SCALABLE AND HIGH PERFORMANCE COLLECTIVE COMMUNICATION FOR NEXT GENERATION MULTICORE INFINIBAND CLUSTERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Amith Rajith Mamidala, B. Tech, M. S.

* * * * *

The Ohio State University

2008

ABSTRACT

High Performance Computing is enabling rapid innovations spanning several key areas ranging from science, technology and manufacturing disciplines to entertainment and financial markets. One computing paradigm contributing significantly to the outreach of such capabilities is *Cluster Computing*. Cluster computing involves the use of multiple Commodity PCs interconnected by a network to provide the required computational resource in a cost-effective manner. Recently, commodity clusters are rapidly transforming into capability class machines with several of them featuring in the Top 10 list of supercomputers. The two primary drivers for this trend being: a) Advent of Multicore technology and b) Performance and Scalability of InfiniBand, an open standard based interconnection network. These two factors are ushering in an era of ultra-scale InfiniBand Multicore clusters comprising of tens of thousands of compute cores.

Utilizing Message Passing Interface (MPI) is the most popular method of programming parallel appplications. In this model, communication occurs via explicit exchange of data messages. MPI provides for plethora of communication primitives out of which *Collective* primtives are especially significant. These are extensively used in a variety of scientific and engineering applications (such as to compute fast fourier transforms and multiply large matrices, etc.). It is imperative that these collectives be designed efficiently to ensure good performance and scalability. MPI collectives pose several challenges and requirements in terms of guaranteeing data reliability, enabling efficient scalable means of data transfers and providing for process skew tolerance mechanisms. Moreover, the characteristics of underlying network and multicore systems directly impact the behavior of the collective operations and need to be taken into consideration for optimizing performance and resource usage.

In this dissertation, we take on these challenges to design a *Scalable and High Performance Collective Communication* subsystem for MPI over InfiniBand Multicore clusters. The central theme used in our approach is to have an in-depth understanding of the capabilities of underlying network/system architecture and leverage these to provide optimal design alternatives. Specifically, the dissertation describes novel communication protocols and algorithms utilizing a) InfiniBand's hardware Multicast, RDMA capabilities and b) System's shared memory to meet the stated requirements and challenges. Also, the collective optimizations discussed in the dissertation take into account the different transport methods of InfiniBand and the architectural attributes of Multicore systems. The designs proposed in the dissertation have been incorporated into the open source MVAPICH software used by more than 680 organizations worldwide. It is used in several cluster installations, and currently used by the world's third fastest supercomputer.

Dedicated to Amma, Baba and Nandu

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. Dhabaleswar K. Panda for his guidance throughout my PhD study. I am grateful for his tremendous effort, friendship and counsel through the course of my PhD. Especially, I would like to thank him for his invaluable advice and support during many tough periods at OSU.

I would like to thank my committee members Prof. P. Sadayappan, Prof. Feng Qin and Prof. Srini Parthasarathy for their help and suggestions.

I am grateful to Ohio State University, National Science Foundation (NSF) and Department of Energy (DOE) for the financial support.

I would also like to thank Dr. Rajeev Thakur and Dr. Bill Gropp for their advice during my internship at Argonne National Labs.

I am thankful to Dr. Jiuxing Liu who was my mentor during the first and second years of my PhD. I am grateful to Dr. Hyun-Wook Jin for his effort and advice during my study. I would also like to thank my seniors Dr. Jiesheng Wu, Dr. Pavan Balaji, Dr. Darius Buntinas and Dr. Weikuan Yu for their helpful comments and suggestions. I am fortunate for having worked closely with Abhinav Vishnu, Sundeep Narravula, Gopal Santhanaraman, Rahul Kumar, Karthikeyan Vaidyanathan, Sayantan Sur, Lei Chai, Wei Huang, Matthew Koop, Ranjit Noronha, Savitha Krishnamurthy, Sushmitha Kini, Uday Bondhugula, Bala Chandrasekaran, Qi Gao, Ping Lai, Prachi Gupta, Karthik and Jaidev.

I would like to thank all my friends, Sandy, Mallu, Gopal, Devi, Naps, Naveen, Boondi and Vishnu for making my stay at Columbus a wonderful experience.

Finally, I am thankful to my parents and brother for their love, support and friendship.

VITA

June 18, 1980	Born - Secunderabad, India.
July, 1998 - June, 2002	B.Tech, Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India
September, 2002 - May, 2003	Graduate Teaching Associate, The Ohio State University.
June, 2003 - May, 2006	Graduate Research Associate, The Ohio State University.
June, 2006 - September, 2006	Summer Intern, Argonne National Labs, Argonne, IL
September, 2006 - May 2008	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Jiuxing Liu, Amith R Mamidala and Dhabaleswar K Panda. "Performance Evaluation of InfiniBand with PCI-Express", IEEE Micro Journal, 2005

Abhinav Vishnu, Matthew Koop, Adam Moody, Amith R Mamidala, Sundeep Narravula and Dhabaleswar K Panda. "Topology Agnostic Hot-Spot Avoidance with InfiniBand", Concurrency and Computation: Practice and Experience

Rahul Kumar, Amith R Mamidala and Dhabaleswar K Panda. "Scaling Alltoall Collective on Multicore Systems", Workshop on Communication Architecture for Clusters, IPDPS 2008

Amith R Mamidala, Rahul Kumar, Debraj De and Dhabaleswar K Panda. "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics", Int'l Symposium on the Cluster Computing and the Grid (CCGrid), May 2008

Rahul Kumar, Amith R Mamidala, Matt Koop, Gopal K Santhanaraman and Dhabaleswar K Panda. "Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication", EuroPVM/MPI 2008

Amith R Mamidala, Sundeep Narravula, Abhinav Vishnu, Gopal K Santhanaraman and Dhabaleswar K Panda. "Using Connection-Oriented and Connection-Less transport on Performance and Scalability of Collective and One-sided operations: Trade-offs and Impact", Accepted at ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 07

Gopal K Santhanaraman, Sundeep Narravula, Amith R Mamidala, and Dhabaleswar K Panda. "MPI-2 One Sided Usage and Implementation for Read Modify Write operations: A case study with HPCC", In Proceedings of EuroPVM/MPI '07, Paris, France

Sundeep Narravula, Amith R Mamidala, Abhinav Vishnu, Karthik Vaidyanathan and Dhabaleswar K Panda. "High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations", Int'l Symposium on Cluster Computing and the Grid (CCGrid), Rio de Janeiro - Brazil, May 2007

Sundeep Narravula, Amith R Mamidala, Abhinav Vishnu and Dhabaleswar K Panda. "High Performance MPI over iWARP: Early Experiences", International Conference for Parallel Processing, ICPP 2007

