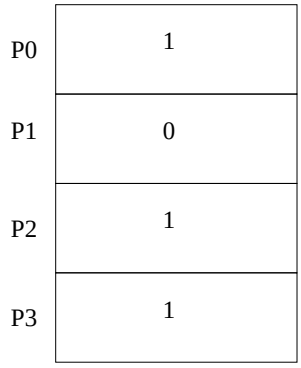


Figure 3.2: Illustration of a simple barrier scheme using multiple shared memory locations

four memory locations. The figure shows the memory location corresponding to each process. In this figure, P0, P2, and P3 have already set the byte in their respective locations when they encounter the barrier and are waiting for P1 to set the value in its location. As soon as P1 sets the value in its location, all processes return from the barrier.

Efficient execution of the above shared memory-based barriers require several issues related to cache coherency to be addressed. If the shared memory is cache coherent, the barrier implementation turns out to be considerably simpler and faster. The processes obtain the data by a simple local read operation without additional complexities. However to reduce false sharing, the memory locations associated with the processes need to be mapped to different cache lines to eliminate false sharing [8].

In a cluster with distributed memory organization, when an operation like barrier takes place, the nodes typically send and receive explicit messages. Barrier algorithms (pair-wise exchange with recursive doubling or gather-followed-by-broadcast [16]) with multiple phases (steps) are used to implement the barrier. Each of the

communication step typically uses a send and receive primitive to communicate. Receiving a message from a node is typically an expensive operation. For example, an MPI over VIA implementation has to take care of unexpected receive messages. When messages come in, the relevant descriptor has to be searched for. If there is no descriptor posted, data is sent to an intermediate buffer. When the actual descriptor gets posted, the data has to be copied from the temporary buffer to the user buffer.

In addition, the layering structure of the libraries like MVICH adds considerable overhead on the message latency, making each of the communication step slower and the entire barrier operation slower.

The method of RDMA communication offers a new mechanism for transferring data, by directly writing into the memory of a remote processor/node. Consider a set of buffers being allocated at each remote processor/node and their addresses being exchanged at the start of the program. The collection of these buffers (together with their addresses) provide a logical shared memory region (without coherency) for all processors. Now, the processors can exploit the advantages associated with shared memory-based algorithms to implement the barrier.

We extend the same explanation as that of the barrier to all the other collective operations. We thus exploit the logical shared memory capability provided by RDMA operations to support efficient collective operations. In this thesis, we limit the proof of concept to the (i) *Barrier*, (ii)*Broadcast*, (iii)*All Reduce* collective operations.

3.2 Design Issues for an RDMA Collective Communication Library

The idea behind using RDMA for collective communication is to utilize the concept of shared memory. The RDMA mechanism and memory registration constraints open

up several major issues for designing a RDMA based collective communication library. One issue is how and when to register the buffers and communicate the addresses of the buffers to all the nodes. Another issue is how to identify valid data at the receiver and how do we safely reuse the buffers.

In this section, we discuss these design issues and present some solutions. In the subsequent chapters we will discuss the design choices for the particular collective communication operation and its implementation.

3.2.1 Registration of buffers and Address Exchange

It is a requirement in VIA that data to be sent and received should use registered buffers. A flexible buffer management scheme is required for this purpose in the context of collective operations. In our scheme, we register the buffers statically before the operation or dynamically during the operation.

Static Buffer Registration

We statically register a contiguous region in memory for each communicator for various types of collective operations. This region is generally registered when the communicator is being created. This contiguous region is split into fixed size buffers. Since the memory allocated is contiguous, only the starting address of the memory (the address of the first buffer) needs to be communicated to all the nodes. The length of the buffer space is the same for all the nodes in the same communicator for the same operation. There will be certain constraints on the order of using these buffers, which shall be discussed in the incoming sections. In this scheme, the address is communicated only once and the buffers are reused as and when needed.

Dynamic Buffer Registration

In the dynamic registration scheme, we allow the use of non-contiguous buffers. This will make it mandatory to communicate the addresses of all the buffers to all the nodes for every collective operation instance. Dynamic registration need not be done at the start of the program or when the communicator is created. It is done as a part of the operation itself after the requisite user buffers have been declared. However, in this approach the buffer addresses need to be communicated whenever the buffers are created dynamically. Hence, if we register the buffer in the collective operation, we will have additional overhead of address communication with the destination set in the collective operation before sending the actual data to the destination set.

3.2.2 Data Validity at the Receiver end

RDMA write is receiver transparent. It does not require that the receiver post a descriptor or perform any action in anticipation of the incoming data. The receiver process receives no indication that any new data has been written. When the destination needs the data it goes to the memory location and fetches the data from there by performing a local read operation. Thus, we need a mechanism for indicating to the receiver that the data in the memory is valid data.

There are various ways in which this can be done. One method is to let the receiver NIC interrupt the receiver once it receives an RDMA message. But this is a very expensive operation and thus detrimental to high performance.

Another approach is to use the immediate field in the RDMA descriptor and set the field when the last RDMA write operation has taken place. However, this requires consumption of a descriptor at the receive end. This also requires that the receiver

be aware of the data coming and post a receive descriptor in advance. This approach disturbs the illusion of shared memory and is not feasible.

Another approach is to write a special value, known to the receiver at a special location in the receiver's memory for each buffer in the pool. The value will be written after the sender has finished writing to the destination memory. This special value will indicate the data validity at the destination end. RDMA write supports the reading of data from non-contiguous locations but does not support scattering of data in a single RDMA write operation. Thus in a single operation, writing the data to the destination buffer and writing the special value to a separate buffer location at the destination end is not possible. To perform the transferring of the data, we will need to perform two RDMA writes, the first for sending the data to destination buffer and the second for updating the special byte which indicates the validity of data at the destination end. But performing two RDMA writes is very expensive. Also, the order in which the destination NIC will write the data in the destination memory is not fixed. The destination NIC, on receiving both RDMA writes may decide to write the special byte first, thus defeating the entire purpose of using special byte for indicating data validity at the receiver end.

Another approach would be to attach the special byte to the end of the data. Thus the sender sends the data and an extra byte with a special value to the destination. The destination knows when and how much data is arriving and thus it checks the byte at the end of data and determines the validity of data.

3.2.3 Safely reusing the buffers

In the static buffer allocation scheme, buffers are allocated during the initialization time. No new buffers are allocated in the course of the program. These finite number of buffers need to be reused. In the RDMA scheme, the sender has no indication as to when the receiver has read the data written. Before the buffer can be reused, the sender needs some confirmation from the receiver that the buffer can be reused. The buffers are contiguous in nature and are used contiguously. Thus for the same communicator and the type of collective operation, the receiver knows when exactly the buffer is going to be reused by a sender. The receiver can then explicitly RDMA write a notification to the sender and the sender can proceed with the writing of data after it has received the notification.

In the dynamic buffer allocation scheme, the problem of reusing buffer does not arise as the buffers are allocated separately for each collective communication operation.

In this chapter, we discussed the motivation and the design choices for a RDMA collective communication library. The next chapters explore the RDMA implementations of the Barrier, Broadcast and AllReduce operations.

3.3 Summary

In this chapter we gave the basic motivation for this work. We also discussed the various issues and alternatives related to buffer allocation and management, schemes for data indication at the receiver end and address exchange mechanisms. Chapter 4 introduces the RDMA Barrier, a synchronization collective communication operation.

We will discuss the algorithm used, the design challenges faced and the performance results that were obtained.

CHAPTER 4

THE RDMA BARRIER

In this chapter, we focus on the *Barrier*, one of the most frequently used collective communication operations. Barrier, as stated earlier is a synchronization operation. Barrier enables a process to stall till all the other processes reach the same point in the program, thereby enabling synchronization.

All popular MPI implementations including MVICH, an MPI implementation on VIA, currently use MPI point-to-point communication primitives like *MPI_Send()* and *MPI_Recv()* as the underlying method of implementing the Barrier operation.

In this chapter, we start with the RDMA Barrier algorithm, followed by the design choices and implementation details. We end the chapter by focusing on some of the results obtained.

4.1 The RDMA Barrier Algorithm

The algorithm we use is pairwise exchange with recursive doubling [4][5]. This algorithm was chosen for its simplicity and efficiency. This algorithm is also currently used in the mvich-1.0 distribution with the send and receive communication model. It also helps us to give a fair comparison between the efficiency of the Barrier using send and receive primitives and our new implementation of Barrier with RDMA.

When the RDMA barrier is called, the pairwise exchange method follows. During each barrier invocation by a process involving the same communicator, each process keeps a static count of the barrier number it is participating in that communicator. The Pairwise Exchange Algorithm is a recursive algorithm. The nodes pair up and each node does a RDMA write to the other process's buffer using the destination buffer address and the memory handle. The sender writes the barrier number which it is involved in for that communicator. Since the receiver is also in the same barrier for that communicator, it also knows the barrier number and it can read the barrier number from its local location. Thus, the nodes in the pair do a RDMA write to each others memory and read the data that the other node has written from its own local memory. The nodes form a group. In the next step two groups pair up and a node from one group does a RDMA write and checking for written data with one node from the other group. These groups are then merged together. This process of pairing up, writing data to each others buffers and then merging is repeated until only one group is left. The barrier is then finished.

Overall, in this approach each node performs $log_2 N$ RDMA writes, where N is the number of nodes in power of two. Figure 4.1 demonstrates the steps in an 8 node barrier.

For non-power of two nodes, we divide the set of nodes N into two sets S and S' where S is the maximum power of two less than N and S' is the set of nodes in N but not in S. Initially, every node in S' does a RDMA write to another node in S. Then the nodes in S perform a pairwise exchange barrier. Once the nodes in S reach a barrier, they RDMA write to the corresponding nodes in S'. This concludes the

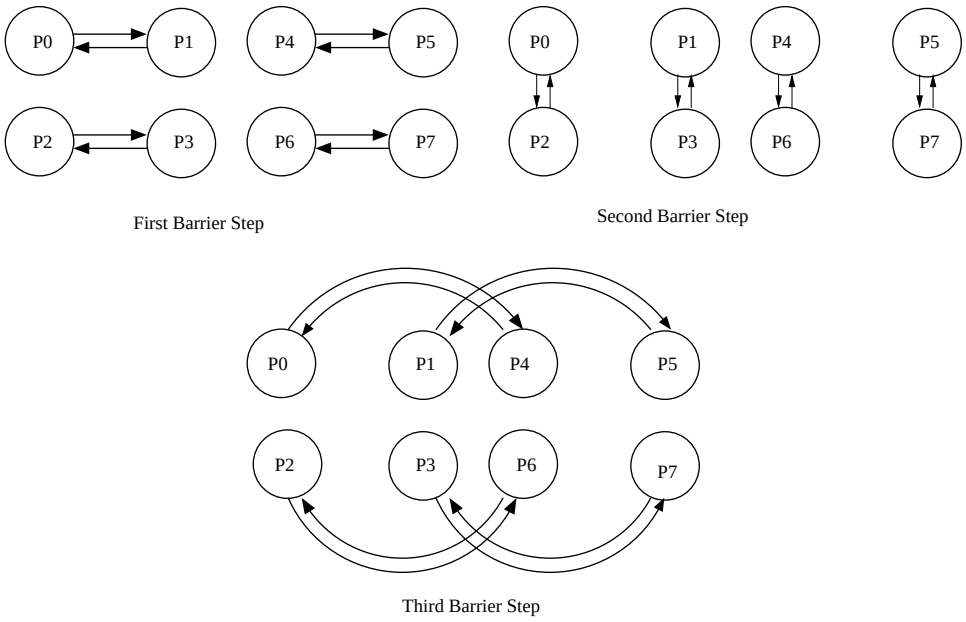


Figure 4.1: Steps in an 8 node barrier

barrier for the N nodes. The number of steps it takes to reach the barrier is $log_2 N + 2$ steps.

4.2 Design Solutions and Implementation Details

The implementation of the RDMA Barrier was based on the following design solutions. We indicate the nodes that take part in the barrier by the term 'barrier group'.

4.2.1 Buffer Registration

We register a buffer with every process. Every byte in the buffer is reserved for a different node in the barrier group. Nodes are differentiated on the basis of their *id* in their barrier group. Each node has a different *id* in the barrier group.

The first byte is reserved for the node with *id 0*, the second byte for the node with *id 1* and so on. Figure 4.2 shows RDMA Barrier in a 4 node cluster having processes P0, P1, P2, P3 each on a different node. Each node has reserved a four byte buffer, with one byte reserved for every node in that barrier group.

4.2.2 Buffers and Address exchange

For a node to write data in some other process's memory, the node needs to know the destination address and have the right memory handle. Thus, the addresses of the buffers need to be communicated from one node to all the other nodes. Since the buffers are contiguously allocated and have the same handle, only the starting address needs to be communicated to the other nodes. The RDMA Barrier implementation does address exchange when the barrier group is created by using explicit send and receive primitives. Communicating the address needs to be done once, only when you create the communicator.

4.2.3 Buffer Initialization

Since barrier is essentially a synchronization operation and the data passed is not relevant, we initialize all the buffer bytes reserved for barrier operation to a negative constant. It shall be shown in the next sub-section that we never indicate a barrier operation by a negative number and hence the initial value of negative constant suffices. Figure 4.2 shows the first barrier operation of a program. The buffer bytes are initialized to the negative constant -1.

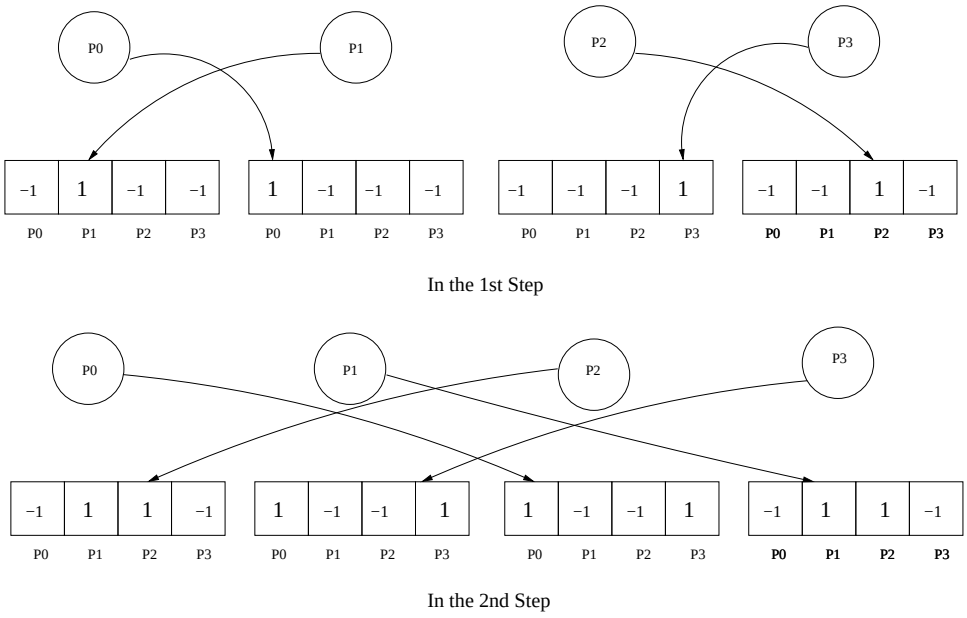


Figure 4.2: RDMA Barrier between 4 nodes

4.2.4 Data identification at the Receiver end

When the receiver needs the data, it goes to the corresponding memory location and reads the data from there. The algorithm writes the barrier number to the destination buffer. The receiver knows the barrier number that will be written. Hence it checks the location for the desired data. However the implementation does not check for the desired value, but instead it checks to see if the value written in its location by the other node is greater than or equal to the barrier number. This is to take care of consecutive barriers.

In a barrier between a pair of nodes, one of the participating nodes might be slower than the other. Consider two processes and P_A and P_B which are in the i^{th} barrier, with the barrier number as i. P_A writes the barrier number i in P_B buffer and starts polling for P_B to write the same number in its buffer. Meanwhile P_B, which

is the slower node, writes the barrier number i in P_A buffer. P_A, which was waiting for this value, reads it and immediately enters the next $i + 1^{th}$ barrier and writes the barrier number $i + 1$ to P_B barrier buffer. If P_B is a slow node, it may happen that P_A writes the new $i + 1$ value before P_B has read the i value. So, the nodes will need to poll for the value i or $i + 1$ to be written in their buffers.

The barrier number will always have an upper limit depending on the data type that is used. We wrap around the barrier number once it reaches that upper limit. Hence, if barrier number is a 4 byte datatype, the upper limit will be 127. We wrap around the barrier number to the value 1. Hence in the 127^{th} barrier, polling is done for the values 127 or 1 to be written to the barrier buffer.

Thus, as shown in Figure 4.2 if the nodes in the cluster are in the first barrier, process P0 will poll for a value of 1 or 2 to be written in its buffer by the various processes.

4.2.5 Safely reusing the buffers

In the barrier, no data is being communicated. The nodes RDMA write the barrier number to the destination location. The barrier number, a static value in the barrier algorithm is incremented by 1 for all consecutive barriers in the same communicator. Hence, the value of the special byte (which is the barrier number) written is different for consecutive barriers, thus allowing the nodes to reuse the same buffers without any extra notification from the receiver to the sender's side.

4.3 Modifications to Mvich-1.0

The MPI-1.0 standard does not provide any primitives based on the RDMA model. It provides *MPI_Send()* and *MPI_Recv()* for sending messages and receiving messages. The MPI-2.0 standard does provide primitives for one-sided communication operations like *MPI_Put()* and *MPI_Get()* but there are currently no available implementations of this relatively new standard.

Mvich-1.0 is an implementation of the MPI-1.0 standard for VIA platforms. Referring to Figure 2.2 in chapter 2, we notice that the VIA specific code is contained in the *ADI* layer. VIA as we discussed in the previous chapter does provide RDMA write support. To enable Mvich to support RDMA collective communication operations, we modify the code in the *ADI layer*. We add explicit support for RDMA write in the *ADI layer*.

The RDMA Barrier algorithm is written above the ADI level and contains only MPI primitives and is independent of the underlying communication protocol. Hence, we need a MPI primitive like *MPI_Put()*, which the Barrier function can call to perform a RDMA write. Such a primitive will call the required VIA functions in the *ADI* layer and the VIA functions will do the RDMA operation.

We avoid adding a new MPI primitive to the Mvich-1.0 implementation, because such a MPI primitive will not be a part of the current MPI-1.0 standard. Hence, to provide RDMA write support, we override the *MPI_Send()* primitive itself as and when needed. We cannot add any new constant parameters to *MPI_Send()* and hence this modification is done by setting a *global variable* before calling *MPI_Send()*. This *global variable* visible below the *ADI layer* allows the ADI to choose between Message send and RDMA write. The *global variable* is by default set to *FALSE*, which indicates

a message send to the *ADI* layer. We set this primitive to *TRUE* when we wish to perform a RDMA write.

Figure 4.3 shows the RDMA barrier pseudo code for *power of 2* cluster size. Lines 1 and 2 use MPI primitives to obtain the rank of the current process and the size of the communicator. Lines 3-6 increment *the barrier_number*, a static byte and wrap it around to 1, when it reaches the upper limit. The *barrier_number* is present in a registered memory location. Line 9 finds the destination rank for the process to communicate with. In Line 10, we call *set_rdma()*, a function which sets the global variable to *TRUE*. Line 11 calls *MPI_Send()*. At the VIA level, since the global variable is *TRUE*, the *barrier_number* will be remotely written to the correct destination buffer. Since the remote addresses are exchanged in the initialization phase, at the VIA level the correct remote buffer is chosen based on the *destination* rank and the collective operation type. Lines 12-15 show the process polling for the barrier number to be written by the destination process.