Abhinav Vishnu, Matt Koop, Adam Moody, Amith R Mamidala, Sundeep Narravula and Dhabaleswar K Panda. "Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective", Int'l Symposium on Cluster Computing and the Grid (CCGrid), Rio de Janeiro - Brazil, May 2007

Abhinav Vishnu, Amith R Mamidala, Sundeep Narravula and Dhabaleswar K Panda "Automatic Path Migration over InfiniBand: Early Experiences", Third International Workshop on System Management Techniques, Processes, and Services, to be held in conjunction with IPDPS '07, March 2007

Abhinav Vishnu, Prachi Gupta, Amith R Mamidala, and Dhabaleswar K Panda. "A Software Based Approach for Providing Network Fault Tolerance in Clusters with uDAPL interface: MPI Level Design and Performance Evaluation", Supercomputing, SC 06

Amith R Mamidala, Abhinav Vishnu and Dhabaleswar K Panda. “Shared Memory and RDMA based design for MPI_Allgather over InfiniBand”, EuroPVM/MPI Conference, 2006

Amith R Mamidala, Lei Chai, Hyun-Wook Jin and Dhabaleswar K Panda. “Efficient SMP-Aware MPI-Level Broadcast over InfiniBand’s Hardware Multicast”, Workshop on Communication Architecture for Clusters, IPDPS 2006

Jiuxing Liu, Amith R Mamidala and Dhabaleswar K Panda. “Performance Evaluation of InfiniBand with PCI-Express”, IEEE Micro, 2005

Amith R Mamidala, Hyun-Wook Jin and Dhabaleswar K Panda. “Efficient Hardware Multicast Group Management for Multiple MPI Communicators over InfiniBand”, EuroPVM/MPI Conference 2005

Sayantan Sur, Uday Bondhugula, Amith R Mamidala, Hyun-Wook Jin and Dhabaleswar K Panda. “High Performance RDMA Based All-to-all Broadcast for InfiniBand Clusters”, International Conference on High Performance Computing, HiPC 2005

Abhinav Vishnu, Amith R Mamidala, Hyun-Wook Jin and Dhabaleswar K Panda. “Performance Modeling of Subnet Management on Fat Tree InfiniBand Networks using OpenSM”, Workshop on System Management Tools for Large-Scale Parallel Systems, IPDPS 2005

Amith R Mamidala, Jiuxing Liu and Dhabaleswar K Panda. “Efficient Barrier and Allreduce on InfiniBand clusters using Hardware Multicast and Adaptive algorithms”, International Conference on Cluster Computing, Cluster 2004

Jiuxing Liu, Amith R Mamidala and Dhabaleswar K Panda. “Fast and Scalable MPI-Level Broadcast using InfiniBand’s Hardware Multicast Support”, International Parallel and Distributed Processing Symposium, IPDPS 2004

Jiuxing Liu, Amith R Mamidala and Dhabaleswar K Panda. “Performance Evaluation of InfiniBand with PCI-Express”, Hot Interconnects: Symposium on high performance interconnects, HOTI 2004

Jiesheng Wu, Amith R Mamidala and Dhabaleswar K Panda. “Can NIC memory in InfiniBand Benefit Communication Performance?: A Study with Mellanox Adapter”, Technical report, OSU-CISRC-4/04-TR20

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. Dhabaleswar K. Panda
Computer Networks	Prof. Dong Xuan
Software Systems	Prof. Srini Parthasarathy

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

High Performance Computing is rapidly making inroads into diverse disciplines from all walks of life. The availability of significant and cheap compute power is accelerating the frontiers in basic sciences, technology, manufacturing and even making profound impact in the entertainment, financial sectors. One of the major factors contributing to this phenomenon is the increasing use of high-end supercomputers built from commodity components. This paradigm of parallel computing, also known as Cluster Computing, has gained enormous popularity and usage. In fact, the Top 500 list of supercomputers feature hundreds of large scale clusters delivering massive amounts of computational power. The easy availability of low cost commodity PCs together with scalable and high performance interconnection networks is making Compute Clusters more affordable and cost-effective.

Figure 1.1(a) summarizes a typical cluster comprising of compute nodes interconnected by a network. As shown in the figure, each of the compute nodes can contain more than one processing elements. Interconnecting hundreds or thousands of these nodes is the cluster interconnect. There exists many approaches to program applications for these architectures. One of the most popular and successful programming model is to parallelize an application into several distinct

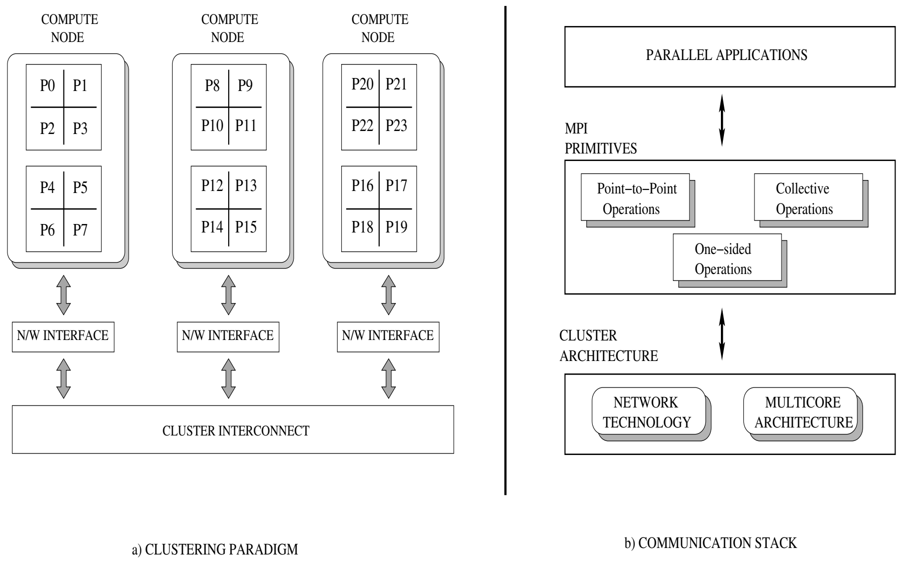


Figure 1.1: Cluster Architecture

processes with each of the processes potentially running on separate processing elements. These processes perform computation on their individual address spaces and explicitly exchange data by communicating via messages. The semantics of the communication are standardized and adopted as the well known Message Passing Interface or MPI. MPI provides an easy, powerful and portable abstraction for exchanging data between processes. As shown in Figure 1.1(b), MPI functions as a communication middleware hiding the details of the underlying network or system architecture. It is the critical component bridging the gap between the hardware and the communication requirements of the application. MPI exposes a plethora of communication calls to the application. In order to achieve high parallel speed-ups it is important that, firstly the applications choose these operations intelligently and secondly, these operations in turn be designed to scale to thousands of processes and deliver best performance.

Broadly, the MPI operations can be divided into three main classes: (a) Point-to-Point primitives (b) Collective primitives and (c) One-sided primitives. Out of these, the collective operations are especially significant as these provide simple and manageable methods of moving data between group of processes. Infact, these operations form the communication core of many application codes. As an example, FT and IS exclusively use MPI_AlltoAll personalized collective communication calls in their compute kernels. Further, certain operations like MPI_Reduce, MPI_Allreduce also permit global reductions on the data sets. The rationale for providing a separate set of collectives rather than using the available Point-to-Point primitives is to ensure high performance and scalability. This can be accomplished either by using optimal collective algorithms or by taking advantage of the underlying network features. For instance, in some of the high performance interconnects certain collective primitives are offloaded to the network hardware. One example of this would be hardware supported multicast operation where a data packet injected into the network is replicated and sent to multiple destinations. As discussed earlier, it is important that MPI designs leverage such primitives and efficiently transfer the raw capabilities of underlying network and system architectures to the application.

Recently, two popular trends have ushered in an era of peta-scale computing in commodity clusters. The first significant factor being the advent of Multicore architecture. Due to the physical limitations of increasing clock-rates, the recent trend of Multicore technology is to substitute a single high clocked processor with multiple low-frequency compute cores. There could be tens of these cores on a single node. For example, the Barcelona Multicore chip from AMD already has

16 cores on a single node and many more cores are planned to be added in their next generation processors. The other trend is the rapid proliferation of InfiniBand Architecture (IBA) [18]. Good scalability together with high performance has made IBA a popular choice of clustering thousands of processing cores. In addition to delivering low latency and high bandwidth, IBA provides a rich set of network primitives like Remote Direct Memory Operations (RDMA), hardware-level Multicast and Send/Shared-Receive Queue capabilities over different message transports. These two technology drivers are enabling peta-scale capable clusters comprising of tens of thousands of cores. Infact, the recent induction of Ranger [41] with 3936 nodes each having 16 cores (totaling 62,976 cores) exemplifies this phenomenon.

As discussed above, MPI plays a pivotal role in harnessing the capabilities of these ultra-scale clusters. As MPI provides the necessary communication interface to the applications, it becomes imperative that the MPI primitives be scalable and channel the hardware capabilities efficiently to the application while adding very low overheads. This is true especially with MPI Collective operations whose designs are intricately intertwined with underlying parallel architecture of the system. Moreover, Collective operations are realized using communication algorithms which need to take into account application requirements and characteristics such as load imbalances, communication behavior, etc.

The main objective of this thesis is to design a *"Scalable Collective Communication Subsystem in MPI for the Next-generation Multicore InfiniBand clusters"*. The central goals are to:

- Leverage IBA mechanisms to build efficient and scalable MPI collectives

- Understand the impact of multicore architecture on collectives and design communication algorithms accordingly

- Take into account application's characteristics and behavior to propose a) new collective algorithms and b) new network primitives

In the following chapters of this dissertation, we describe the detailed solutions to achieve these objectives. Specifically, Chapter 2 discusses the background and motivation of our research. We describe the salient issues and challenges in MPI collective communication followed by the trends in network and system architectures. The chapter also outlines the architecture of InfiniBand and its capabilities followed by a brief overview of Intel and AMD multicore platforms. Chapter 3 describes the problem studied in this dissertation. We provide the broad methodology for the research pursued in Chapter 4. Chapters 5 through 9 describe the detailed designs, experiments conducted and evaluations of the proposed designs. Chapter 10 provides an overview of the open source MVAPICH distribution. We conclude the dissertation with Chapter 11 summarizing the contribution of the research undertaken and the new research directions that can be pursued in the future.

CHAPTER 2

BACKGROUND AND MOTIVATION

In this chapter, we first provide an overview of MPI Collective Communication. We discuss the salient issues and requirements in designing collective operations in MPI, standard collective algorithms used and their limitations. We then describe the current trends in InfiniBand interconnect technology and Multicore architecture.

2.1 MPI Collective Communication

We begin this section by describing the semantics of the different collective operations in MPI.

2.1.1 Important Collective Operations in MPI

MPI allows for a variety of collective operations. The important collectives together with their operational semantics is explained below.

- *Allreduce:* This operation performs both computation and communication of data. In Allreduce, each process supplies a vector of data. This data can be of any type, e.g. an integer. Also, the data may be stored in non-contiguous chunks in memory. After the operation is complete, each process receives a result vector which is the reduction of the vectors from all the processes participating in the collective. This primitive is extensively used in many scientific applications like Parallel Ocean Modeling (POP) [22], etc.

- *Reduce:* The semantics of this operation are the same as that of Allreduce except that after the operation is complete, not all the processes receive the resulting vector. Only one process, called as the root of the collective, receives the final data.

- *Broadcast:* This is a data moving primitive. In this collective, the root process supplies a vector of data. After the operation is complete, all the processes in the collective receive this vector of data. Once again, the data can be of any type and non-contiguous. This is true for all the collectives explained in this section except Barrier synchronization. An efficient implementation of Broadcast can be used in all the major collectives such as Allreduce, Barrier, etc.

- *Alltoall:* This important collective is a personalized operation. In this operation, each process sends a distinct vector of data to each of the other processes. As the semantics state, this collective is a data intensive operation. The communication kernels of applications like FFT [28] use Alltoall exclusively.

- *Allgather:* Like Alltoall, this operation is also data intensive. However, this operation involves an alltoall broadcast of data. In this collective, each process supplies a vector of data which is broadcasted to everyone else. This is heavily used in matrix multiplication kernels.

- *Barrier:* Unlike all the other collectives described above, barrier is a synchronization operation. The semantics of this operation state that a process can complete a barrier only after all the other processes have invoked this operation.

2.1.2 Salient Issues and Requirements for MPI Collectives

MPI offers various collective routines with different functionality and communication patterns. As discussed earlier, efficient and scalable implementation of these collective operations is critical for achieving superior application performance. This raises several interesting challenges and requirements that need to be thoroughly understood and carefully evaluated for achieving the desired objective. In this chapter, we provide an overview of these issues. In subsequent chapters, we explain how to meet these requirements on modern multicore clusters using the popular interconnect, InfiniBand.