4.4 Performance Results

In this section, we discuss the results that have been obtained for RDMA Barrier and compare it with the results for the MPI Barrier for a cluster of nodes.

We evaluated our implementation on the following clusters.

Cluster 1: A cluster of 8 nodes, each with a 66MHz PCI bus, 700MHz Pentium III machines, 1GB of Main memory and Linux version 2.2.17. The machines are connected using a GigaNet 5300 switch.

```
1   MPIR_Comm_rank ( comm, &rank );
2   MPIR_Comm_size ( comm, &size )
3   if ( barrier_number == 127)
4      barrier_number = 1;
5   else
6      barrier_number = barrier_number + 1 ;
7   if ( size > 1 ) {
8      for ( d = 1; d < size; d<<=1) {
9         destinaton = (rank ^ d);
10       set_rdma(1);
11       MPI_Send(barrier_buffer, 1, MPI_CHAR, destination, 1, comm->self)
12        if ( barrier_number < 127 )
13           while (barrier_buffer[destination] < barrier_number) dummy=1;
14        else
15           while ( ( barrier_buffer[destination] != 127 ) && ( barrier_buffer[destination] != 1 ) ) dummy = 1;
16    }
17  }
```

Figure 4.3: RDMA Barrier pseudo code for *power of 2* nodes

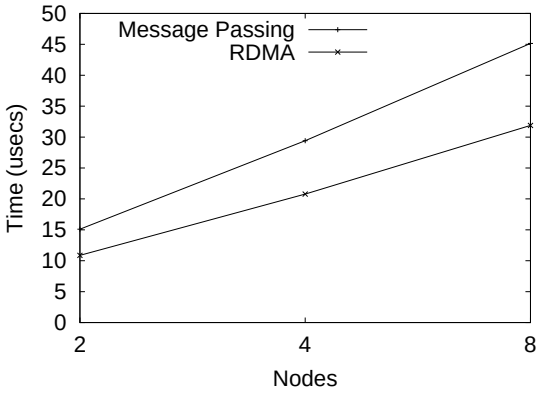
Cluster 2: A cluster of 16 nodes, each with a 33MHz PCI bus, 1000MHz Pentium III machines, 512MB of Main memory and Linux version 2.2.17. The machines are connected using a GigaNet 5300 switch.

To obtain the barrier latency, we ran 10000 iterations of *MPI_Barrier()* and took the average of the barrier latencies at each node. The MVICH version used is mvich-1.0.

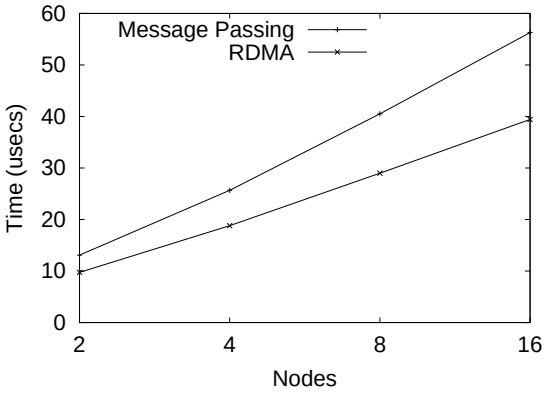
We ran the original *MPI_Barrier* without modification, the results of which are labeled under Message Passing. The RDMA Barrier is labeled under RDMA.

Figures 4.4(a) and 4.4(b) show the barrier latencies for power of 2 nodes for Cluster 1 and Cluster 2 respectively.

For Cluster 1, the RDMA Barrier for 8 nodes completes in 31.88μs as compared to the Message Passing Barrier, which completes in 45.14μs. For every message sent, we



(a) Cluster 1



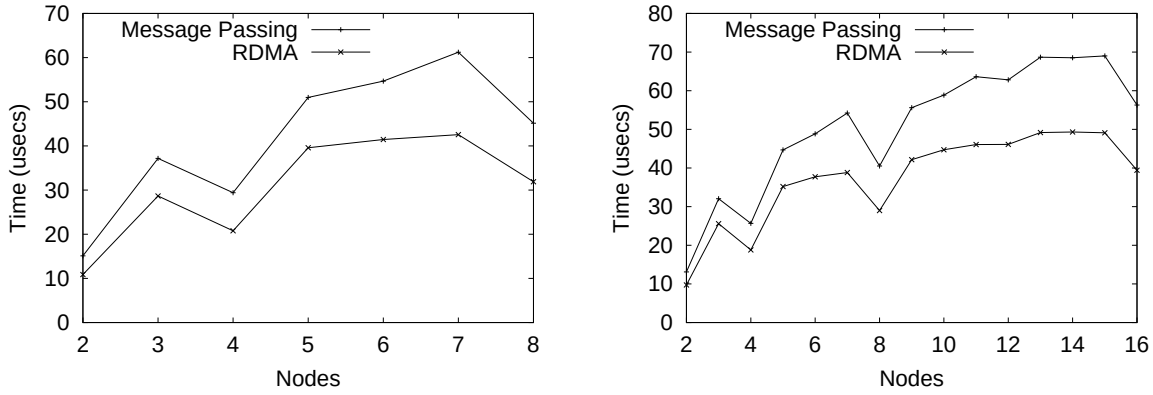
(b) Cluster 2

Figure 4.4: Barrier Latency for Power of two nodes

save 4μs in one-way latency. The RDMA Barrier outperforms the Message Passing Barrier for all power of 2 cases. This leads to a 29.37% performance improvement on the 8 nodes.

Similar results are obtained in Cluster 2 where we see that RDMA Barrier for 8 nodes completes in 29μs as compared to 40.5μs of the Message Passing Barrier. The results for 16 nodes in Cluster 2 show that RDMA Barrier completes in 39.4μs as compared to Message Passing Barrier which takes 56.3μs. This leads to 28.4% improvement on the 8 nodes and 30% for 16 nodes and is thus scalable.

Figures 4.5(a) and 4.5(b) show the barrier latency for all nodes in Cluster 1 and Cluster 2. The barrier latency for non-power of 2 nodes is greater than the power of 2 nodes because they execute larger number of steps. The timings for non power of two nodes also demonstrate an improvement in performance of RDMA Barrier as compared to Message Passing Barrier.



(a) Cluster 1

(b) Cluster 2

Figure 4.5: Barrier Latency for All nodes

4.5 Summary

In this chapter, we presented a new approach for implementing efficient barrier which exploits remote memory operations across nodes. The RDMA barrier gives a performance benefit of 30% for a 16 node cluster as compared to the message passing barrier. In the next chapters, we shall see the design and algorithm details regarding the RDMA Broadcast and the RDMA All-Reduce collective operations.

CHAPTER 5

IMPLEMENTATION OF THE RDMA BROADCAST

Most parallel applications frequently need to communicate the same piece of data to all nodes. Given a vector of data owned by one node in the group (the root), the broadcast operation duplicates the data on all the other nodes in the communicator. This is demonstrated in Figure 5.1, where given 4 processes P0, P1, P2 and P3, Process P0 sends the data to the remaining 3 processes P1, P2 and P3.

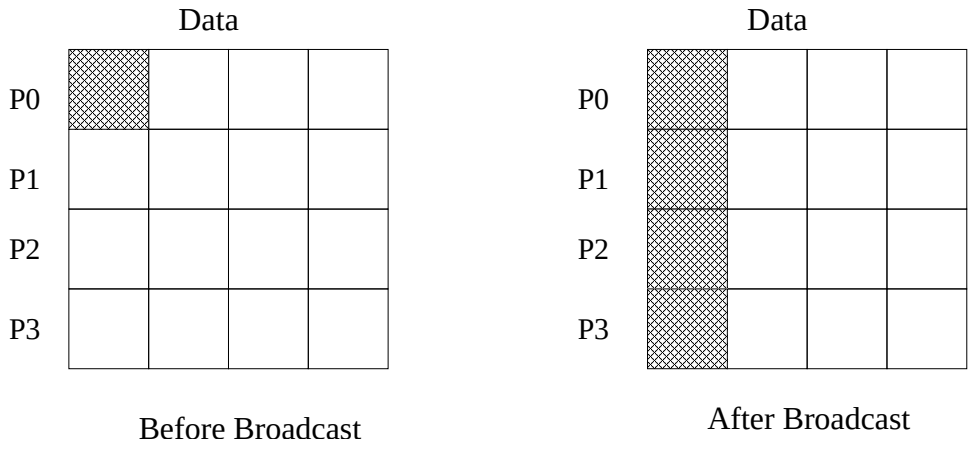


Figure 5.1: Broadcast Operation in 4 processor group

5.1 Introduction

In a distributed programming model with no shared memory, broadcast is generally done with the help of message passing. The root sends the data while the other nodes receive it. Depending upon the topology and speed of the system, the sending mechanism can be implemented in different ways. Consider *ts* as the sending time i.e. time taken by the sender to send the message to the network and let *tr* be the cumulative time taken by the receiver to receive the message. Thus, *tr* is the time to send the message, to traverse the network and reach the destination user buffer. Depending upon the values of *ts* and *tr*, we can implement several broadcast algorithms.

For a cluster where the interconnect has very high network latency, *(tr-ts)* can be high and we can implement broadcast as a one-level flat tree, where the root sequentially sends the data to all the other nodes in the cluster. This is shown in Figure 5.2

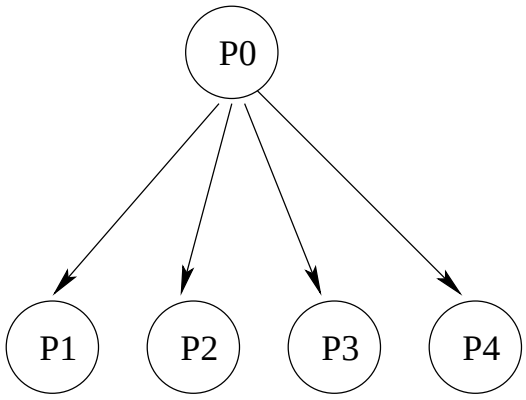


Figure 5.2: Broadcast using Flat Tree Algorithm

For a cluster with very less network latency, we can implement broadcast as a binomial operation. This binomial scheme is demonstrated in Figure 5.3

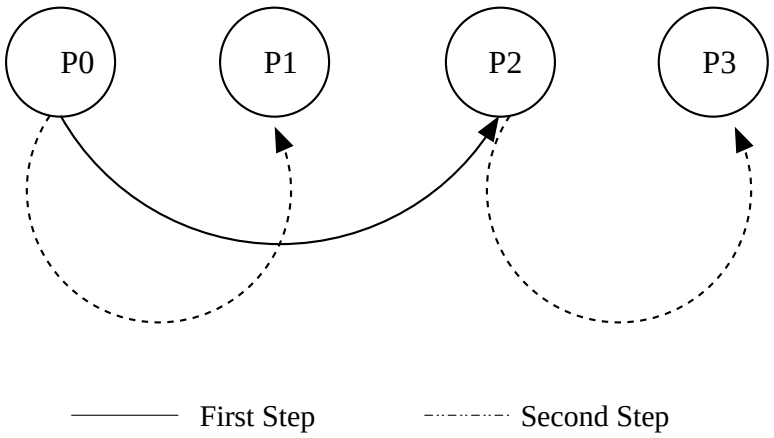


Figure 5.3: Broadcast using Binomial Algorithm

The flat tree and the binomial tree form two extreme cases for broadcast implementation. In between these two scheme, can lie the different k-nomial or k-ary implementations of the broadcast operation.

In this chapter, we start with an overview of the RDMA Broadcast algorithm. We follow it up with the design solutions and end it with a discussion of the results obtained.

5.2 Algorithm Overview and Design Solutions

Most high performance clusters are connected using fast interconnects. The current MVICH broadcast operation uses the binomial broadcast mechanism which is

implemented using *MPI_Send()* and *MPI_Recv()*. Hence, we choose the bi-nomial algorithm for RDMA Broadcast. This enables us to obtain a fair comparison between the RDMA Broadcast and the Message Passing Broadcast.

In the binomial broadcast algorithm, sending of data is divided into steps. Sending the data is done by RDMA writing into specific buffers at the receiver's end. Consider a cluster of 4 nodes P0, P1, P2, P3 where node P0 is the root with *rank id* 0. Nodes P1, P2, P3 have *ranks ids* 1, 2, 3 respectively. In Figure 5.3, in the first step, root P0 sends the the data to the node at size/2 away i.e to node 2. In the second step, root P0 sends the data to node 1. At the same time, node 2 becomes the root of a new subtree and forwards the data to node 3. The process of forming new subtrees continues till the data reaches all the nodes.

For power of 2 nodes, $logN$ steps are needed for performing the binomial broadcast, where N is the number of nodes.

For now-power of 2 nodes, the steps taken are $logN'$ where N' is the immediate higher power of 2 than N, where N is the total of nodes.

The data structures and the working of the algorithm is discussed in the following subsections. The RDMA write feature is implemented at the MPI level by modifying *MPI_Send()* to perform RDMA write at the *ADI* level.

5.2.1 Registration of buffers and Address Exchange

The RDMA Broadcast works in the following two modes. These modes are chosen to obtain the best performance for all data sizes. For smaller messages typically less than $5K$, the memory copy time is less and hence we use a static buffer management

scheme. For messages greater than $5K$ when the memory copy time is high we use the dynamic registration scheme.

Message Size $< 5K$

For messages less than $5K$, we follow the static buffer registration scheme. We allocate a contiguous section of memory and register it during the initialization time. We divide the contiguous memory into blocks of fixed size denoted by *block_size*. The address is communicated once during the address exchange phase at the initialization time. Thus, every node has the first block address as well as the memory handle for the remote broadcast blocks at each node for every communicator.

In addition to the main broadcast blocks, we also reserve a buffer called notification buffer which is used to indicate the safe reusing of the broadcast buffers. Every node registers its notify buffer. The size of the notify buffer in bytes is equal to the number of nodes in the communicator involved in the broadcast operation. The address of the notify buffers is also exchanged in the address exchange phase at the initialization time.

Figure 5.4 shows the 4 processes P0, P1, P2 and P3 and their buffers. This instance shows the broadcast buffer divided into 4 blocks of *block_size*. The blocks are numbered from 0 onwards. Hence, the first block will be called as block no. 0, the second block as block no. 1 etc. Figure 5.4 also shows a 4 byte notify buffer registered for every process.

Message Size $> 5K$

For messages greater than $5K$, we adopt the *rendezvous* approach, a dynamic scheme similar to the one implemented by MPICH/MVICH implementation of MPI.

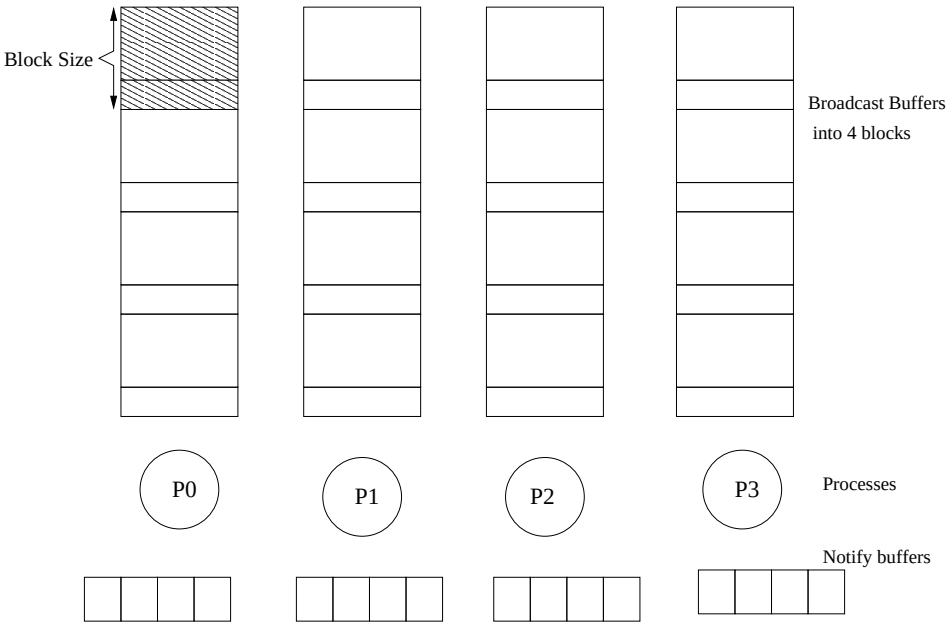


Figure 5.4: Buffer Allocation in RDMA Broadcast

In the *rendezvous* approach, the sender sends a request to the receiver asking it to register its receive user buffer and send back the address of the registered buffer. After the receiver sends back the address, the sender can directly RDMA write to the receive user buffer. The exchange of first 2 messages happens by using the *MPI_Send()* and *MPI_Recv()* primitives as shown in Figure 5.5.

This eliminates the need for having pre-registered broadcast buffers. In the static scheme, as we shall later show, the data needs to be copied from the broadcast buffers to the specified user buffers. In the dynamic scheme, we avoid the copying but we need an extra round trip for exchanging addresses.

However, for large messages, typically greater than $5K$, copying in static scheme becomes more expensive compared to the round trip overhead in *rendezvous* scheme. Hence we use the *rendezvous* scheme for higher message sizes.

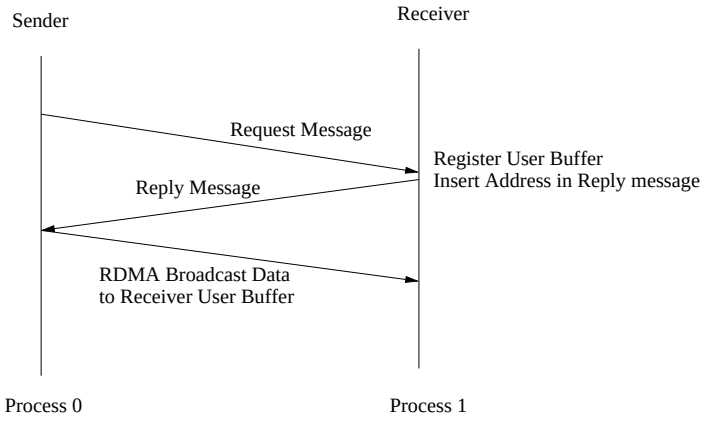


Figure 5.5: Rendezvous in RDMA Broadcast

Since the *rendezvous* scheme is currently implemented by mvich-1.0 implementation, hence we will not discuss it further. Further analysis and discussions is constricted to the static broadcast scheme only, which is used for sending messages less than $5K$ bytes.

5.2.2 Buffer Initialization

The broadcast and the notify buffers are initialized to -1 during the initialization time. In later sections, we will show that we do not indicate the data validity at the receiver end by a negative constant.

5.2.3 Data Validity at the Receiver end

Consider Figure 5.6 with 4 processes P0, P1, P2, P3, where process P0 is the root and the broadcast instance shown is between the processes P0 and P2.

For every communicator, we have a static counter called *broadcast counter* which is incremented by 1 for every broadcast operation, by every process within that communicator. Consider the first broadcast of data size *block_size/2* bytes. The *broadcast*

counter for the first broadcast is set to 1. The sender appends the *broadcast counter* byte at the end of the data to be written. The root P0 can RDMA write the data of size *block_size/2 + 1*, which includes the appended byte to block no. 0 of process P2. Since for a communicator, every node is involved in the collective operation, hence every node can keep track of the number of blocks used for that particular collective operation.

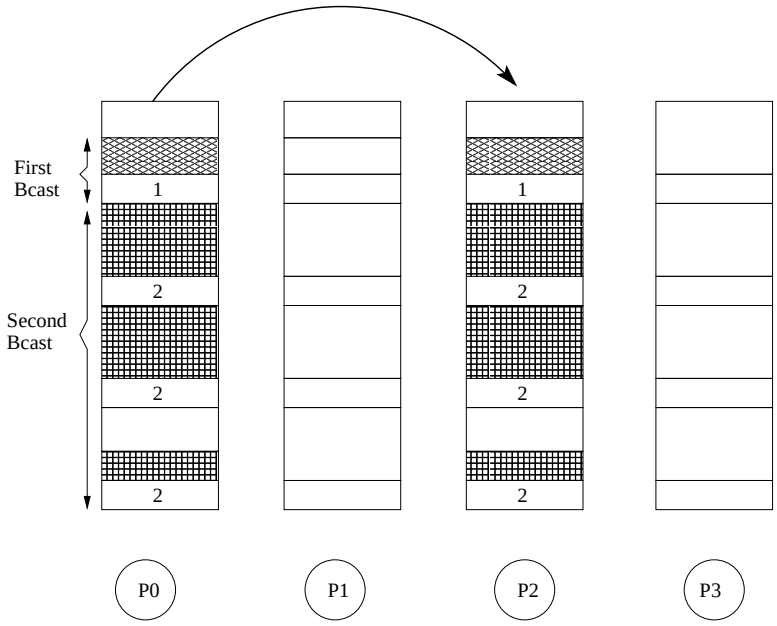


Figure 5.6: Sending Instance in Two consecutive broadcasts

The data is written bottom up, as shown in Figure 5.6, so that the sent *broadcast counter* is always written in the last byte of the block.

Since the receiver shares the communicator with the sender, hence the *broadcast counter* at the sender and receiver will have the same value. Thus, the receiver can

poll for the *broadcast counter* on the last byte of the received block and check for the validity of the data.

If the data to be sent is greater than the *block_size*, the data is split up into blocks of size *block_size - 1*, the broadcast_byte is appended to each block and the data is then written to the remote node. Figure 5.6 also shows the second broadcast of *2 * block_size* bytes. Root P0 writes the first and second blocks of *block_size - 1*, with *broadcast counter* of 2 appended, to block no. 1 and block no. 2 (of Process P2)respectively. The additional 2 bytes are written in block no. 3, again with the *broadcast counter* of 2 attached for data validity. Writing large messages by breaking them into blocks at the lowest level also enables pipelining of messages.

Once the data is read by the receiver, the receiver if needed can forward the data to the other nodes directly from the received buffer.

In our scheme, a node forwards the message only after it receives all the blocks of that message. A message is split into blocks below the ADI level.

Another approach would be to forward each block as we receive it. However, since the algorithm is written above the ADI level, hence to forward each block as we receive it would require us to call *MPI_Send()* (modified for RDMA write) multiple times, the overhead of which will increase the latency for the operation. Another approach would be to implement the broadcast algorithm at the ADI level, but which will result in a loss in portability.

After sending the data to the other nodes, the receiver will need to reset the last polling byte of the received blocks to -1, so that they can be safely reused at a later stage.

5.3 Buffer reusing

The memory allocated in the static scheme is constant and limited. Hence, when all the blocks have been used, we need to reuse the blocks safely. The sender cannot directly write to the used block because the receiver does not return any indication of whether the data written in the previous broadcast in the same block has been read or not. Hence, explicit notification is needed from the receiver.

When the receiver realizes that data to be written is in the first block of the broadcast buffer, it writes a special value in the senders notification buffer as shown in Figure 5.7. Before writing, all the last bytes of all blocks have to be set to -1. This is because we might encounter a situation where a *broadcast counter* with value 20 was written in an earlier broadcast. As the static *broadcast count* is incremented and wrapped around when its limit is reached, hence during a later broadcast, we may end up writing the same broadcast value, in which case the receiver might end up reading the old broadcast data.

Writing data from bottom-up requires us to reset only the last byte of each block. If data was written starting from the top of the block, the polling byte would have been at an arbitrary location depending on the size of data being sent and hence all the bytes of all blocks would have to be reinitialized which may be an expensive operation.

5.4 Performance Results

In this section, we discuss the results that have been obtained for RDMA Broadcast and compare it with the results for the MPI Broadcast for a cluster of nodes.

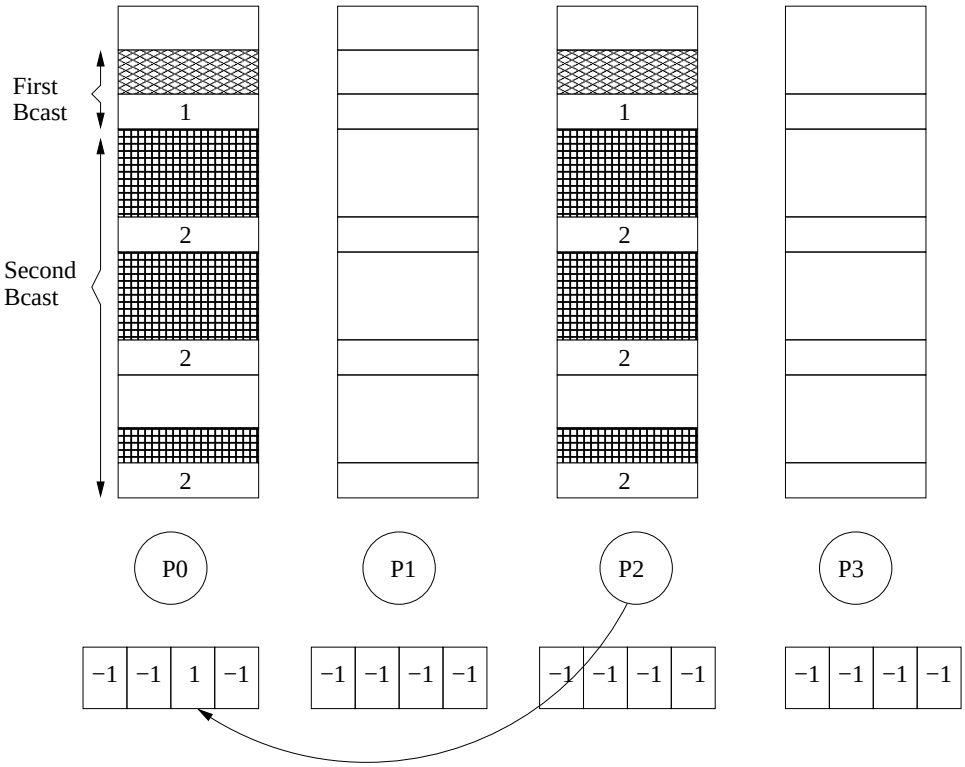


Figure 5.7: Notification by Process 2 to Process 1 before reusing first block

We evaluated out implementation on a cluster of 8 nodes, each with a 66MHz PCI bus, 700MHz Pentium III machines, 1GB of Main memory and Linux version 2.2.17. The machines are connected using a GigaNet 5300 switch.

5.4.1 Broadcast Benchmark

To obtain the broadcast latency, we ran 5000 iterations of *MPI_Bcast()* by using the algorithm given in Figure 5.8.

```
Synchronization Barrier
For 5000 iterations
   Start Timer at Root
        MPI_Broadcast
        If ( Node_Id == ROOT )
                MPI_Recv(Message From Last Leaf)
        If (Node_Id == LAST LEAF)
                Delay(T1)
                MPI_Send(4 byte ack to ROOT)
    Stop Timer
Time = (Stop − Start ) − (1way Latency of 4byte ack) − T1
```

Figure 5.8: The Broadcast Benchmark Algorithm

The benchmark algorithm does a broadcast to all the nodes. It however waits for an 4 byte acknowledgment from the last node. The last node is the one to whom the data reaches in the maximum number of steps and takes the longest time. The last node is made to incur a delay of time T1 (typically 20us in the experiments) before it sends back the acknowledgment. This is because we don't want the incoming acknowledgment to interfere with the *sends* that the root may still be doing. The

broadcast latency is the 1-way latency of the 4 byte acknowledgment and time T1 subtracted from the cumulative latency.

The broadcast buffers are divided into constant size blocks given by *block_size*. A large message has to be broken down into blocks of size *block_size - 1* and is thus scattered over many contiguous blocks at the remote end. The total number of blocks a large message is broken into depends upon the *block_size*. For example, if the block size is 3073 bytes, a 2048 bytes message can be sent as 1 block. This will require 1 RDMA write operation. However, since data has to be copied to a registered buffer at the *root* before sending, hence if the *block_size* is large, the copying cost will be higher. If the node is an intermediate node, it can forward the data to the other nodes first and then copy the data to its user buffer, thus overlapping the copying with the *sends*. If the node is a last node, the data will be copied immediately on receiving, hence for a large message the copying cost will be higher. This is demonstrated in Figure 5.9 for data size of 2K bytes.

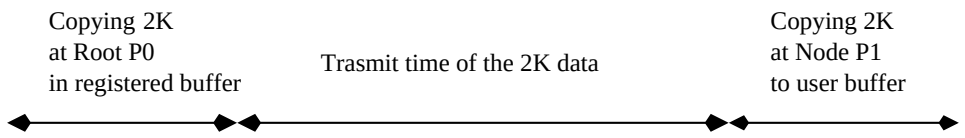


Figure 5.9: Timing Diagram for a 1 block send

However, if the block size is of 1025 bytes, a 2048 bytes message will be sent as 2 blocks of 1024 (with an additional *broadcast counter* byte in each block) bytes. Each block will require a different RDMA write operation. The advantage of this approach is that data can be copied in smaller chunks and overlapped with the send operation

as shown in Figure 5.10. If the node is the last node to receive data, the copying to the user buffer can also be overlapped. If it is an intermediate node, then we can perform the *sends* to the other nodes first and then copy the data to the user buffer. Notice however that the consecutive *sends* get sequentialized at the switch. Hence, if the *block_size* is too small, then for a large message we will have many *sends* to deal with. On a reliable connection of GigaNet cLAN, each of these *sends* will be acknowledged. Thus, processing the *sends* might offset the benefit obtained by overlapping the copies.

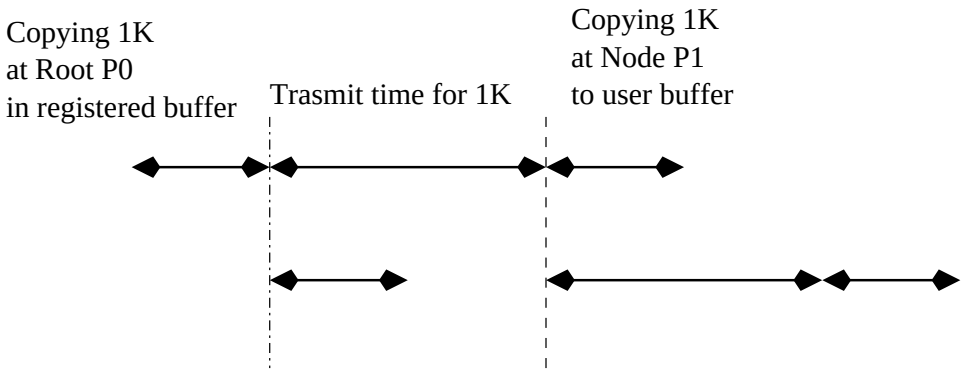


Figure 5.10: Timing Diagram for a 2 block send

We need to find an optimal *block_size* where the cost of processing multiple *sends* does not kill the benefit achieved by overlapping memory copies.

The message passing system sends data in blocks of 1024 bytes. In order to ensure fair comparison, we also tested the RDMA Broadcast with different block_sizes starting with 1025 (1 extra byte for the counter) bytes.

In Figure 5.11, we give a comparison of RDMA Broadcast with *block_sizes* of 1025, 2049, 3073 and 4097 bytes and the message passing Broadcast for small messages of

size 4 bytes to 1024 bytes for a 16 node cluster. Small messages from 4 bytes to 1024 bytes show the same timings because all the messages use 1 block to write to the remote node as the least *block_size* is 1025 bytes.

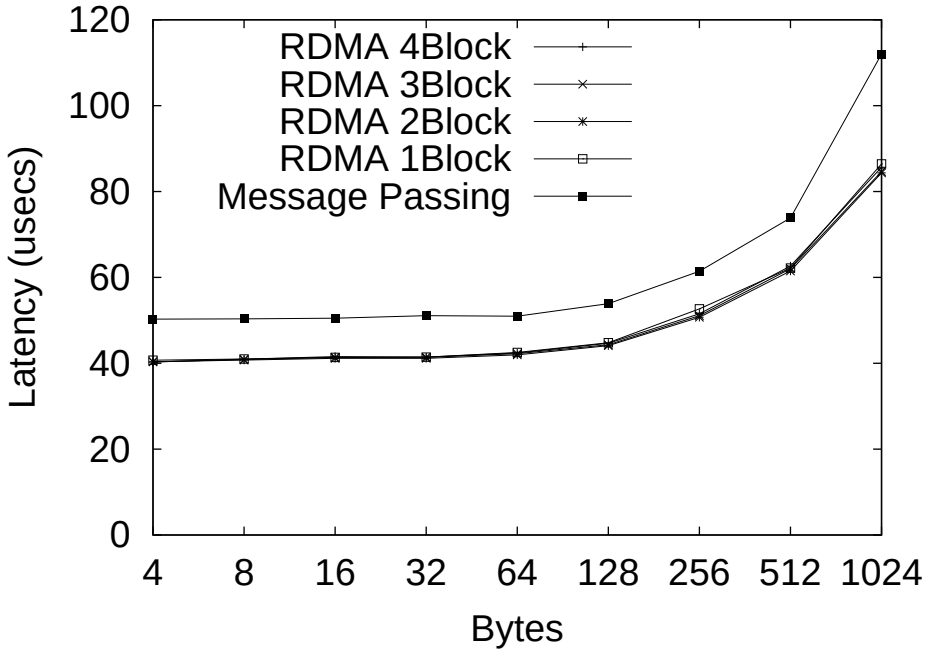


Figure 5.11: Comparison of RDMA Broadcast for varying block_size messages between 4-1024 bytes and Message Passing Broadcast

The difference can be seen in Figure 5.12. We see that to transmit 4096 bytes with *block_size* of 1025 bytes, we need 4 blocks. For higher *block_sizes*, the number of sends decreases and so do the timings. We obtain the best result for a *block_size* of 3073 bytes. In fact, a *block_size* of 3073 bytes gives the most optimal result for all message sizes from 1025 to 5000 bytes. RDMA Broadcast for all the given *block_sizes* gives better performance as compared to the message passing Broadcast.

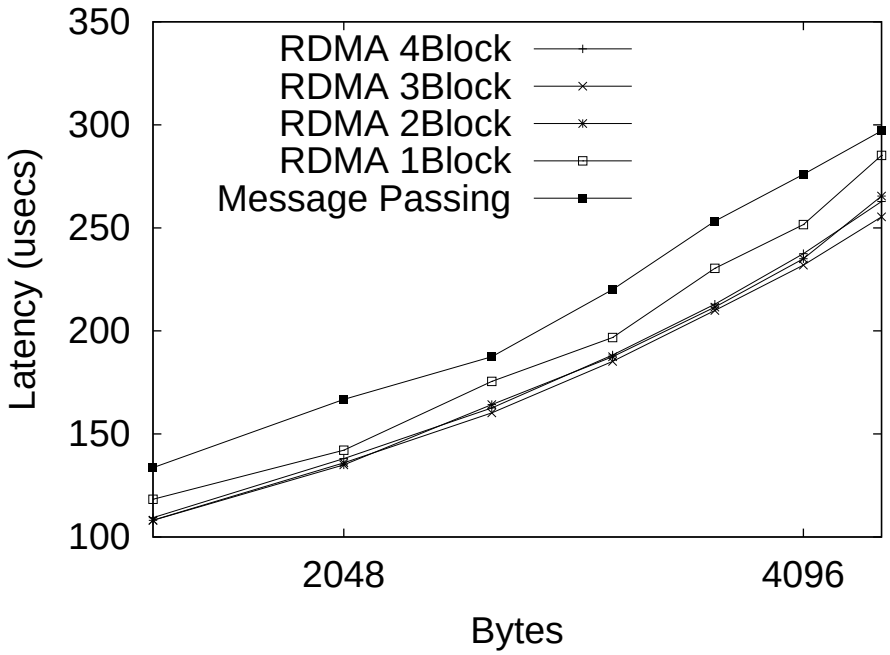


Figure 5.12: Comparison of RDMA Broadcast for varying block_size messages between 1025-4608 bytes and Message Passing Broadcast

Figure 5.13 and Figure 5.14 shows the comparison between a RDMA Broadcast with *block_size* of 3073 bytes and the Message passing Broadcast for 16 nodes.

For small messages of 4 bytes, we see a benefit of around 19.7%. For larger messages we see a benefit of around 14.4% for 4608 bytes.

5.5 Summary

In this chapter, we discussed the design issues and alternatives related to the RDMA Broadcast. We implemented RDMA Broadcast using the Binomial algorithm. Large messages were split into smaller blocks to bring about maximum overlapping and pipelining. We tested different *block_sizes* for RDMA Broadcast starting from

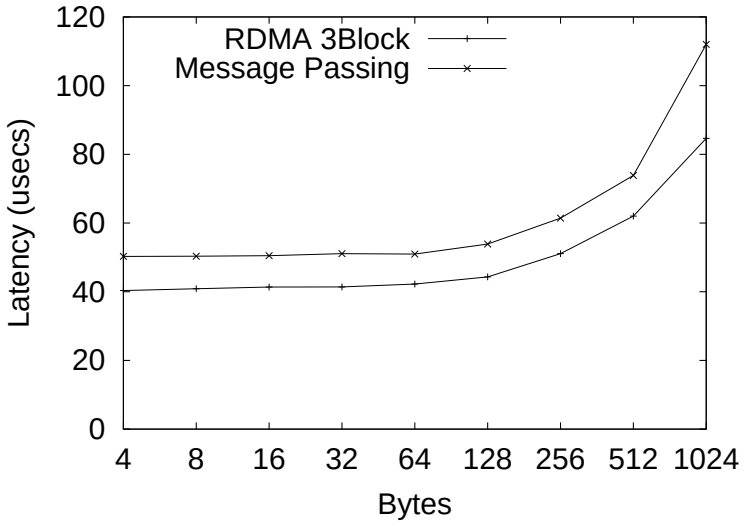


Figure 5.13: Comparison of RDMA Broadcast with block_size of 3073 bytes and Message Passing Broadcast in 16 node cluster for message size 4-1024 bytes

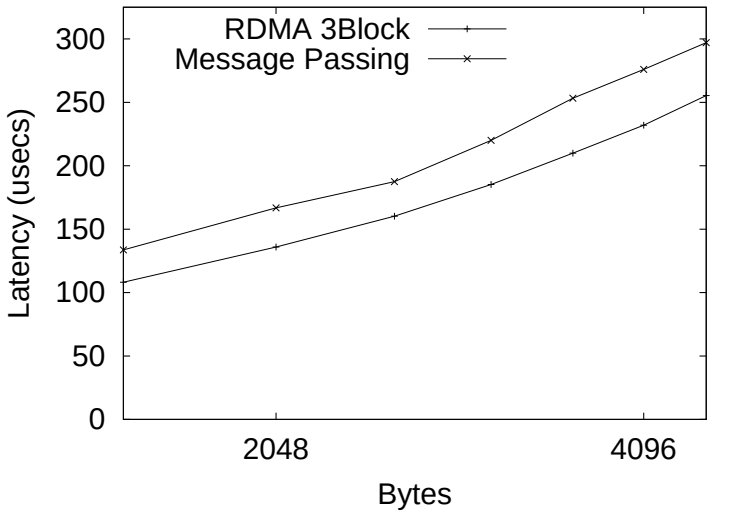


Figure 5.14: Comparison of RDMA Broadcast with block_size of 3073 bytes and Message Passing Broadcast in 16 node cluster for message size 1536-4608 bytes

1025 bytes. RDMA Broadcast for all *block_sizes* gives better performance than message passing Broadcast. However the best results are obtained for *block_size* of 3073 bytes. The RDMA Broadcast of *block_size* of 3073 bytes gives us a 14.4% improvement for 4608 bytes and 19.7% improvement for 4 bytes on a 16 node cluster as compared to the Message Passing Broadcast.