We now describe the salient issues and requirements for MPI Collectives on emerging multicore clusters:

a) Efficient Communication Protocols:

As explained earlier, MPI specifies different collective operations having distinct communication semantics and patterns. Further, MPI guarantees data reliability. In other words, after the MPI collective operation returns, the final data in the message buffers is correct and is from the same source that the application is expecting. Apart from efficient collective algorithms, the onus to deliver the data reliably with low overhead lies on the data transfer protocols. Most modern networks like Quadrics, Myrinet and recently InfiniBand, have advanced network primitives for *unicast* (one-to-one) and *multicast* (one-to-many) communication. Collective algorithms can utilize these primitives for obtaining the best performance. For most of these primitives, the data reliability is handled by the network stack. For example, checking data correctness by doing checksum calculations is handled by the network hardware. However, they may not provide guarantees in data delivery for some cases. This is true with some of the transports of InfiniBand. In such scenarios, the MPI middleware should guarantee the necessary data reliability via robust protocols.

In MPI, communication occurs within groups of processes called as *communicators*. Each *communicator* has a distinct context so that messages from one *communicator* don't interfere with the other. It is the responsibility of the MPI layer to manage these multiple *communicators* efficiently, especially for the collective functions. This management can potentially involve interactions with the network management modules. So, it is important that all these activities proceed in a non-obtrusive manner to the application.

b) Fast Data Transfer Methods:

Figure 2.1 illustrates the general methodology adopted in typical MPI designs for collective communication. The basic approach is to layer collectives over the existing point-to-point MPI primitives. In the figure two such blocks are shown, one for communication across the nodes and the other for communication within the node. The collective algorithm optimally chooses the point-to-point calls in order to make progress with the least communication steps and/or communication time. However, as shown in the figure, data copies are not avoided. For example, there are separate set of buffers (A and C) used by the point-to-point communication substrate differing from the set of buffers (B and D) used by the underlying data transfer layers.

A very desirable approach would be to avoid the extra set of buffers (A and C) used in the above scenario. Staging and copying the buffers is a costly operation potentially consuming CPU resources and system memory. In order to enable fast data movement in a collective operation these extraneous costs need to be avoided. These could be avoided in two possible ways: 1) Bypass the point-to-point layer and 2) Integrate the disjoint sets of the two point-to-point communication operations into a unified collective layer.

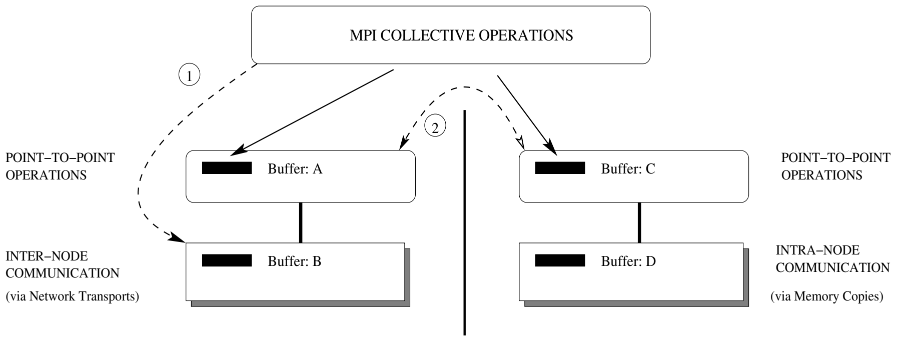


Figure 2.1: Typical MPI Collective Designs

c) Process Skew in Collective Communication:

A collective function is called by all the processes. The standard communication algorithms for collective which define the schedule of data transfers assume that all the processes arrive at the same time. For example, Figure 2.2(a) shows the schedule for barrier synchronization. The algorithm described uses two phases: *gather phase* followed by a *broadcast phase*. This approach has been well studied in literature [20] and used in different scenarios. As shown in the figure, there are three steps in this collective. In the first step, the leaf processes send a message to their parents. In the second step, the message is forwarded to the root. Once the root receives the messages from both its children, it releases all the processes. This is the final step in the collective. These algorithms work optimally if all the processes arrive very close to each other. Figure 2.2(b) elucidates the scenario if one of the leaf process arrives late. As seen from the Figure 2.2(c), it takes additional two steps before the message receives the root. The message from other process has been already forwarded to the root. Thus, the schedule mentioned above is not optimal when the processes are skewed in an application.

Process skew is common on large scale clusters and as shown in [33] little system noise can potentially alter the arrival patterns of the processes. Thus, for obtaining best performance, it is important that skew be factored into the design of the communication algorithm.

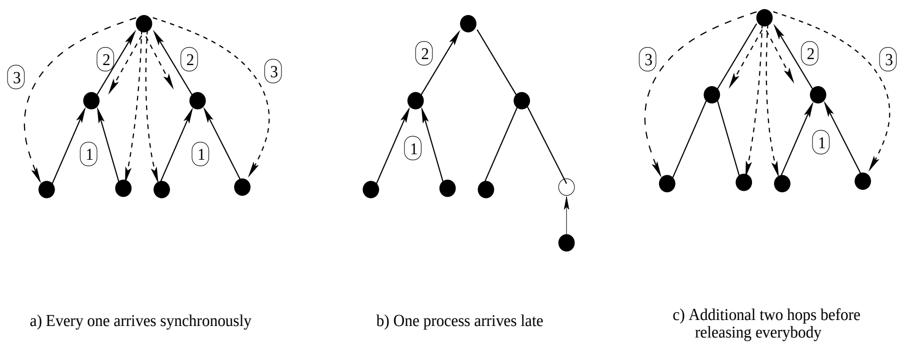

1
2
3
a) Every one arrives synchronously
b) One process arrives late
c) Additional two hops before releasing everybody

Figure 2.2: Impact of Process Skew

d) Performance and Memory Scaling:

One important criterion for any collective primitive design is the amount of memory consumed for a given performance margin. As the system scale increases, this issue gains high prominence as the application may allocate significant chunks of memory for communication straining available resources. This adversely affects the performance of the application. A best example of such scenario is the alltoall collective where each process communicates with all the other processes leading to a potential $O(n)$ usage of communication resources. The underlying message transport protocols play a key role in determining the resource consumption behavior. In order to enable good scaling, it is important that efficient transport methods be used. Moreover, the memory usage needs to be studied in conjunction with the performance obtained. Also, different collectives have varying communication patterns stressing the communication subsystem in separate ways. Thus, it is important to understand, in detail, the requirements placed on the system resources by a given collective primitive keeping into account the performance differences.

e) Collectives on Multicore Architectures:

Recently, owing to the physical limitations in speeding up clock rates there has been a paradigm shift in computing technology. Instead of a high clocked processing core, the trend is towards deploying several of them albeit with a lower frequency clock. The key to efficiently use these new breed of processors is to extract parallelism from the application. There are several flavors of multicore architecture emerging in the computing arena. As described below, the architecture of these machines can directly influence the performance of a collective operation.

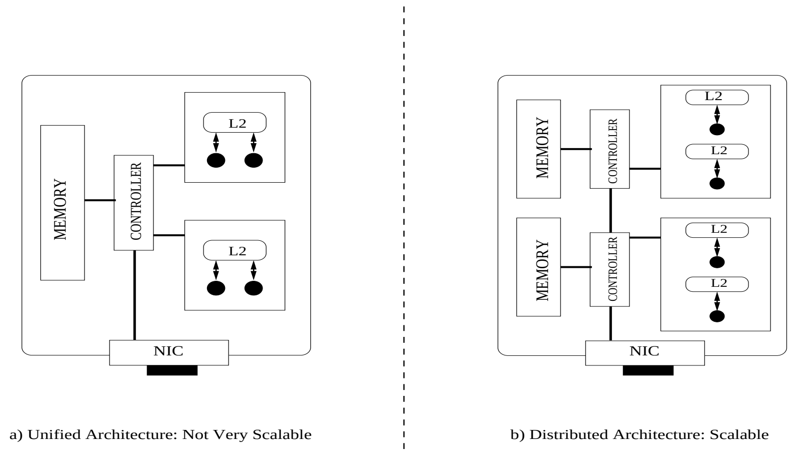

MEMORY
CONTROLLER
L2
L2
NIC
a) Unified Architecture: Not Very Scalable
MEMORY
MEMORY
CONTROLLER
CONTROLLER
L2
L2
L2
L2
NIC
b) Distributed Architecture: Scalable

Figure 2.3: Multicore Architecture

Figure 2.3 shows two broad strategies deployed in current multicore processors. One of them exhibits a unified architecture which limits the scalability but is simple to design. But, a distributed approach is more scalable incurring increased complexity for the design and fabrication. However, the trend is towards the latter approach as more and more cores are added to a single chip or socket. Collective communication within a node and across multiple processing cores usually comprises of multiple processes accessing shared data items in the memory. In such scenarios, depending on the underlying architecture, hot-spots could arise in the chip. These hot-spots could be either due to accessing the same cache/memory block or due to contention in the communication links or the bus on the chip. An in depth understanding of the multicore architecture is crucial to optimally design and tune the collective algorithms.

2.1.3 Collective Algorithms

We now provide an overview of all the important collective algorithms referred in this thesis. We also explain the basic limitations of these algorithms in meeting the requirements stated earlier.

Standard Algorithms for MPI Collectives

In this section, we present the standard algorithms in literature that are used to design MPI collectives. The details of this algorithms can be found in [42, 15, 8].

- *Pair-wise, Hypercube:* In this algorithm, the participating processes are arranged as the corners of an n dimensional hypercube where $n = log(p)$, p is the number of processes. The algorithm comprises of n steps where in step i processes exchange data in dimension i. This algorithm is applied for doing collective reductions. The data that is exchanged in ith step is obtained by reducing the data of a process with that obtained from its neighbor in step $i-1$.

- *Recursive Doubling, Hypercube:* This algorithm is similar to the one discussed above. It also performs $log(p)$ steps but instead of reducing the data, it is appended. In other words, the data that is exchanged in ith step is obtained by appending the data of a process with that obtained from its neighbor in step $i-1$. This is used for doing an Allgather or AlltoAll broadcast of data.

- *Dissemination Algorithm:* This algorithm is used for barrier synchronization. The basic idea is that process a sends a message to process $b = (a + 2^i)$ mod p in step i. It then waits for a message from process $(a + p - 2^i)$ mod p. This algorithm also takes $\lceil log(p) \rceil$ steps but the number of processes can be either even or odd. The Hypercube algorithms require the processes to be even.

- *Bruck's Algorithm:* This algorithm is similar to Hypercube algorithm and comprises of $n = log(p)$ steps. The data from a given process is routed along n hops and incurs the cost of storing and forwarding the message. It is used for doing Alltoall personalized communication for short messages.

- *Pair-wise, Linear:* For large messages, Alltoall personalized uses a linear pair-wise exchange algorithm which consists of $(n - 1)$ steps. In step i, the process with rank r sends and receives data from process with rank $r \oplus i$.

- *Direct Algorithm:* In this algorithm, each process directly sends messages to all the other processes. This is used for medium messages in Alltoall personalized communication. Messages are scattered so that not all messages are directed towards a node to avoid congestion and hot-spot effects. A simple rule could be for a process r to send the ith message to process whose rank is $(r + i)\%p$.

- *Large Vector Algorithms:* The basic idea in large vector algorithms is to maximally utilize all the communication links in the network. For example, a large message broadcast can be split into two phases. In the first phase, the message can be scattered across all the processes. In the next phase, the separate fragments are broadcast parallely. Similar approach is taken for Reduce/Allreduce. A Reduce-scatter is first done followed by Gather in Reduce and Allgather in Allreduce.

Limitations in current approaches

The algorithms described above are not enough to meet the requirements or address the issues explained in the beginning of this chapter. The reasons are as follows. All these algorithms are based on sending point-to-point messages i.e. at any given step, only two processes send or receive messages from one another. Modern networks provide the capability of sending a message to multiple nodes in one operation. This primitive called as *multicast* is supported in major interconnection networks. This capability cannot be leveraged by the standard algorithms. Thus, specialized approaches are necessary. Another reason why the above algorithms are not efficient is because they assume each process to be running on separate nodes. As described earlier, the evolution of multicores is enabling more cores per node. The locality of the processes is an important parameter that cannot be ignored while attempting to avoid the extraneous copying in a collective. Another important reason to have specialized algorithms is because the standard algorithms are oblivious to the transport protocols used by the network. This would result in incorrectly ascertaining the resource requirements and performance trade-offs for a given message transport. On a large scale cluster, this issue is of paramount importance. Moreover, these algorithms do not consider into account the architecture of multicores. Since collectives are directly impacted by the underlying architecture, new specialized algorithms are necessary. Finally, all the above algorithms are designed assuming that there is very little process skew in the system. These do not adapt well in varying skew conditions. Hence, there is a need for new and improved designs for collective operations and algorithms.

2.2 Trends in Interconnect and Multicore Systems

In this section, we provide an overview of the InfiniBand Architecture and its features. Specifically, we explain the different communication semantics provided by IBA and the associated transports on which these are based on. Further, we also provide an overview of Multicore architecture and the recent systems from Intel and AMD.