The next chapter will talk about the RDMA AllReduce collective operation.

CHAPTER 6

THE RDMA ALLREDUCE COLLECTIVE OPERATION

The AllReduce collective operation is a global reduction operation, which takes place across all the members in the communicator. The AllReduce operation is a variation of another global operation called Reduce. The Reduce operation combines values from different processes based on the reduction operation and the result is stored at the node labeled as the root. In the AllReduce operation, the result is not stored at a single node but communicated back to all the nodes in the group.

Most traditional methods implement AllReduce as a combination of the following two operations :

(i) Reduce Operation where the result is stored at the *root*

(ii) The Broadcast operation, which sends the computed data present at the *root* to all nodes in the communicator.

For the RDMA implementation of AllReduce, we concentrate on the implementation of the RDMA Reduce part. The RDMA Broadcast part has been discussed in the previous chapter.

This chapter starts with an introduction of the AllReduce operation. In the later sections, we explore different RDMA-based AllReduce algorithms. We, however, explain the buffer management issues and the design choices from the point of view of

the binomial AllReduce algorithm. The latter part of the chapter discusses an analytical scheme for choosing the best RDMA AllReduce algorithm based on the data size and the number of nodes involved. We conclude the chapter with a discussion of the results obtained.

6.1 Introduction to the AllReduce Operation

The AllReduce combines values, based on the type of a reduction operation, from all the processes and distributes the result back to all the processes. The reduction operation can be a user-defined operation or a predefined operation. MPI provides around 12 common pre-defined reduction operations which include finding the summation, maximum, minimum, product and performing bitwise operation on the data distributed across the set of nodes.

The AllReduce collective operation has the following definition in the MPI Standard. This definition gives us a clearer picture of the elements involved in the All Reduction operation.

MPI_AllReduce(Send buffer, Receive buffer, Count, Datatype, Reduction operation, Communicator)

Sendbuffer contains the data of type *Datatype* on which the operation specified by *Reduction operation* will be performed. *Count* indicates the total number of elements in the *Sendbuffer*. *Receivebuffer* is the buffer which contains the computed results.

All the processes use the same count, data type, reduction operation and communicator. The MPI standard says that the *Reduction operation* whether predefined or user defined is always assumed to be associative. Also, all the predefined operations

are commutative. However, the users are allowed to define operations that are associative but not commutative. The MPI standard allows Reduce implementations to change the order of evaluation in order to take advantage of the commutativity and associativity properties of the operation. However, changing the order of evaluation may change the result of reduction for operations like floating point addition, which are not strictly associative and commutative.

The AllReduce operation is demonstrated in Figure 6.1 where we see 4 processes P0, P1, P2 and P3. The operation defined is MPI_SUM and each process contains the elements 1, 2 in its send buffer. After the AllReduce operation, the result (which is the sum of the individual elements) i.e, (4, 8) is present is the receive buffers of each process.

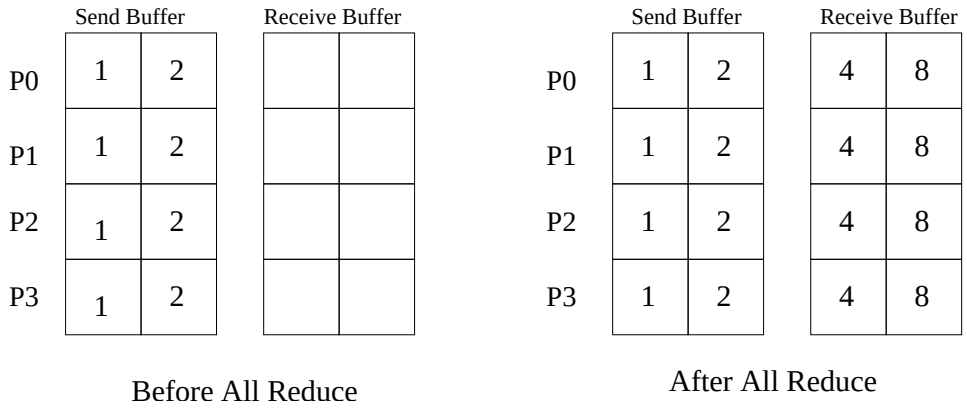


Figure 6.1: AllReduce Operation between 4 processes

6.2 The RDMA All Reduce Algorithms

The RDMA AllReduce is implemented as a combination of RDMA Reduce followed by a RDMA Broadcast operation. The node with *rank 0* is chosen as the *root* for both the RDMA Reduce and RDMA Broadcast operation.

There are various ways in which the Reduce part of the AllReduce operation can be implemented. In this thesis, we introduce a Degree-k tree based scheme for implementing this Reduce part.

Definition: An *Degree-k tree-based AllReduce* defines a tree where any node can receive messages from at most k nodes in any step of the Reduce operation. The variable k is a *(power of 2) - 1* value. For a cluster of size N, where N is a *power of 2* value, we can use all those *Degree-k tree-based* AllReduce schemes, where k is *(power of 2) - 1* and $k < N$.

Computing Clusters generally have a *power of 2* size. Hence, we explore and constrain the Degree-k tree-based RDMA AllReduce scheme to a cluster having *power of 2* nodes. Hence, in the remaining part of the chapter, the cluster size is implicitly assumed to be a *power of 2*, when it is mentioned in relation to the Degree-k tree-based RDMA AllReduce scheme.

To understand the Degree-k tree-based AllReduce concept, let us consider a 4 node cluster. An AllReduce operation on such a cluster can be implemented using Degree-1 or Degree-3 tree-based RDMA AllReduce scheme. Figure 6.2 shows the tree for 4 processes P0, P1, P2, and P3 having *rank ids 0, 1, 2 and 3 respectively.* The square brackets indicate the *step number* for that node. In a Degree-1 tree-based RDMA AllReduce scheme, every node will receive data from at-most 1 node in each step. Hence, in the first step Process P1 and Process P3 send data to Process P0

and Process P2 respectively. P0 will perform the required computation with the data acquired from P1 and store the result. Similarly, P2 will perform the computation with the data received from P3 and store the result. In the second step, Process P2 forwards its computed result to Process P0. Process P0 will then perform the computation with its own result of the previous step and the newly received result from P2 to get the final result. This final result will then be broadcasted to all the other nodes involved in the AllReduce operation. The Degree-1 tree-based RDMA AllReduce scheme is similar to the binomial algorithm, which is used in the mvich-1.0 implementation for message passing.

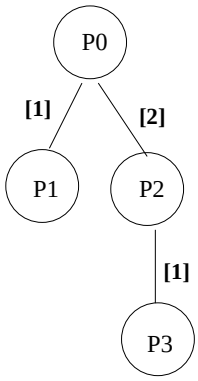


Figure 6.2: Degree-1 tree-based RDMA AllReduce between 4 processes

Consider a Degree-3 tree-based RDMA AllReduce scheme on the same cluster. In the Degree-3 tree-based RDMA AllReduce scheme, a node can receive data from at-most 3 nodes in a step. Hence, as seen in Figure 6.3, processes P1, P2 and P3, having *ranks 1, 2 and 3* respectively send the data to process P0, which has *rank 0*. P0 will first perform the reduction operation on its own data and on the data sent by the node having *rank 1*. The second operation is performed by P0, on this new result

and the data sent by Process P2, having *rank 2*. The last operation is done on the most recently computed result by P0 and the data sent by Process P3, with *rank 3*. Thus, P0 will choose the order of evaluating the data based on the ascending order of the ranks of the sending nodes.

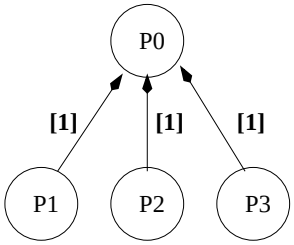


Figure 6.3: Degree-3 tree-based RDMA AllReduce between 4 processes

A Degree-3 tree-based RDMA AllReduce for a 32 node cluster will have the tree as described in Figure 6.4. The square brackets indicate the *step number* for that node.

Thus, as stated by the definition, a 8 node cluster can implement Reduce using a Degree-1, Degree-3 or Degree-7 tree-based RDMA AllReduce scheme and a 16 node cluster can use a Degree-1, Degree-3, Degree-7 or Degree-15 tree-based RDMA AllReduce scheme.

There is a trade-off involved between the number of steps in the Degree-k tree-based RDMA AllReduce collective operation and the overhead incurred by the node in performing the reduction operation. For example, in a Degree-1 tree-based RDMA AllReduce scheme in a 4 node cluster, there are 2 steps involved and in each step, a receiver node receives only 1 message and hence performs only 1 operation. In a Degree-3 tree-based RDMA AllReduce scheme in a 4 node cluster, the number of

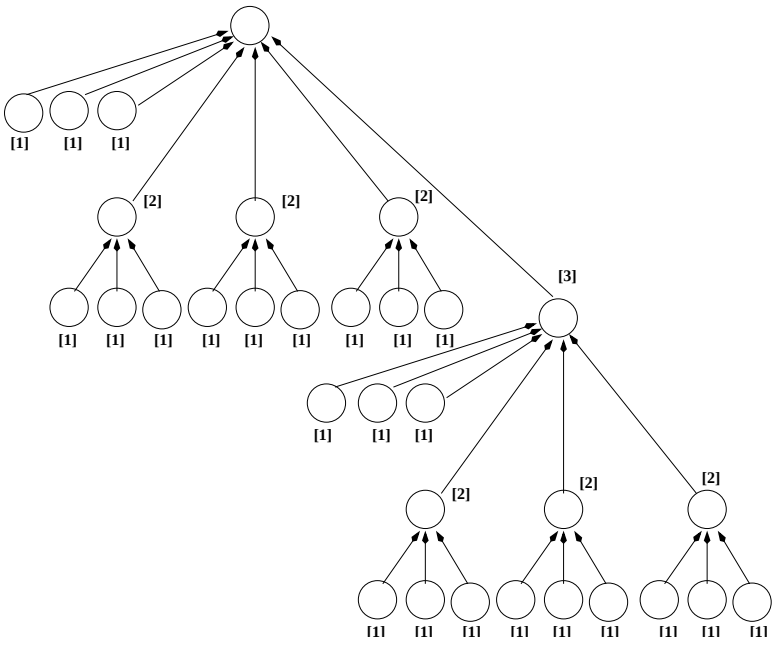


Figure 6.4: Demonstration of steps in a Degree-3 tree-based RDMA AllReduce in a 16 node cluster

steps is 1 but 3 nodes are sending the data and hence we have to do 3 operations in that step. So, depending upon the number of nodes, the number of steps and the number of operations involved, we can choose different Degree-k tree-based RDMA AllReduce algorithms.

6.3 Design solutions for RDMA All Reduce operation

In this section, we discuss the design solutions for Degree-k tree-based RDMA AllReduce algorithms. The design solution for all the Degree-k tree-based RDMA AllReduce algorithm are the same. However, we will explain the design solution from the point of view of a Degree-1 tree-based RDMA AllReduce algorithm, which happens to be the same as the binomial algorithm.

We implement the RDMA AllReduce operation by using the Degree-1 tree-based scheme for Reduction as described in Figure 6.2. Processes P0, P1, P2 and P3 in this figure will serve as an example for the discussion of design issues. The processes P0, P1, P2, P3 have *ranks 0, 1, 2, 3* respectively relative to the root. Assume that every process contains 2 elements of value 2 in its send buffer and a summation AllReduce operation is to be performed.

6.3.1 Registration of buffers and Address Exchange

The RDMA Reduce algorithm works in 2 modes, depending on the size of the data to be transferred. These modes are similar to the RDMA Broadcast modes discussed in the previous chapter. We choose the static registration scheme for data size lesser than $5K$ because memory copy for smaller bytes is not very expensive. For data size larger than $5K$, memory copy becomes expensive and so we use the dynamic registration scheme.

Static Registration Scheme

For messages less than $5K$ bytes, we allocate a contiguous section of memory and register it during the initialization time. The memory region is broken down in *block_size* of *(5K + 1)* bytes, because *5K* is the maximum size of data that can be transferred in the static mode. Also, the total memory region reserved need not be greater than $block\_size * N$, where N is the number of processes in the communicator. This is because, in the *(N-1)-RDMA* AllReduce algorithm case, a maximum of $N-1$ processes can write to a receiver node. For sending the data in the static scheme, the sender RDMA writes the data to the receiver's AllReduce buffers. Figure 6.5 shows the 4 processes, each having 4 contiguous blocks of memory reserved and registered

for the AllReduce operation. The address of the Reduce buffer space is exchanged in the address exchange phase.

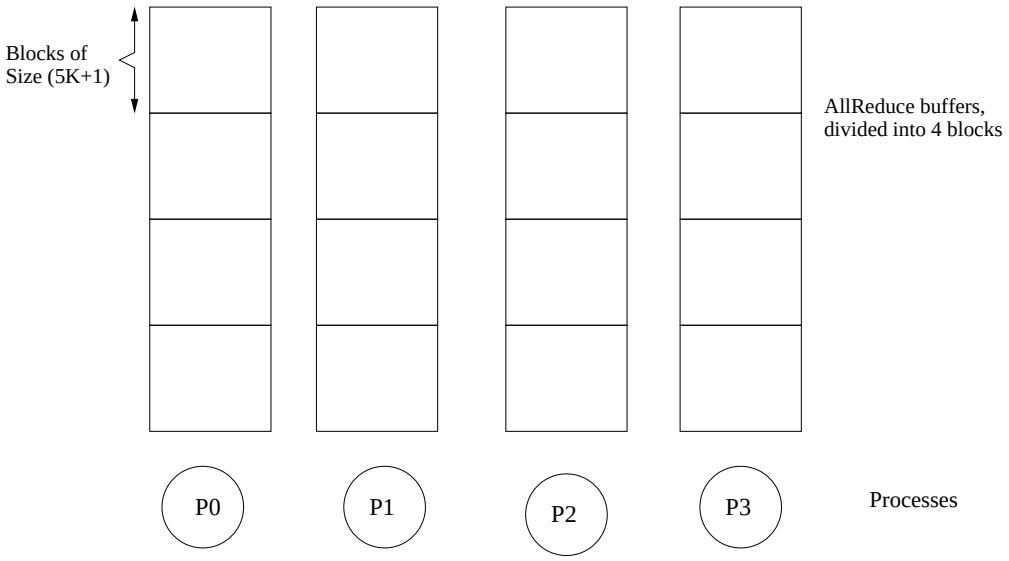


Figure 6.5: Buffer management in All Reduce in a 4 node cluster

Dynamic Registration Scheme

For messages greater than $5K$, we follow the *rendezvous* scheme as described in the Broadcast chapter. A request and a reply message containing the receiver user buffer address are exchanged before the operation. Then, the sender can RDMA write the data to the receiver's user buffer directly. As mentioned earlier, there is an overhead of an extra round trip time involved in such a type of data transfer. This is the same mechanism that is implemented by mvich-1.0 for messages greater than $5K$ and because we do no other optimization to it, hence the remainder of the thesis will be discussed from the point of view of the static scheme only.

6.3.2 Buffer Initialization

The AllReduce buffers are initialized to -1 at the start of the program. However, re-initialization is not required for the AllReduce collective operation.

6.3.3 Data Validity at the Receiver end

Sending of the data in AllReduce is similar to that in broadcast operation. Consider the node P1 sending data to node P0 and node P3 sending data to node P2.

The node P1, with *rank 1*, RDMA writes the data to block no. 1 of the receiver's AllReduce buffer. A node always writes the data to the block having the same number as its rank. As mentioned earlier, the AllReduce buffers are split into a maximum of N blocks, each of $5K+1$ size. Hence, any node with any rank can write to any other node's AllReduce buffer at a location indicated by its rank. This indicates to the receiver the sender identification for the data and also enables an ordered evaluation of the data.

Node P1 sends the entire data in 1 single RDMA write to the node P0. Appended to the data is an *allreduce_counter* byte. For a communicator, every process in that communicator has a static *allreduce_counter*, which is incremented for consecutive AllReduce operations involving the same communicator. Since all nodes are aware of the *allreduce_counter* value, it serves as means for received data indication at the receiver end. The data is written to the receiver AllReduce buffer in a bottom up manner, so that the *allreduce_counter* is always written in the last byte of the block. The receiver node knows the *allreduce_counter* value and hence can poll for the data to arrive. On getting the data, the receiver can perform the required operation. The result is stored in the same location as that of the latest received data. Figure 6.6

shows node P1 with *rank 1* and node P3, with *rank 3*, writing the data to node P0's block no. 1 and to node P2's block no. 3 respectively. Assuming this is the first AllReduce, hence *allreduce_counter byte* is set to 1.

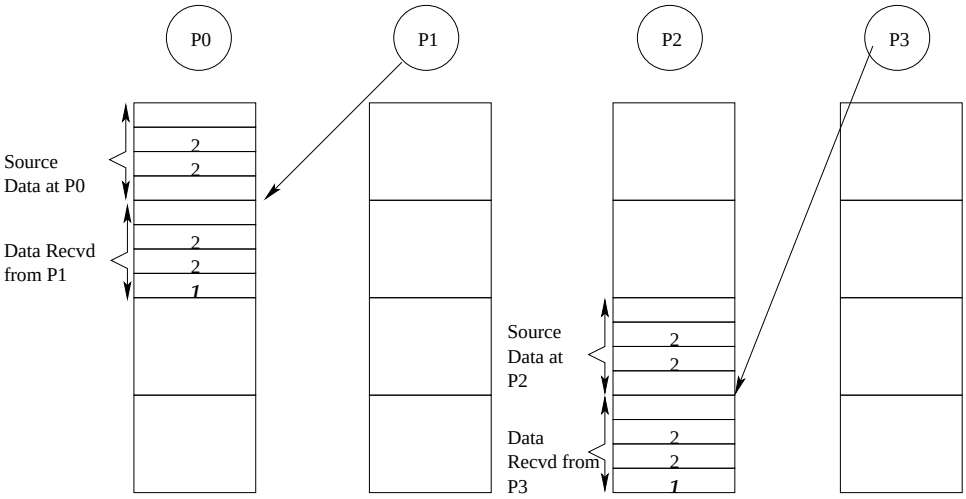


Figure 6.6: Step 1 of Degree-1 tree-based RDMA AllReduce

Figure 6.7 shows node P0 and node P2 performing the operation and writing the intermediate results in blocks no. 1 and block no. 3, respectively.

Data is not broken down into smaller blocks and sent because the entire data is needed to perform the computation. If data were sent in blocks, then there would be an additional overhead of assembling and packing the data together before performing the required computation. The overhead of copying and packing data is larger than the overhead of sending the entire data in a single RDMA write operation.