2.2.1 InfiniBand Architecture Overview

InfiniBand Architecture (IBA) [18] is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. As shown in Figure 2.4, a typical IBA cluster consists of switched serial links for interconnecting processing nodes and the I/O nodes. The processing nodes are connected to the fabric by Host Channel Adapters(HCA). HCA's semantic interface to the consumers is specified in the form of IB Verbs. The interface presented by Channel Adapters to consumers belongs to the transport layer. A queue-pair based model is used in this interface. Each Queue Pair is a communication endpoint. This can be seen in Figure 2.5. A Queue Pair consists of a send queue and a receive queue. Two QPs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple QPs. Communication requests are initiated by posting descriptors (WQRs) to these queues. InfiniBand supports different classes of transport services. These are explained in the following section.

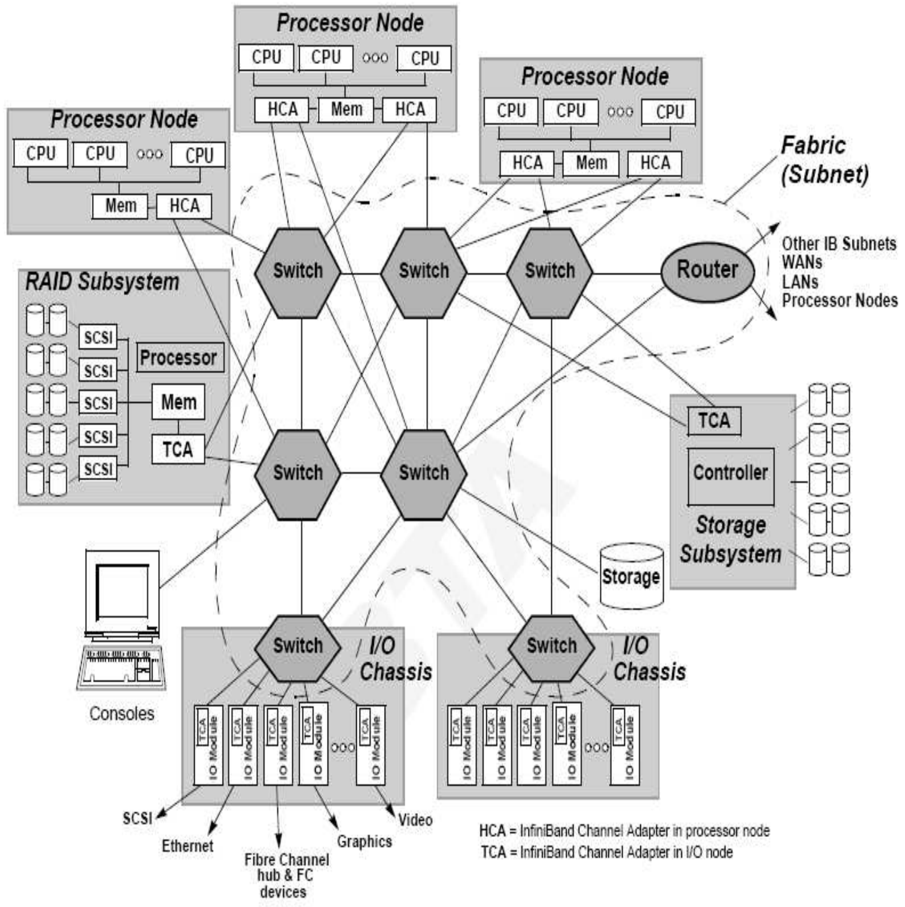


Figure 2.4: InfiniBand Architecture (Courtesy InfiniBand Trade Association)

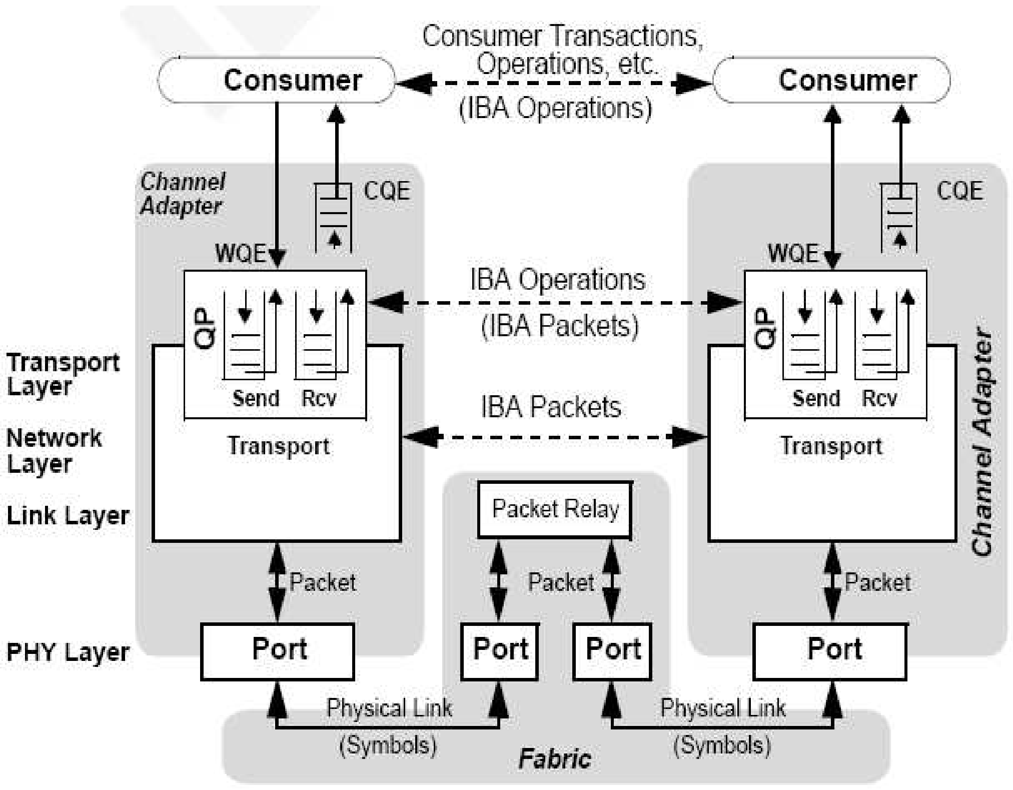


Figure 2.5: InfiniBand Protocol Stack (Courtesy InfiniBand Trade Association)

2.2.2 InfiniBand Transport Services

IBA provides for five transport services:

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Reliable Datagram (RD)
- Unreliable Datagram (UD)
- Raw Datagram

Except for RD, the current generation IBA adapters provide the support for all the other transport services. We now provide brief overview of the two relevant transports used in our study: Reliable Connection and Unreliable Datagram.