In the second step of the Degree-1 tree-based RDMA AllReduce, node P2 with *rank 2* will RDMA write its computed result to the AllReduce block. no 2 of node P0

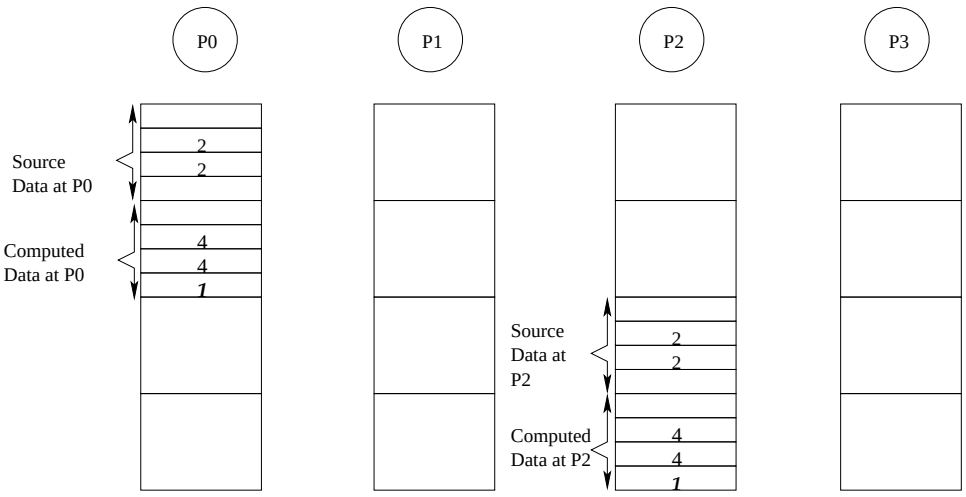


Figure 6.7: Reduce Computation at P1 and P2

with the *allreduce_counter* of 1 attached at the end of the data as shown in Figure 6.8.

Figure 6.9 shows node P0 performing the operation on the newly received data from node P2 and its own computed result obtained in the previous step. The result of this operation is stored in block no. 2 at P0.

The result is copied by the root, i.e., node P0 to its receive buffer after it is done with its final computation. The result is broadcasted from this receive buffer to all the nodes.

6.3.4 Buffer reusing

The blocks can be safely reused by the nodes without any additional messages being sent. In an AllReduce, the Reduce operation is followed by a broadcast. Consider two consecutive AllReduce operations. In the first operation, the nodes write the data to the specified destinations. The second AllReduce operation starts only

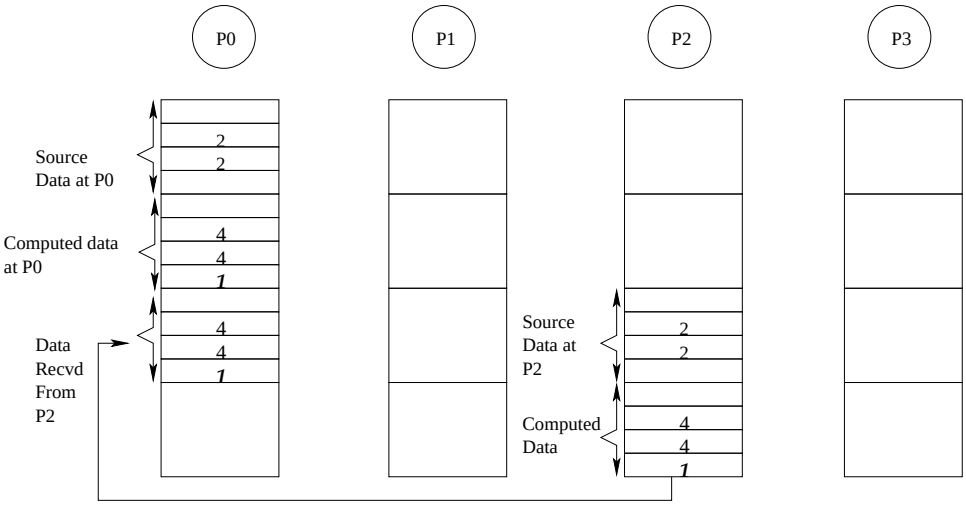


Figure 6.8: Step 2 of Degree-1 tree-based RDMA AllReduce

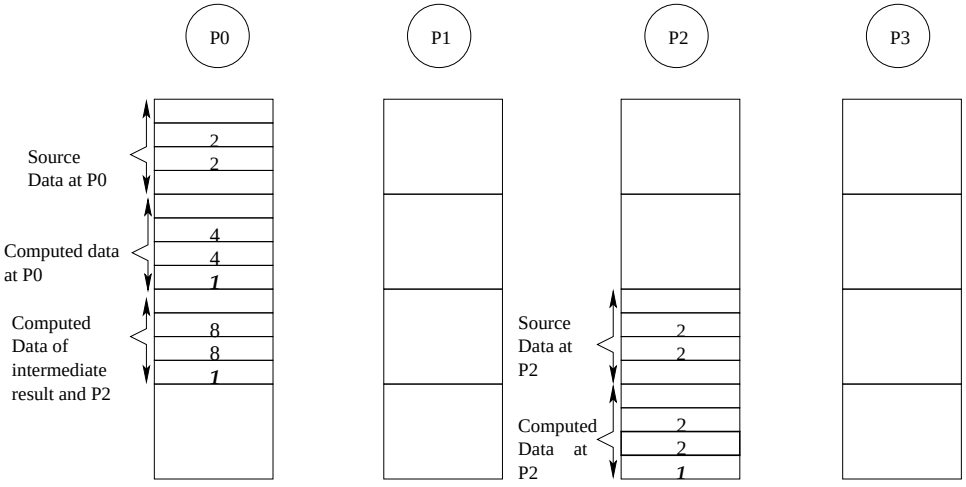


Figure 6.9: Final Reduce Computation at P0

after all the nodes have received the broadcasted results of the first operation. Hence, the nodes can RDMA write the data to the same locations, as the data previously written has been used for successful computation.

6.4 Selecting the right algorithm using an Analytical Model

For a given set of *power of 2* nodes, we have different algorithms available. In this section, we present an analytical model for Degree-k tree-based RDMA AllReduce, which enables us to choose the right value of the variable k on the basis of some known parameters. We use this generalized model to give an estimate of the degree k that would be suitable for an All Reduction operation for a certain number of nodes and a given message size in bytes.

We base the analytical model on the following parameters :

1. Message Size (given in *bytes*)
2. Total Time taken for performing the *Reduction operation* for the total *count* elements (given as *T_operation*)
3. The data transmit time (given as *T_transmit = T_transmit_per_bytes * bytes*)
4. DMA Startup time (given as *T_startup*)
5. Memory Copy Rate (given as *T_copy_rate*. The total copy time is given as *T_copy*)
6. NIC Processing time after the descriptor is posted and data needs to be sent (given as *T_nic*)

We assume that *T_nic* is less than *T_startup*. Also, for large messages, *T_operation* is much lesser as compared to *T_transmit* but the same is not true for very small messages having a smaller count.

6.4.1 Events in an AllReduce Message Transfer

A message transfer at the sender side consists of the following events and overhead. An MPI call is made to do the RDMA write. Hence, there is the overhead due to the MPI library. We term this overhead as the *T_mpi*. The data has to be copied to a registered buffer. Hence, there is a copying cost involved. Some amount of time is spent in posting the send descriptor. This time is termed as *T_descriptor*. The data is then DMAed to the NIC. There is a DMA Startup overhead associated with each message or frame that is DMAed. The data is then processed by the NIC and sent on the wire. Thus, at the sender side we have the copying cost, the MPI overhead, the time for posting a send descriptor, the DMA Startup time and the NIC processing time

The data is then transmitted to the destination. If many nodes are sending data to the same destination, the data gets sequentialized at the switch connected to the destination node.

At the receiver end, the destination NIC receives the data and processes it. It finds the destination address and the data is DMAed to that address. The host on receiving the data, performs the reduction operation. Thus at the receiver side, for every message, we have the NIC processing cost, the DMA startup cost and the operation cost.

After performing the required operation with all the received data and obtaining the result, the data has to be copied to the user specified buffer. This result can then be broadcasted to all the other nodes.

The analytical model has various cases based on the values of the above parameters. In the following subsections we present the analytical equations with the time diagrams for all these cases. For all the examples, we consider a Degree-3 tree-based RDMA AllReduce scheme in a 4 node cluster having processes P0, P1, P2 and P3 with *ranks 0, 1, 2, 3* respectively. Hence, the AllReduce operation takes place as shown in Figure 6.3. It involves one step with P1, P2, P3 sending to P0 and all the operations take place at P0.

6.4.2 Handling Large Messages

Messages where *T_transmit > (T_nic + T_startup)*, i.e., messages that have transmit time more than the sum of the NIC level processing and DMA startup time are categorized as large messages.

For large messages, the transmit time is very high. In the Degree-k tree-based RDMA AllReduce case, the receiver operates on the data on the basis of the rank of the node sending the data. Hence, if P1, P2 and P3 are sending the data, the receiver polls for the data from the node with *rank 1*, namely P1 to arrive first and first operates on this data. Since, many nodes are sending data to the same destination, the messages from these nodes get sequentialized at the switch. Since, any message from any node can arrive first at the destination NIC, there is a fair probability of not getting the required message when needed. Hence, the analytical model for RDMA AllReduce gives the best and the worst time estimates.

The best time estimate assumes that the required data is the first to arrive, the DMA Startup and the NIC processing can be overlapped with each other.

The worst time estimate assumes that the required data is the last to arrive. It also assumes that the NIC processing and the DMA Startup can't be overlapped due to sharing to the system bus.

To understand this concept, consider a Degree-3 tree-based RDMA AllReduce scheme in a 4 node cluster, where P1, P2 and P3 write to P0. Figure 6.10 shows the time line chart for the events that happen at the sender side. Since P1, P2 and P3 send data at the same time, the copying of data, MPI overhead, posting of descriptor, DMA startup and NIC Processing for the various processes gets overlapped at the sender side. During this period, the Process P0 is polling for data from Process P1, which has the rank 1, to arrive. Thus, the sender side cost can be termed by the parameter *T_sender*.

Thus, *T_sender = (T_copy_rate * bytes) + T_mpi + T_descriptor + T_nic + T_startup*.

Figure 6.11 shows the best case scenario at the receiver end. The message from P1 arrives first to the switch. When this message reaches node P0, it does the NIC level processing, DMAing and starts the reduction operation. The transmission of the second message happens in parallel with the NIC processing, DMAing and operation of the first message. By the time the destination NIC receives the next message, it has already finished processing the first one. By the time the host process receives the second message, it has finished the operation on the first one. For large messages, the *T_operation* parameter is very small as compared to the *T_transmit* parameter. This is the perfect scenario where the time taken at the receiver is *(T_transmit **

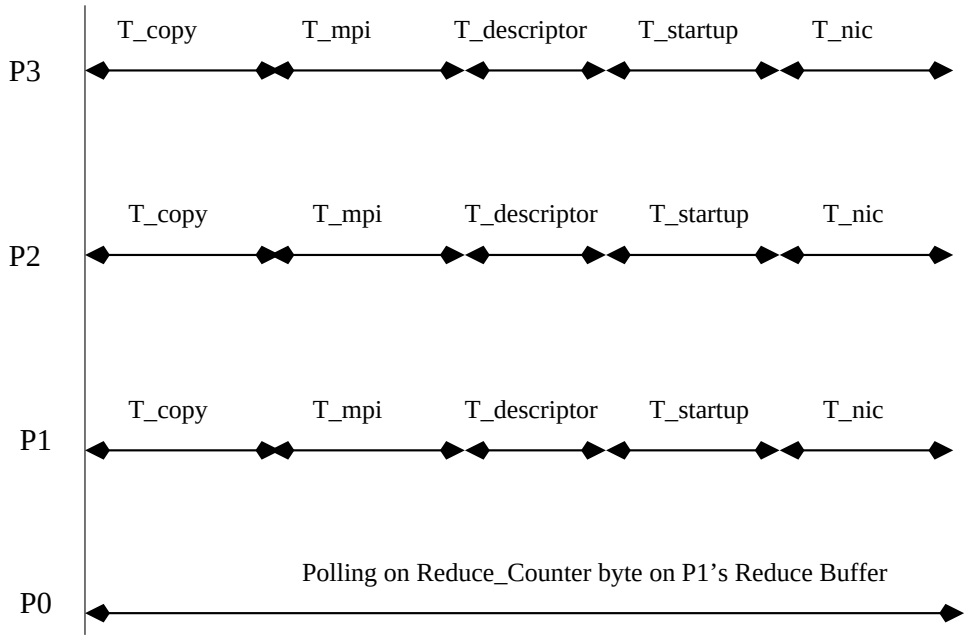


Figure 6.10: Sending side events

No. of Sending nodes) + T_nic + T_startup + T_operation. There is also a copying cost involved after the operation is done. Hence the total time for the entire Reduce operation is :

*T_sender + (T_transmit * No. of Sending nodes) + T_nic + T_startup + T_operation + (T_copy_rate * bytes).*

The worst case is depicted in Figure 6.12 where the message from P1 reaches last. P0 can start performing the operations only after receiving data from P1. Hence, the time at the receiver end is given as: *(T_transmit * No. of Sending nodes) + T_nic + T_startup + (T_operation * No. of Sending nodes).*

Thus, the total time for the entire Reduce operation is :

*T_sender + (T_transmit * No. of Sending nodes) + T_nic + T_startup + (T_operation * No. of Sending nodes) + (T_copy_rate * bytes).*

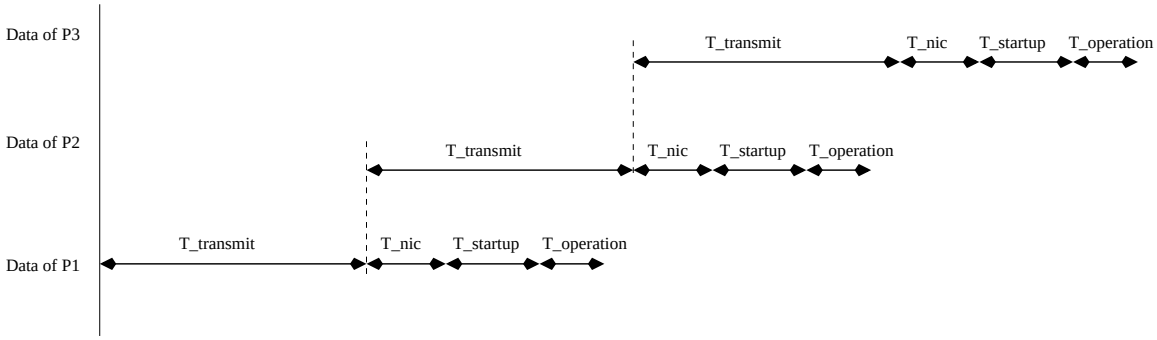


Figure 6.11: Best case receiver scenario for large messages

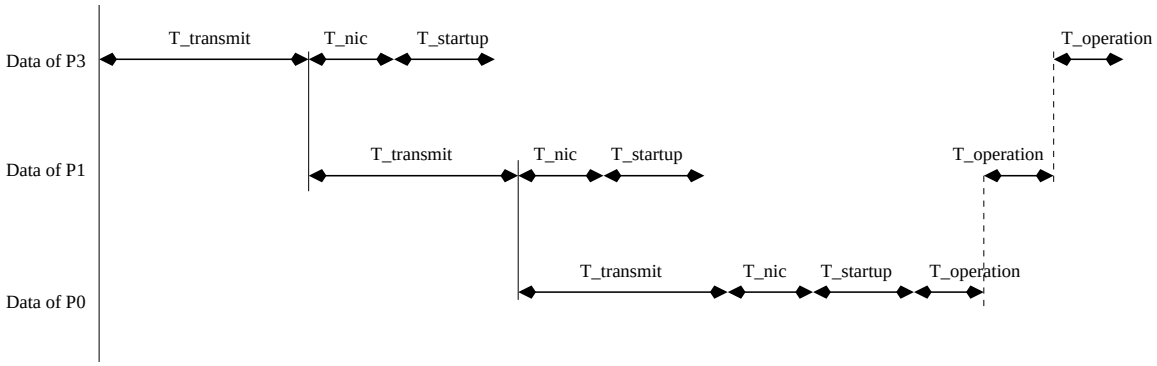


Figure 6.12: Worst case receiver scenario for large messages

6.4.3 Handling Small Messages

Messages whose transmit time is less than or equal to the sum of the destination NIC processing and DMA startup time are categorized as small messages, i.e., *T_transmit <(T_nic + T_startup).*

As the length is less, the number of operations to be done is also less. The events at the sender side remain the same as shown in Figure 6.10. The events at the receiver are shown in Figures 6.13 and 6.14. At the receiver side, the messages from various nodes destined for one node are still sequentialized, however since the amount of data to be transmitted is less, hence the transmit time is very less. When the transmit time is less, the operation time *T_operation* for such messages is also very less. In many cases, the NIC processing and the DMA startup cost will be the major contributors of the overhead.

The evaluation of the best cases for small messages are divided into two sections. We consider that case first where *T_transmit <= T_startup* and this best case is shown in Figure 6.13. Here we assume that the NIC processing can be done in parallel with the DMA Startup and that the required data is always obtained first. Hence, the message from P1 is the first to reach P0. When P0's NIC is processing the message, DMAing it and performing the operation, the message from P2 can be transmitted and processed. The time taken at the receiver end is given by *T_transmit + T_nic + (T_startup * No. of Sending nodes) + T_operation.*

The total time for All Reduce for the best case scenario is :

*T_sender + T_transmit + T_nic + (T_startup * No. of Sending nodes) + T_operation + (T_copy_rate * bytes).*

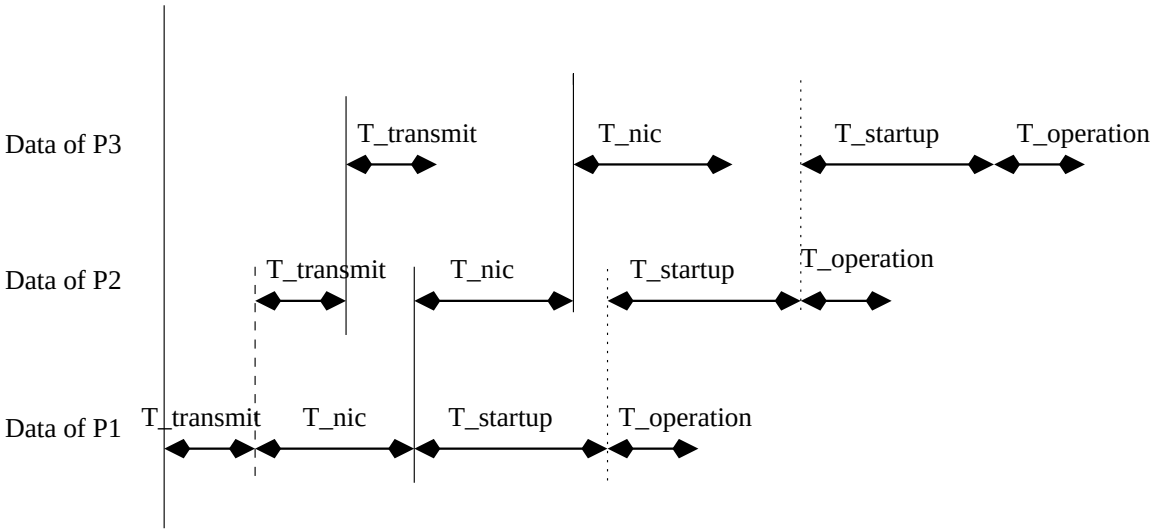


Figure 6.13: Best case receiver scenario for small messages where *T_transmit* <= *T_startup*

The second case is when *T_transmit* > *T_startup*, as shown in Figure 6.15. The message from node P1 reaches node P0 first. The transmit time and the nic processing of the second message is overlapped with the first one. However, as *T_transmit* > *T_startup*, hence the *T_startup* time for the second message is delayed till the *T_startup* has been completed. The duration of this delay is given as the *T_transmit* - *T_startup* time.

Hence, the best estimate of the receiver time is given by *T_transmit + T_nic + (T_startup * No. of Sending nodes) + ((T_transmit - T_startup) * (No. of Sending nodes - 1)) + T_operation.*

The total time for All Reduce for the best case scenario for *T_transmit* > *T_startup* is :

*T_sender + T_transmit + T_nic + (T_startup * No. of Sending nodes) + ((T_transmit - T_startup) * (No. of Sending nodes - 1)) + T_operation + (T_copy_rate * bytes).*

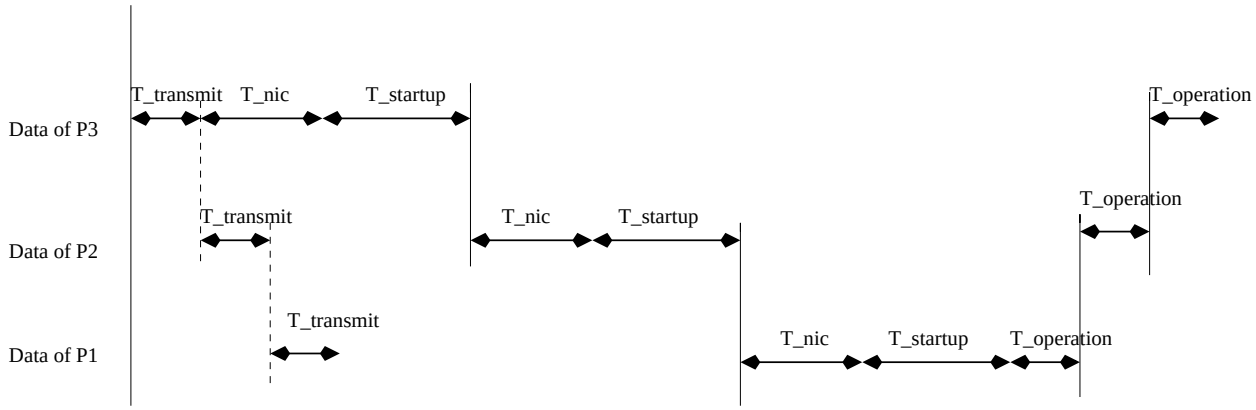


Figure 6.14: Worst case receiver scenario for small messages

If the NIC processing and the DMA Startup can't be overlapped, then we have a worse case scenario as shown in Figure 6.14. Here, we also assume that the required data is the last to arrive. Hence the data sent by Process P3 reaches first and is processed by the NIC and DMAed. Meanwhile, the message from node P2 arrives but it can't be processed because the NIC is busy processing and DMAing the first message received from node P3. So the NIC processing and DMAing are sequentialized for each message with no overlap happening. When the data from node P1 arrives, the first operation is done. Thus the worst case scenario at the receiver end is : *T_transmit + (T_nic + T_startup + T_operation) * No. of Sending Nodes.*

The total time for the worst case scenario for small messages is :

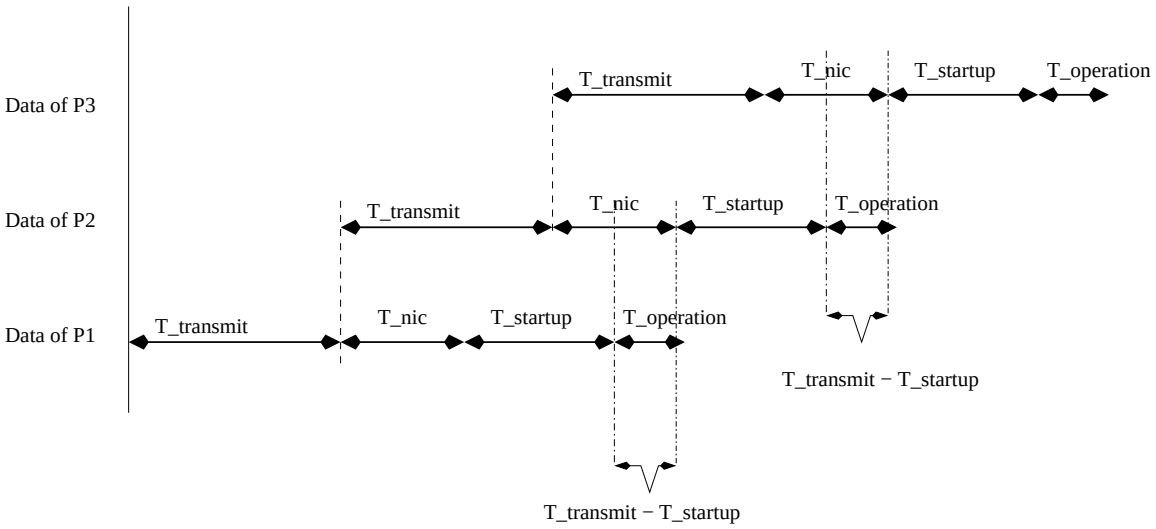


Figure 6.15: Best case receiver scenario for small messages where *T_transmit* > *T_startup*

*T_sender + T_transmit + (T_nic + T_startup + T_operation) * No. of Sending Nodes + (T_copy_rate * bytes).*

The pseudo code for getting the values analytically for the various Degree-k tree-based RDMA AllReduce algorithms is shown in Figure 6.16.

The variable k stands for Degree-k tree-based RDMA AllReduce value,i.e., the maximum number of nodes from whom a node can receive data in one step. The function calculates the best and the worst time estimates based on the various parameters provided. The first *while loop* iterates for the total number of complete steps, where *complete steps* are defined as steps where k nodes send to other nodes in an Degree-k tree-based RDMA AllReduce algorithm.

The second part of the code is called when the number of nodes sending data to a node is lesser than k. This function works for *power of 2* clusters, i.e. when the variable $Nodes$ is a *power of 2* and we are testing for a Degree-k tree-based RDMA

```
Quotient      = Nodes / ( k + 1 );
T_transmit  =  bytes  *  T_transmit _per_byte;
Best_time    =  Worst_time  =  0;


while ( Quotient  !=  0 ) {
      Sender_side = T_mpi + T_descriptor + T_startup + T_nic
     If  ( T_transmit >  ( T_startup  + T_nic ) ) then
            Receiver_side  = (  T_transmit  *  Nodes  )  +  T_startup  +  T_nic  +  T_operation
            Best_time    +=  Sender_side  +  Receiver_side ;
            Worst_time +=  Sender_side  +  Receiver_side  + T_operation * ( k  − 1 )
    else If  (T_transmit <  (T_startup + T_nic )  ) then
            If  ( T_transmit <  T_startup  ) then
                 Receiver_side = T_transmit +  T_nic + (T_startup * k) + T_operation
            else
                 Receiver_side = T_transmit +  T_nic + (T_startup * k) + (T_transmit − T_startup) * (k−1)  + T_operation
          Best_time    +=  Sender_side + Receiver_side
          Worst_time +=  Sender_side + Receiver_side +  (T_nic + T_operation ) * (k − 1)
          Endif
     Endif
     Dividend = Quotient ;
     Quotient = Dividend/(k+1);
     Remainder = Dividend % (k+1)
End− while

If ( Remainder != 1 ) then
      Sender_side  = T_mpi + T_descriptor + T_startup + T_nic
     If  (  T_transmit  >  ( T_startup + T_nic ) ) then
            Receiver_side  = ( T_transmit  * ( Remainder  −  1 )  ) + T_startup + T_nic + T_operation
            Best_time     +=  Sender_side  +  Receiver_side
            Worst_time  +=  Sender_side  +  Receiver_side  +  T_operation  *  ( Remainder  −  2 )
      else if (T_transmit <  (  T_startup + T_nic  )  ) then
           If  ( T_transmit <  T_startup  ) then
                Receiver_side = T_transmit +  T_nic + (T_startup * (Remainder −1)) + T_operation
           else
                Receiver_side = T_transmit +  T_nic + (T_startup * (Remainder −1 )) +
                                  + ( ( T_transmit − T_startup) * (Remainder −2 ) )  + T_operation
            Best_time    +=  Sender_side + Receiver_side ;
            Worst_time +=  Sender_side + Receiver_side +  ( T_nic + T_operation) * (Remainder − 2)
           Endif
      Endif
Endif
Worst Time += ( bytes * T_copy_rate ) * 2
Best_Time   += ( bytes * T_copy_rate ) * 2
```

Figure 6.16: Pseudo code for Optimal All Reduce Algorithm

AllReduce where the value of k is a *power of two - 1* and $k < Nodes$. It gives us the best and the worst estimates between which the actual values might lie.

On comparing the analytical values with the practical ones observed, we conclude that the results obtained have a maximum error rate of 8% from the best value and the worst value. We show some of the results in the performance section.

6.5 Performance Results

In this section, we discuss various results related to RDMA AllReduce. We start with a comparison between the current binomial message passing algorithm in mvich-1.0 and the Degree-1 tree-based RDMA AllReduce scheme. We than compare various Degree-k tree-based RDMA AllReduce algorithms for different cluster sizes and message sizes and different values of k. We also give a comparison between the current binomial message passing algorithm and an assorted combination of Degree-k tree-based RDMA AllReduce algorithm for different cluster sizes and different number of bytes. We also compare various Degree-k tree-based RDMA AllReduce with similar Degree-k tree-based message passing AllReduce algorithms.

For all our timings, we use a cluster of 16 nodes, each with a 33MHz PCI bus, 1000MHz Pentium III machines, 512MB of Main memory and Linux version 2.2.17. The machines are connected using a GigaNet 5300 switch.

The timings were obtained by running 5000 iterations of AllReduce and taking the average of the iterations over all nodes. The operation used is MPI_SUM, the datatype MPI_INT of size 4 bytes and the count of the elements was varied from 1 (4 bytes) to 1024 (4096 bytes)

6.5.1 Binomial message passing vs Degree-1 tree-based RDMA AllReduce Scheme

The mvich-1.0 implementation of the MPI standard uses the binomial algorithm to implement the Reduce part of the AllReduce operation in message passing. The binomial algorithm happens to be the same as the Degree-1 tree-based RDMA AllReduce scheme. Both algorithms take $log(N)$ steps for the Reduce operation for a cluster of size N. In each step only one node sends the data to any one destination. The results of the comparison are shown in Figure 6.17. The results are taken for a cluster of 16 nodes. As seen in the Figure, the Degree-1 tree-based RDMA AllReduce scheme shows around 20.73% benefit for smaller messages of 4 bytes and around 9.32% for larger messages.

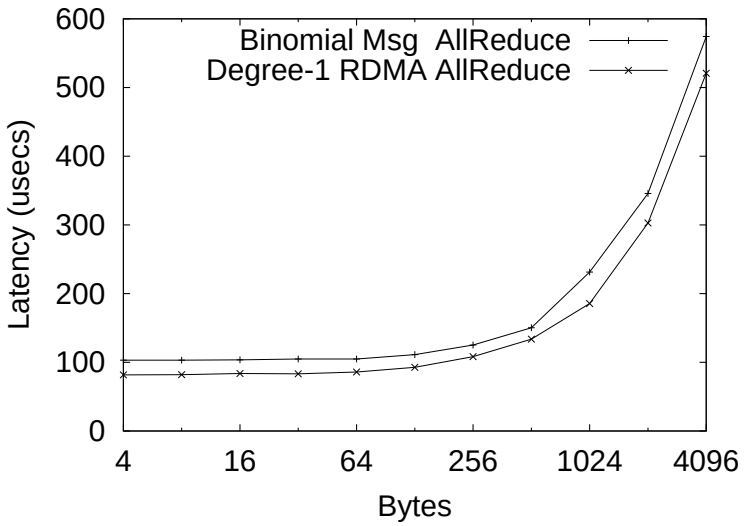


Figure 6.17: Comparison between Current message passing AllReduce and Degree-1 tree-based RDMA AllReduce scheme

6.5.2 Degree-k tree-based RDMA AllReduce Algorithms - Actual Performance

We have various Degree-k tree-based RDMA AllReduce algorithm for a cluster of *power of 2* size.

For a 16 nodes cluster, we implemented Degree-1, Degree-3, Degree-7 and Degree-15 tree-based RDMA AllReduce schemes. Some of these algorithms perform better than others for different message sizes. For example, in Figure 6.18, we see the timings for 16 nodes with byte size ranging from 4 (1 count element in AllReduce) to 512 (128 count element in AllReduce). The timings are taken for *MPI_AllReduce()* operation which has the Degree-k tree-based RDMA AllReduce Algorithm followed by the RDMA Broadcast algorithm.

In Figure 6.19, we see the timings for 16 nodes with byte size ranging from 512 (128 integer element in AllReduce) to 4096 (1024 integer element in AllReduce). From both the graphs, we see that Degree-3 tree-based RDMA AllReduce performs better than the other algorithms for message size up-to 1024 bytes.

In the latter graph, we observe that Degree-1 tree-based RDMA AllReduce starts winning over the other algorithms for larger messages (above 1K to 5K). This is because in the Degree-3 tree-based RDMA AllReduce case, 3 nodes write to 1 node and 3 operations are done at the receiving node. As the data size increases, the number of operations increase and computation becomes very expensive and hence Degree-1 tree-based RDMA AllReduce fares better because in Degree-1 tree-based RDMA AllReduce, the operations are distributed to a greater section of the nodes. Degree-15 tree-based AllReduce shows poor performance especially for larger messages because

of the cost of doing all the operations at a single node and because all nodes are sending messages to a single node, thereby causing contention at the switch.

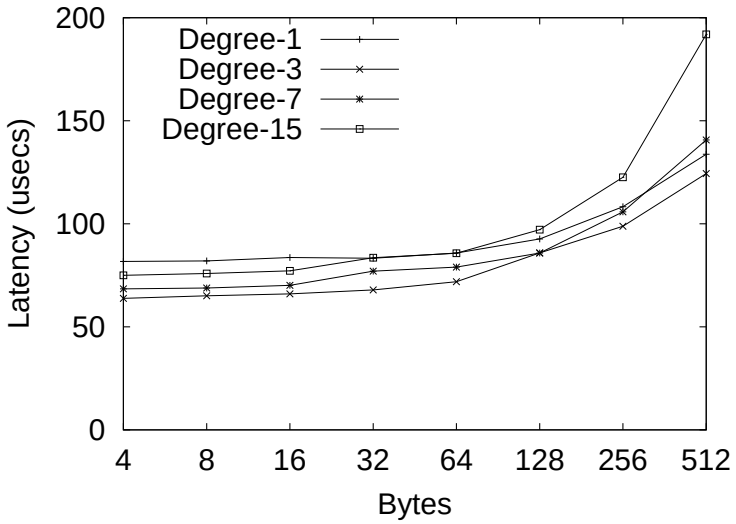


Figure 6.18: Degree-k tree-based RDMA AllReduce Performance comparison for a 16 node cluster for message size (4 to 512 bytes)

For an 8 node cluster, we have Degree-1, Degree-3 and Degree-7 tree-based RDMA AllReduce schemes. In Figure 6.20, we see that Degree-7 tree-based RDMA AllReduce performs the best for smaller messages till 256 bytes, but as the message size increases above 512 bytes, Degree-3 tree-based RDMA AllReduce starts performing better. But as shown in Figure 6.21 for larger messages beyond 1024 bytes, the Degree-1 tree-based RDMA AllReduce scheme proves to be the best. The Degree-7 tree-based RDMA AllReduce does not give this good performance in a 16 node cluster for smaller bytes because we need an extra step in a 16 node cluster which adds to the latency of the AllReduce operation.

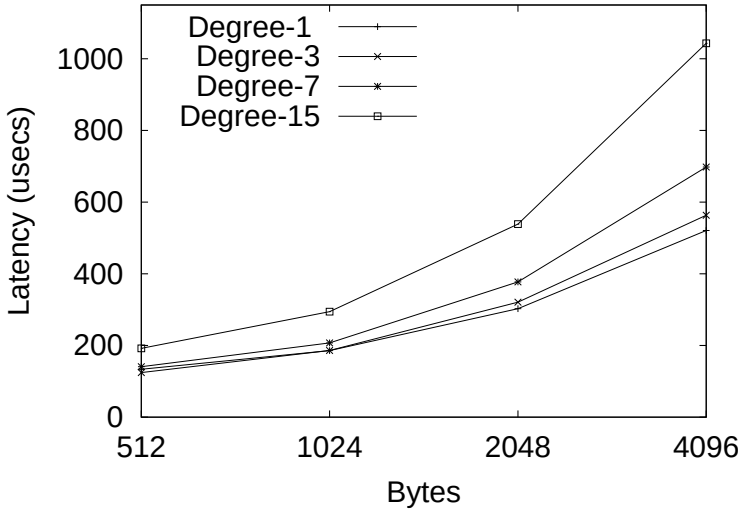


Figure 6.19: Degree-k tree-based RDMA AllReduce Performance comparison for a 16 node cluster for message size (1024 to 4096 bytes)

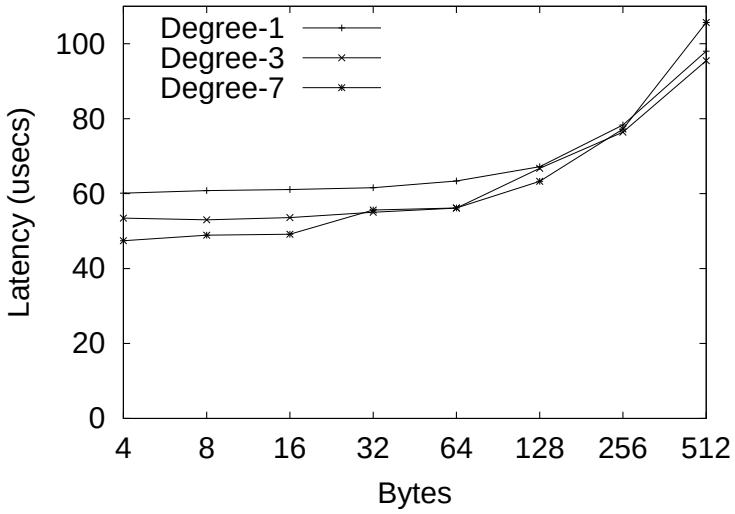


Figure 6.20: Degree-k tree-based RDMA AllReduce Performance comparison for a 8 node cluster for message size (4 to 512 bytes)

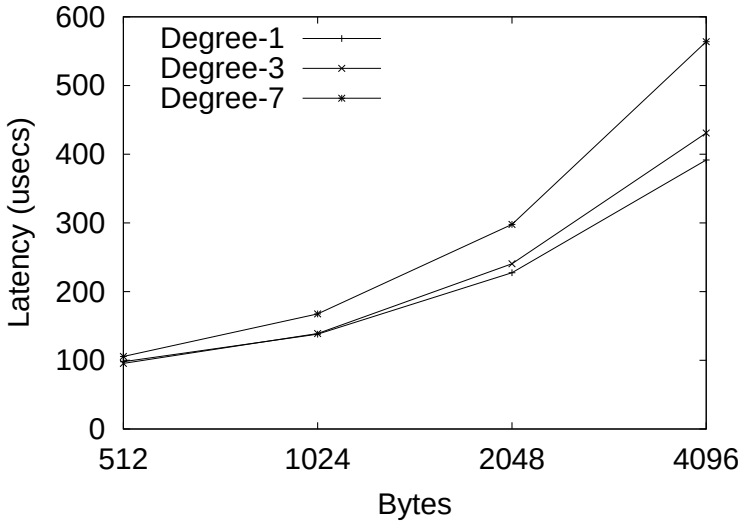


Figure 6.21: Degree-k tree-based RDMA AllReduce Performance comparison for a 8 node cluster for message size (1024 to 4096 bytes)

6.5.3 The Degree-k tree-based RDMA AllReduce Analytical model

In this section, we present the results of the Analytical model presented in Figure 6.16 in previous subsection. We show the best and the worst estimated timings for a 16 node cluster for messages from 4 bytes to 4096 bytes. We also compare these timings with the actual ones obtained.