Connection-Oriented Reliable Connection (RC): IBA specifies the RC transport layer as a reliable communication layer. In this transport layer a connection needs to be established between two QPs, one on each node, before the data transmission. After connection establishment, the communication packets are completely managed by the RC transport layer. A message to be sent is broken down into required number of network MTUs and each of the packet is sent across the network. Further, the RC transport manages the reliability and ordering of all the network packets and delivers the message by combining all the packets at the remote end. However, the number of connections and thus the number of QPs required grows quadratically with the number of participating processes.

Connection-Less Unreliable Datagram (UD): The basic communication is achieved in the UD layer by exchanging network MTU sized datagrams. These datagrams can be sent to a UD QP by any other UD QP in the network. An explicit connection between the two QP's is not needed. Messages larger than the MTU size need to be broken down into multiple MTU sized messages before they can be sent over the network using an UD QP. The reliability and order of delivery of these datagrams is not guaranteed by IBA and needs to be managed by the application. But, compared to RC, only one QP is enough for all processes to communicate with each other.

Communication Context Caching at the NIC: To achieve good efficiency, critical data pertaining to the communication context is often cached at the NIC. The amount of data that needs to be cached is dependent on the type of transport used. For the connection-oriented transport, the NIC maintained cache called as ICM is used to store the queue pair context and completion queue information. It is also used to cache the latest address translation information of the buffers used. In the case of connection-less transport, the demand on the cache resources is much less as one queue pair is enough to communicate with any process in the network unlike the connection-oriented case which needs a QP for each pair of nodes.

2.2.3 Send/Recv and RDMA

IBA supports two types of communication primitives: Send/Recv with Channel Semantics and RDMA with Memory Semantics.

In channel semantics, each send request has a corresponding receive request at the remote end. Thus, there is a one-to-one correspondence between every send and receive operation. Receive operations require buffers posted on each of the communicating QP, which amount to a large number. In order to allow sharing of communication buffers, IBA allows the use of Shared Receive Queues (SRQ). SRQs allow multiple QPs to have a common Receive Queue.

In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations do not require a receive descriptor at the remote end and are transparent to it. For RDMA, the send request itself contains the virtual addresses for both the local transmit buffer and the receive buffer on the remote end. The RDMA operations are available with the RC Transport. IBA also defines an additional operation: RDMA write with immediate data. In this operation, a sender can send a limited amount of immediate data alone with a regular RDMA operation.

Figure 2.6 shows the basic working of both the RDMA and the Send/Recv models. The main steps involved are labeled with sequence numbers. The main difference between the two is the requirement of posting a receive descriptor for the send/recv model. It is to be noted that the current IBA specification supports channel semantics for RC and UD. However, RDMA is not provided over UD but supported in RC.

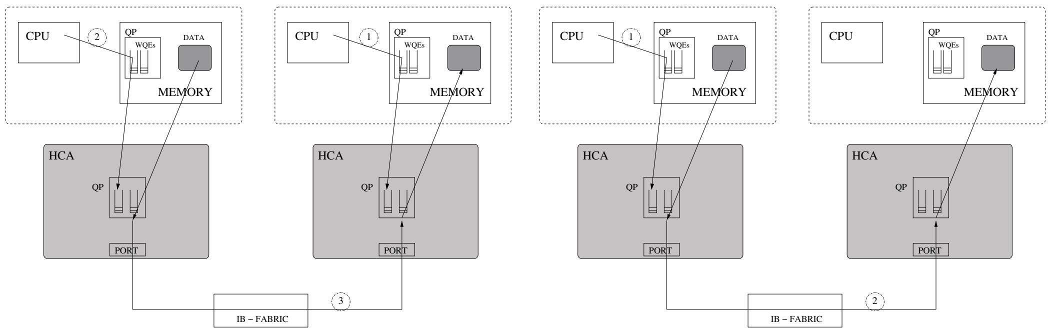


Figure 2.6: InfiniBand Transport Models (a) Send/Recv Model, (b) RDMA Model

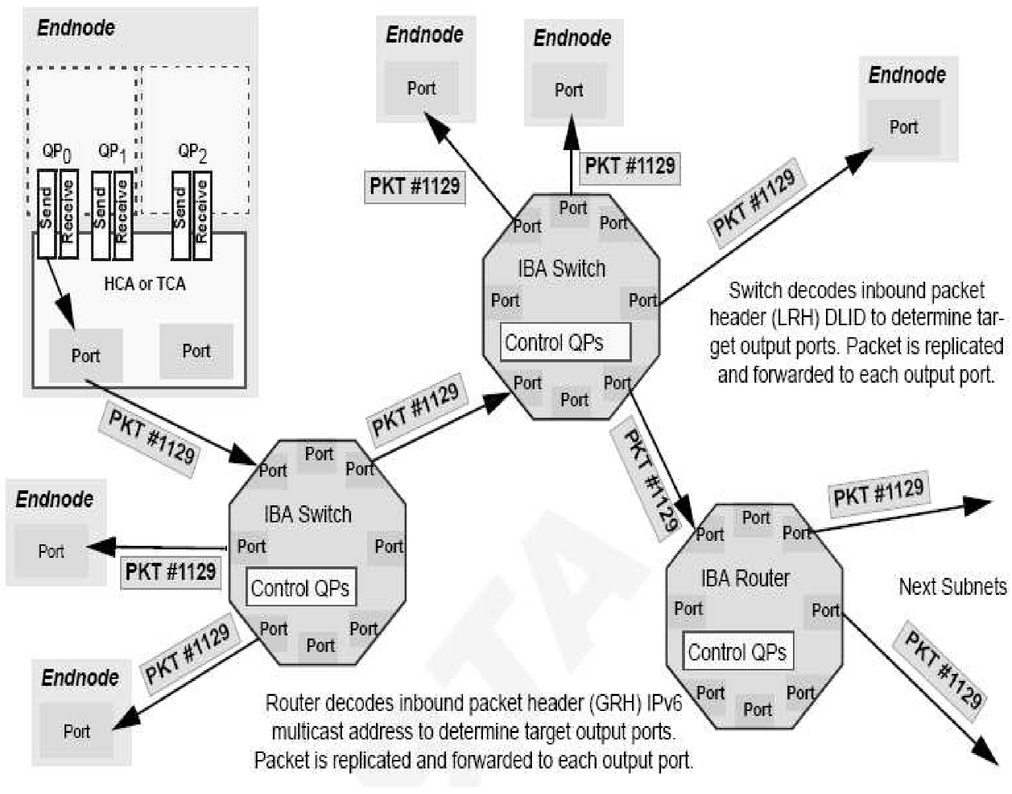

Endnode
QP0
QP1
QP2
Send
Receive
HCA or TCA
Port
PKT #1129
IBA Switch
Control QPs
Switch decodes inbound packet header (LRH) DLID to determine target output ports. Packet is replicated and forwarded to each output port.
IBA Router
Next Subnets
Router decodes inbound packet header (GRH) IPv6 multicast address to determine target output ports. Packet is replicated and forwarded to each output port.