The results for Degree-1, Degree-3, Degree-7 and Degree-15 are presented in this section. The following values have been given to the aforementioned parameters which are a part of the analytical model

1. $T\_transmit\_per\_bytes = 0.010\ us$
2. $T\_startup = 2us$
3. $T\_copy\_rate = 0.027us$

4. *T_nic = 1.52us*

5. *T_descriptor = 0.6us*

6. *T_mpi = 1.3us*

T_transmit and *T_operation* are calculated depending on the total number of *bytes* and *count* of the operation.

Figure 6.22 and 6.23 show the worst and best analytical timings and the actual practical timings for Degree-15 tree-based RDMA AllReduce for 16nodes with smaller and higher data size.

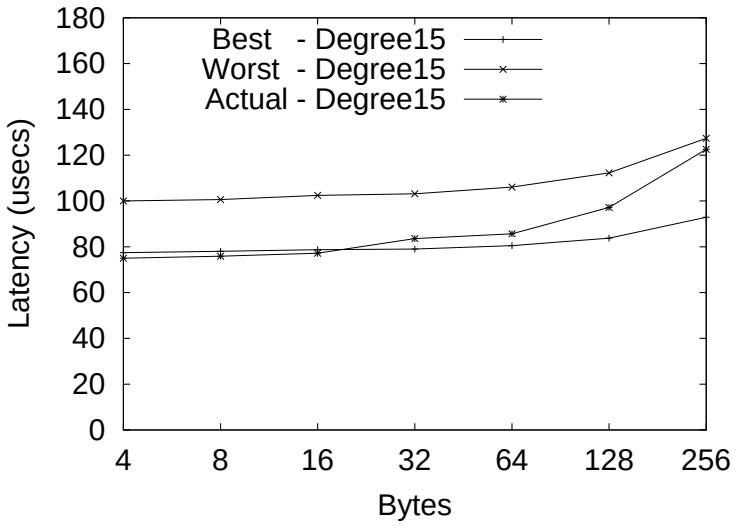


Figure 6.22: Degree-15 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(4-256 bytes) bytes

We see that for all data size the actual values lie between the best and the worst case estimated value. The greatest error rate is around 7.8% for 512 bytes. All other error rates, if any, lie below 2%.

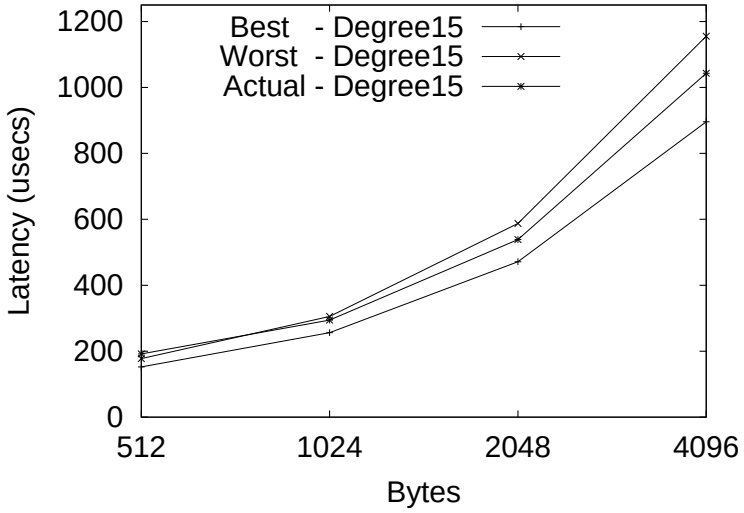


Figure 6.23: Degree-15 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(512-4096) bytes

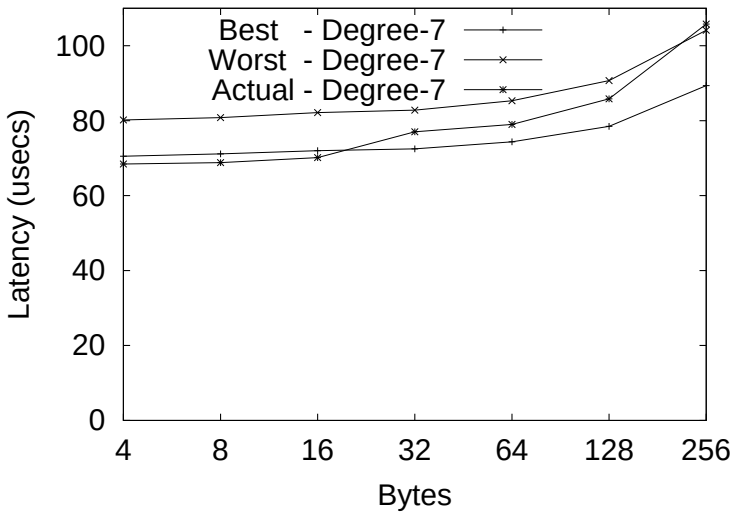


Figure 6.24: Degree-7 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(4-256 bytes) bytes

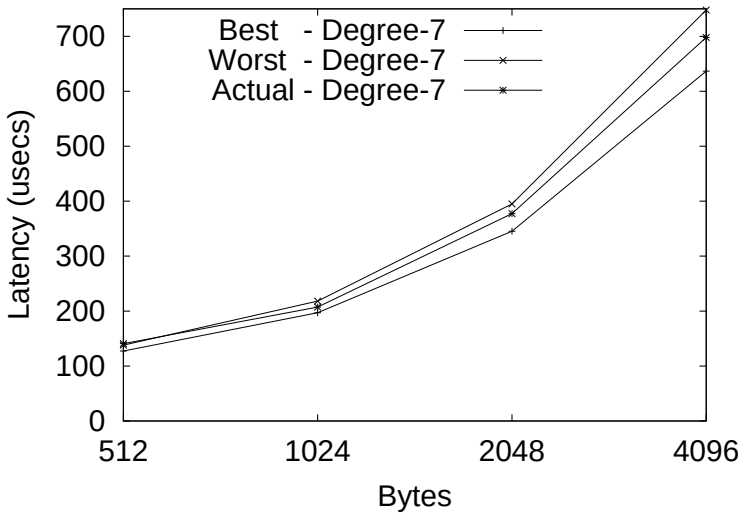


Figure 6.25: Degree-7 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(512-4096) bytes

Figures 6.24 and 6.25 show the timings for the same configuration but now with a Degree-7 tree-based RDMA AllReduce. We see a slight deviation of the actual values from the best and the worst cases, for small data size. However, the error rate of deviation from estimation is still below 3%.

Figures 6.26, 6.27, 6.28 and 6.29 show the timings for Degree-3 tree-based RDMA AllReduce small and higher values of bytes and Degree-1 tree-based RDMA AllReduce small and higher values of bytes.

The maximum error rate for Degree-3 tree-based RDMA AllReduce is about 4.3% for 4 bytes and around 6.21% for Degree-1 tree-based RDMA AllReduce again for 4 bytes.

The analytical results are thus found to give a rough estimate about the optimal algorithm. Thus for 16 nodes, for small data size like 4 bytes, the analytical model

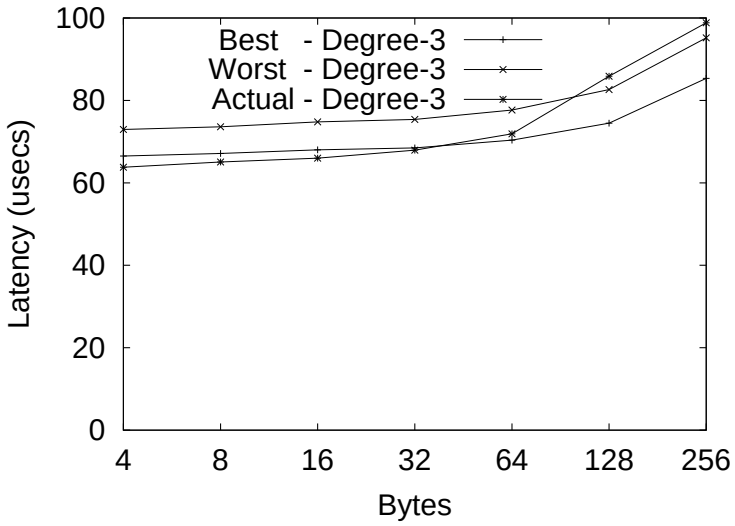


Figure 6.26: Degree-3 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(4-256bytes) bytes

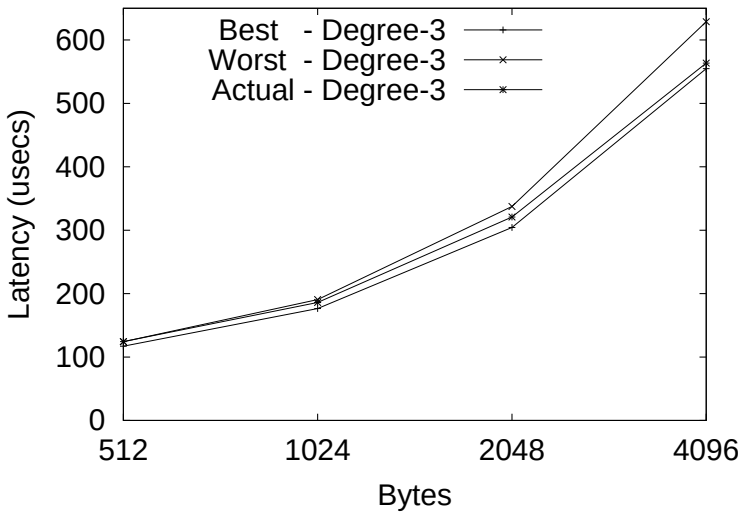


Figure 6.27: Degree-3 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(512-4096) bytes

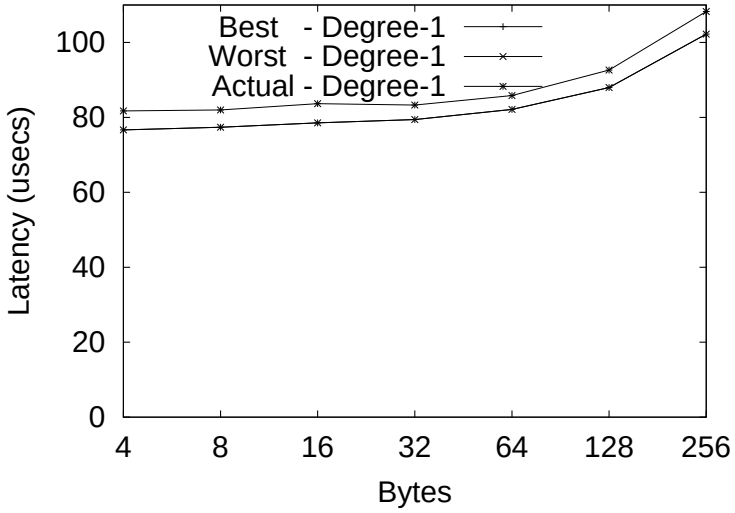


Figure 6.28: Degree-1 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(4-256 bytes) bytes

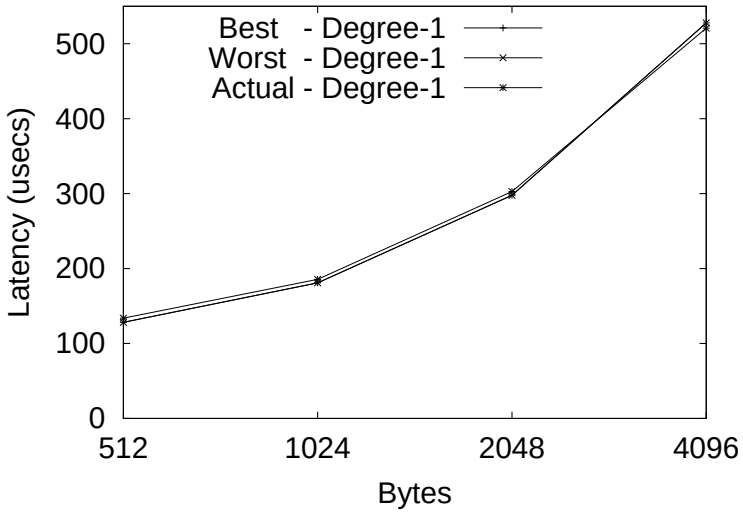


Figure 6.29: Degree-1 tree-based RDMA AllReduce Analytical and Practical comparison for smaller bytes(512-4096) bytes

gives Degree-4 tree-based RDMA AllReduce scheme as the optimal algorithm which we verified practically. For larger messages, the analytical model clearly shows Degree-1 tree-based RDMA AllReduce to be the most optimal one

On the basis of the results obtained, we can give a rough estimate on the optimal algorithms that can be chosen for various configuration and data sizes.

We summarize the results in Figure 6.30. For a cluster of 4 nodes, the Degree-3 tree-based RDMA AllReduce algorithm performs well for smaller messages till 1024 bytes. The Degree-3 tree-based RDMA AllReduce gives good performance for smaller messages in the 16 node and also in the 8 node cluster. But, in the 8 node cluster, we see that we obtain a slightly improved performance if we use Degree-7 tree-based RDMA AllReduce for very small messages. For larger messages above 1024 bytes, the Degree-1 tree-based RDMA AllReduce always gives the best performance.

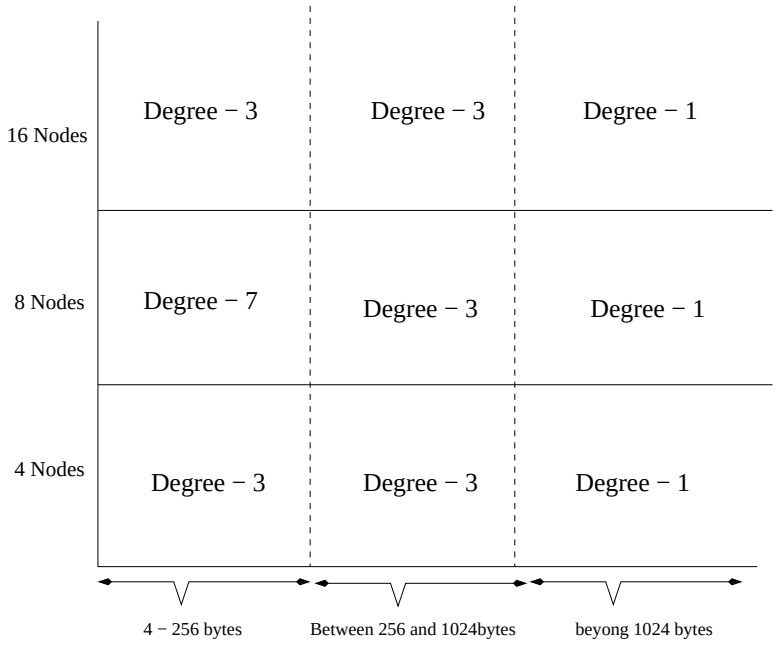


Figure 6.30: Choosing the Optimal algorithm for varying configuration

Thus we can generalize by saying that a Degree-3 tree-based RDMA AllReduce algorithm should give good performance for smaller data size (from 4 to 1024 bytes) and a Degree-1 tree-based RDMA AllReduce scheme can be used for larger data sizes while implementing the Reduce part of the AllReduce collective operation.

6.5.4 Degree-k tree-based message passing AllReduce Algorithms

In one of the previous sections we showed a comparison of the various Degree-k tree-based RDMA AllReduce Algorithms for a 16 node and an 8 node cluster. In this section, we show a comparison of the various Degree-k tree-based message passing AllReduce algorithms. The Degree-k algorithms basically remain the same, but instead of RDMA write, we now communicate through send and receive primitives. Thus in a Degree-3 tree-based message passing AllReduce scheme, three nodes send the data to a single node using send primitives and one node receives the data from all the three by using separate receive primitives.

Figure 6.31 and Figure 6.32 shows the comparison for Degree-1, Degree-3, Degree-7 and Degree-15 tree-based message passing AllReduce algorithms for data size ranging from 4-512 bytes and 512-4096 bytes respectively for a 16 node cluster. We see that degree-3 tree-based scheme performs the best from 4 to 512 bytes. Beyond 512 bytes the degree-1 scheme performs most optimally.

The current MVICH library uses Degree-1 tree-based scheme (Binomial algorithm) for message passing for implementing broadcast. Hence, the Degree-3 tree-based message passing scheme shows better performance than the current MVICH implementation.

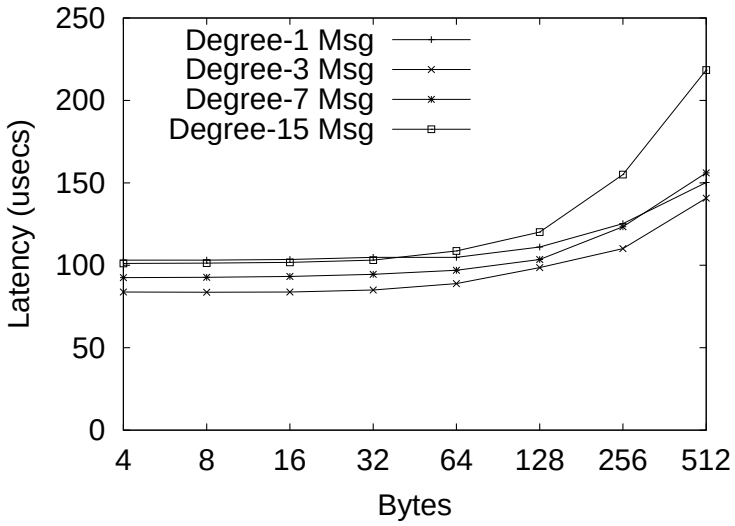


Figure 6.31: Comparison between various Degree-k tree-based message passing AllReduce schemes in a 16 node cluster (4-512 bytes)

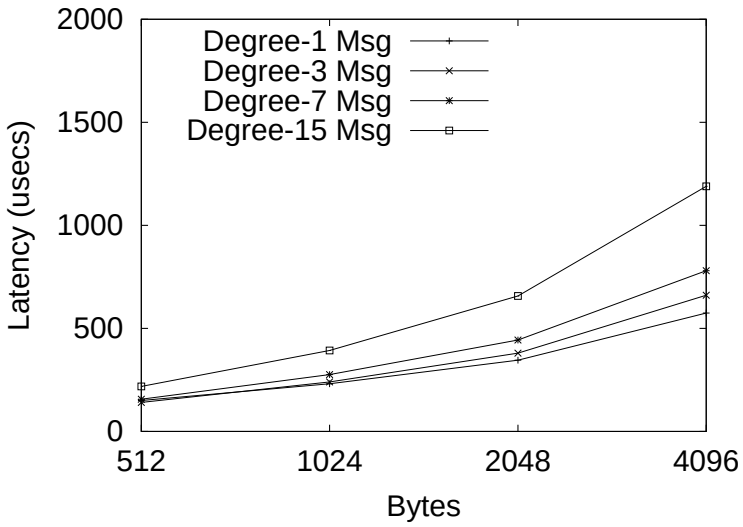


Figure 6.32: Comparison between various Degree-k tree-based message passing AllReduce schemes in a 16 node cluster (512-4096 bytes)

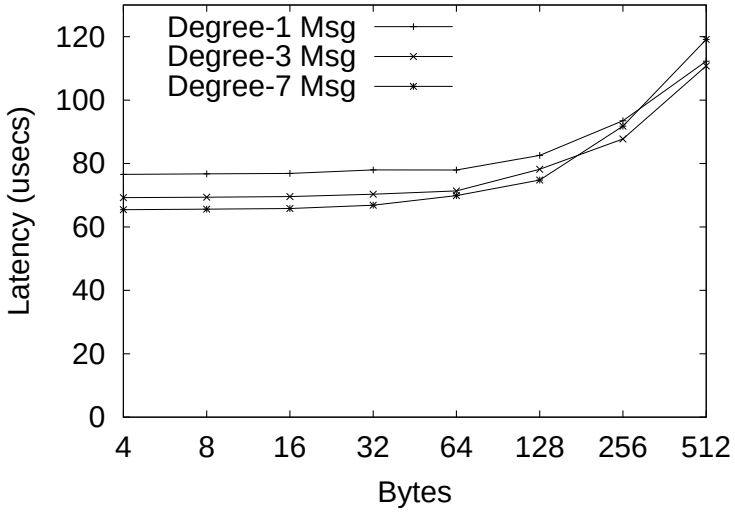


Figure 6.33: Comparison between various Degree-k tree-based message passing AllReduce schemes in a 8 node cluster (4-512 bytes)

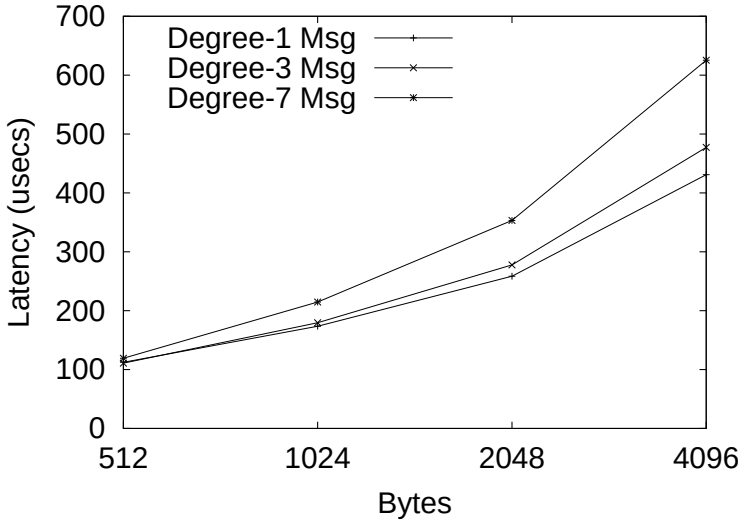


Figure 6.34: Comparison between various Degree-k tree-based message passing AllReduce schemes in a 8 node cluster (512-4096 bytes)

Figure 6.33 and Figure 6.34 shows the comparison for Degree-1, Degree-3 and Degree-7 tree-based message passing AllReduce algorithms for data size ranging from 4-512 bytes and 512-4096 bytes respectively for a 8 node cluster. We see that for a 8 node cluster, the Degree-7 performs most optimally from 4-128 bytes. For a data size between 128 and 512 bytes, Degree-3 performs the best and for larger data sizes beyond 512 bytes, Degree-1 performs most optimally.

6.5.5 Optimal Degree-k tree-based RDMA AllReduce vs Optimal Degree-k tree-based message passing AllReduce

The previous section showed a comparison of the various Degree-k tree-based message passing AllReduce schemes. In this section, we compare the most optimal Degree-k tree-based message passing AllReduce with the most optimal Degree-k tree-based RDMA AllReduce.

Figure 6.35 shows the comparison of the most optimal Degree-k tree-based RDMA AllReduce with the most optimal Degree-k message passing AllReduce for a 16 node cluster.

For 16 node cluster with RDMA AllReduce, Degree-3 tree-based RDMA AllReduce performs the best for data size till 1K and Degree-1 tree-based RDMA AllReduce gives the best performance for data size above 1K.

However, for 16 nodes with message passing AllReduce, Degree-3 tree-based message passing AllReduce scheme gives good performance till 512 bytes. Beyond 512 bytes, Degree-1 tree-based message passing AllReduce gives good performance.

We see in Figure 6.35 that the Degree-k tree-based RDMA AllReduce scheme performs better than the Degree-k tree-based message passing AllReduce scheme for 16 nodes. We see a 23.8% benefit for smaller messages of 4 bytes, when we use

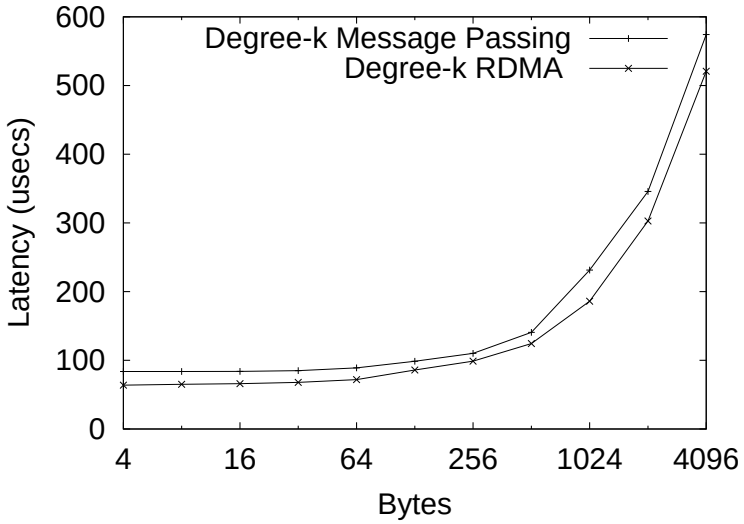


Figure 6.35: Comparison between optimal Degree-k tree-based RDMA and optimal Degree-k tree-based message passing AllReduce schemes in a 16 node cluster

the RDMA scheme. For larger data size, we see that Degree-1 tree-based RDMA AllReduce gives a 9% improvement over the Degree-1 tree-based message passing AllReduce. This benefit obtained in RDMA AllReduce is entirely due to the RDMA mechanism.

For 8 nodes, in RDMA AllReduce, we saw from the previous sections that Degree-7 performs the best for small data size from 4-128 bytes, between 128-1024 bytes Degree-3 performs the best and beyond 1K its Degree-1 RDMA AllReduce which performs the best. For Degree-k message passing AllReduce, we saw in the previous section that the trend almost remains the same except that Degree-1 message passing AllReduce performs most optimally beyond 512 bytes.

The comparison for the most optimal Degree-k RDMA AllReduce and Degree-k RDMA message passing AllReduce is seen in Figure 6.36. We see that the RDMA

AllReduce always performs better than the message passing AllReduce. We see a 27.5% benefit for 4 bytes and 9.13% benefit for 4096 bytes. This benefit is again attributed to the RDMA mechanism because of which we can eliminate the unnecessary memory copies.

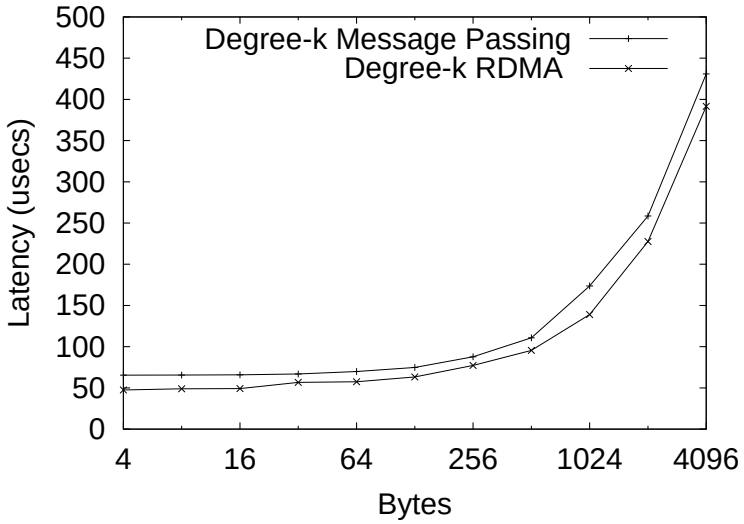


Figure 6.36: Comparison between optimal Degree-k tree-based RDMA and optimal Degree-k tree-based message passing AllReduce schemes in a 8 node cluster

6.5.6 Optimal Degree-k tree-based RDMA AllReduce vs Binomial message passing AllReduce

The comparison of our best Degree-k tree-based RDMA AllReduce algorithms with the message passing binomial algorithm is shown in Figure 6.37. We see that for a 16 node cluster, we obtain a 38.13% benefit for 4 byte messages, when we use the Degree-3 tree-based RDMA AllReduce. For larger messages of 4K, we get a 9.32% improvement on using the optimal Degree-1 tree-based RDMA AllReduce scheme.

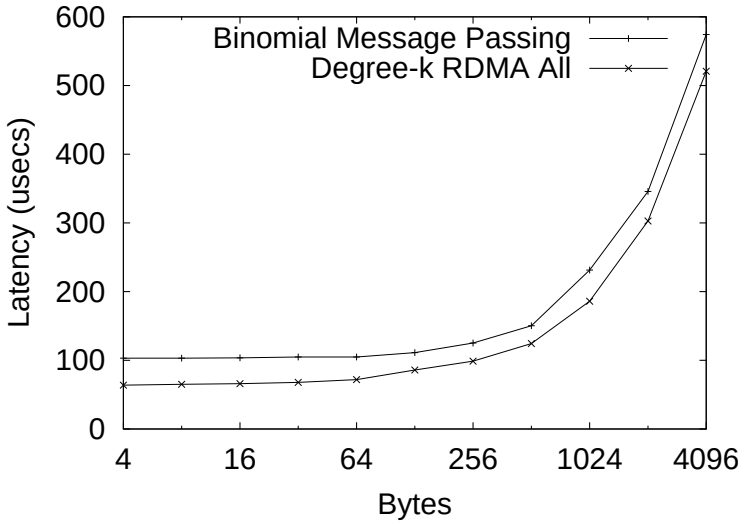


Figure 6.37: Comparison between binomial message passing and the most optimal Degree-k tree-based RDMA AllReduce schemes for a 16 node cluster

The same trend is shown for the 8 node cluster in Figure 6.38. We obtain 38.06% for smaller 4 byte messages on using Degree-7 tree-based RDMA AllReduce and 9.1% for datasize of 4096.

The benefits obtained are due to an optimal algorithm implemented with an efficient RDMA mechanism.

6.6 Summary

In this chapter, we introduced a scheme called *Degree-k tree-based AllReduce* for an efficient RDMA AllReduce implementation. We evaluated the best RDMA AllReduce implementations against the current binomial message algorithms and got performance benefit of 38.13% for 4 byte message for a 16 node cluster and around 9.32% benefit for a 4K bytes for 16 node cluster. To ensure fairness, we implemented the

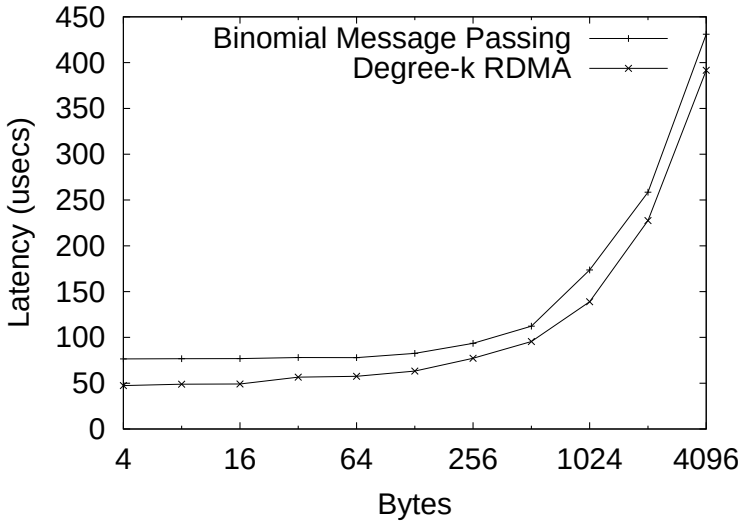


Figure 6.38: Comparison between binomial message passing and the most optimal Degree-k tree-based RDMA AllReduce schemes for a 8 node cluster

Degree-k tree-based RDMA All Reduce algorithms using message passing. The optimal Degree-k tree-based message passing AllReduce schemes were compared to the most optimal Degree-k tree-based RDMA AllReduce algorithms. We saw a performance benefit of 23.8% for smaller datasize of about 4 bytes and around 9% for data size of 4096 bytes for 16 nodes. This proves that the benefits obtained in our RDMA AllReduce scheme are not only because of the new algorithms but also because of the RDMA scheme. We also developed and presented an analytical model to obtain the AllReduce latencies for various Degree-k tree-based RDMA AllReduce algorithms, so that to enable the developer to choose the most optimal algorithm for its cluster configuration.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we introduced a novel method to implement the collective communication operations. We used the Remote Memory Write concept, to give an illusion of shared address space to implement an RDMA based collective communication library on VIA based clusters.

We discussed the design issues that would be the basis of such an implementation. We discussed the buffer management issues which included buffer registration, initialization and safe reusing. We discussed techniques for the receiver node to verify data validity for data intensive operations. Collective communication operations fall in 3 categories namely synchronization, data distribution and global reduction. We have implemented the *Barrier*, *Broadcast*, *AllReduce* each belonging to one of the aforementioned categories.

First, we provided a *RDMA barrier* as an alternative to the message passing barrier. We explained the design issues and buffer management details that would be a part of RDMA implementation. The barrier represented the simplest of all the collective operations because it did not involve any data transfer. The RDMA barrier, implemented using pair-wise exchange with recursive doubling gave a 30% benefit on a 16 node cluster as compared to the current message passing barrier.

Next, we implemented *RDMA Broadcast*, a data distribution operation. Here, we explored the various mechanisms that can be used to validate the receiver data at the destination node. We explored issues related to buffer registration, buffer address exchange and buffer reuse, all of which will play an important part in any RDMA implementation of a data intensive collective operation. The RDMA binomial broadcast gave a 14.4% benefit for 16 nodes for 4608 bytes as compared to the binomial message passing broadcast algorithm

Lastly, we implemented the *AllReduce* operation. We introduced a new concept of *degree-k tree based* AllReduce algorithms which when combined with the RDMA mechanism give improved performance as compared to the message passing algorithms based on the point-to-point communication model. We also presented an analytical model that gave an estimation of the most optimal *degree-k tree based* AllReduce algorithm that can be used for a given cluster and data size. The results obtained by the analytical model had an error rate below 8% and the results estimated by it matched the practical ones obtained. A comparison of the most optimal degree-k tree-based RDMA AllReduce with the message passing binomial AllReduce gave us a benefit of about 38.13% benefit for a small data size of 4 bytes and about 9.32% for data size of 4K bytes for a 16 node cluster. We also compared the most optimal degree-k tree-based RDMA AllReduce with the most optimal degree-k tree-based message passing AllReduce to emphasize on the contribution of the the RDMA mechanism in the above results.

We have presented the basic issues in designing a RDMA based collective communication library and demonstrated its effectiveness with prototype implementations for frequently used collective communication operations. However, before such a

library can be practically implemented, in-depth analysis of the global buffer management for all collective operations needs to be done. Efficient algorithms dealing with user-defined communicators used in conjunction with these collective operations need to be explored. We have worked with the RDMA write feature of VIA, however various protocols provide the RDMA read feature which can be exploited to support collective operations even better. Our Degree-3 tree-based message passing scheme shows better performance for smaller bytes than the current MVICH binomial message passing broadcast algorithm. Thus, an analysis can to be made in order to check the feasibility and advantages of having such dual algorithms for different data sizes for the message passing broadcast operation in the MVICH implementation.

All the RDMA based algorithms in this thesis are written at a portable MPI level. MPI-1.0 standard provided no primitives for *Get* and *Put* operations. However, MPI-2.0 does provide these primitives and it will be interesting to explore how RDMA collective operations can exploit these primitives and their implementations.

The work done in this thesis discussed the RDMA collective operations on a cluster of uniprocessor nodes. However, work has been done in collaboration with *The Pacific Northwest National Laboratories* to extend this concept to a cluster of SMP nodes. Combining the RDMA write scheme with a scheme to exploit shared memory in an SMP, we have developed a fast barrier algorithm for a cluster of SMP nodes [12]. The results obtained by the fast barrier are encouraging and more work is being done in the direction of extending the concepts to other collective operations

Lastly, we have worked with only three collective operation. An effort has to be made to provide RDMA support for all the other collective operations, as defined by the MPI Standard.

BIBLIOGRAPHY

[1] Infiniband Trade Association. http://www.infinibandta.org.

[2] M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP switch-connected NT clusters. In *the Proceedings of the International Parallel and Distributed Processing Symposium, pages 33-42*, May 2000.

[3] 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.

[4] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-Based Barrier over Myrinet/GM. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Apr 2001.

[5] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance Benefits of NIC-Based Barrier on Myrinet/GM. In *Workshop on Communication Architecture for Clusters*, 2001.

[6] P. Buonadonna, A. Geweke, and D. E. Culler. BVIA: An Implementation and Analysis of Virtual Interface Architecture. In *the Proceedings of Supercomputing '98*, 1998.

[7] Myricom Corporations. The GM Message Passing Systems.

[8] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kaufmann, March 1998.

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

[10] W. Gropp and E. Lusk. MPICH Working Note: The Second Generation ADI for the MPICH Implementation of MPI.

[11] Future Technology Group. MVICH: MPI for Virtual Interface Architecture. In *http://www.nersc.gov/research/FTG/mvich*.

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

[13] http://www.viarch.org/. Virtual Interface Architecture Specifications.

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

[15] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via.

[16] P. K. McKinley, Y.-J. Tsai, and D.F.Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers. In *Technical Report MSU-CPS-94-35, Michigan State University*, 1994.

[17] Lee Melatti. Fast Ethernet: 100 Mbit/s Made Easy. *Data Communications on the Web*, Nov. http://www.data.com/tutorials/100mbits-made-easy.html.

[18] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[19] Message Passing Interface Forum. *MPI Version 1.1: A Message-Passing Interface Standard*, June 1995.

[20] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, May 1998.

[21] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[22] J. Nieplocha, RJ Harrison, and RJ Littlefield. Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, pages 197–220.

[23] S. Pakin, M. Lauria, and A. Chein. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of SC*, 1995.

[24] 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.

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

[26] P. Shivam, P. Wyckoff, and D. Panda. Can User Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of International Parallel and Distributed Processing Symposium '02*, April 2002.

[27] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *Proceedings of the International Conference on Supercomputing*, June 1999.

[28] Robert van de Geijn, David Payne, Lance Shuler, and Jerrell Watts. *A Streetguide to Collective Communication and its Application.* Jan 1996.

[29] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for Parallel and Distributed Computing. In *the Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.