Figure 2.7: InfiniBand Hardware Multicast (Courtesy InfiniBand Trade Association)

2.2.4 Hardware Multicast in InfiniBand

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

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

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

- Since the multicast is handled by hardware, it has very good scalability.

However, in InfiniBand H/W Multicast operation is available only under the Unreliable Datagram (UD) transport service. In UD, a connectionless communication model is used. Messages can be dropped or they can arrive out of order.

IBA Subnet Manager/Subnet Administrator: In InfiniBand, before multicast operations can be used, a multicast group which is identified by a multicast address must be created. Creating and joining multicast groups is achieved through the help of InfiniBand Subnet Manager (SM)/Subnet Administrator (SA). The combined functionality of Subnet Manager and Subnet Administrator is responsible for computing the multicast forward tables for a given multicast group. Communication to the SM/SA occurs via Management Datagrams called as MADs. The nodes willing to be included in the multicast groups send MADs to the SM/SA specifying the address of the group and other attributes like data rate and some specialized keys. After receiving the MADs, the SM/SA compute the multicast spanning tree and populate the relevant switches/routers in the fabric with routing information.

2.2.5 Multicore Architecture

In this section, we provide an overview of Intel Clovertown and AMD Opteron multicore platforms.

Intel Clovertown Architecture: We describe some of the important details of Intel Clovertown multicore architecture [19] in this section. These chipsets consist of dual sockets with quad cores in each of the sockets and a pair of adjacent cores share a common L2 level cache of size 4MB. The cache coherency across different L2 level caches is handled using bus based snooping MESI protocol [30]. Also, each of the socket has its own Front Side Bus (FSB) with a bus based snooping protocol to handle cache coherency. To achieve scalability in terms of memory bandwidth, Fully Buffered DIMM technology is used to design the memory system. We provide a detailed evaluation of this architecture in the following sections of the paper.

AMD Opteron Architecture: AMD multicore Opterons [2] are based on NUMA architecture with each of the sockets sharing independent memories. The number of cores in each of the sockets can vary from one to four with the latest Barcelona systems. All of the cores have independent L2 caches. Point-to-point HyperTransport links provide the required bandwidth scalability between the cores. Further, these links are connected by a 2-D mesh topology providing for scalable and less congestion-prone on-chip interconnection network.

CHAPTER 3

PROBLEM STATEMENT

MPI offers various collective routines with different functionality and communication patterns. It is important that these primitives provide optimal performance and scale to hundreds or thousands of cores. InfiniBand, on the other hand, provides advanced communication primitives with varying semantics which are in turn supported over different underlying transport protocols. Moreover, the emergence of multicore technology has fueled the number of cores that can be deployed on a single node. In this dissertation, we address the problem of designing MPI Collectives for these high-end multicore clusters connected by InfiniBand. Specifically, the objective of this dissertation is:

"How to design a Scalable and High Performance Collective Communication Subsystem for MPI by: a) leveraging directly the advanced network primitives of IBA together with their associated transports and b) taking into consideration the architectural attributes of emerging multicores?"

Illustrated in Figure 3.1 are the different issues and requirements of MPI Collective communication. As seen earlier, each of these components presents unique challenges. Also, the advanced network features of InfiniBand together with system capabilities provide good design opportunities to efficiently meet the stated requirements. In this context, we describe the specific problems focused in this dissertation.

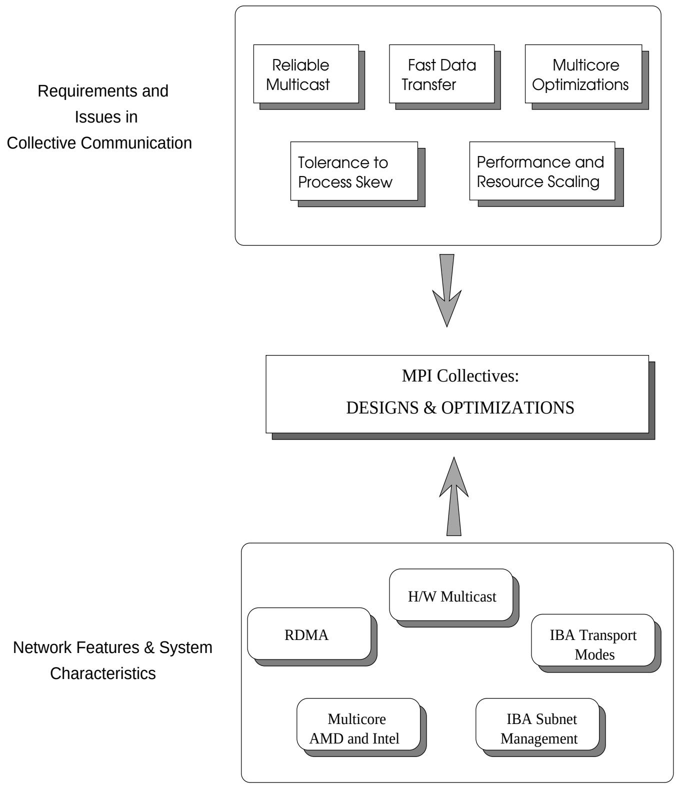

Requirements and Issues in Collective Communication
Reliable Multicast
Fast Data Transfer
Multicore Optimizations
Tolerance to Process Skew
Performance and Resource Scaling
MPI Collectives: DESIGNS & OPTIMIZATIONS
Network Features & System Characteristics
H/W Multicast
RDMA
IBA Transport Modes
Multicore AMD and Intel
IBA Subnet Management

Figure 3.1: Broad Theme

The dissertation addresses the following questions:

(a) What new communication protocols are needed to provide Reliable Multicast over IBA's H/W Multicast?

IBA H/W Multicast offers a scalable and highly efficient mechanism of replicating and delivering an MTU of data from a source process to a given set of destination processes. This primitive can be leveraged to design efficient algorithms for collective operations such as MPI_Allreduce, MPI_Barrier and MPI_Bcast. However, as described earlier, H/W Multicast poses its own suite of problems. Firstly, since H/W Multicast is over UD, it is unreliable and supports only one MTU of data delivery at a time. Secondly, H/W Multicast occurs over multicast groups spanning different nodes in the cluster. This leads to several issues involved in intelligently managing the multicast groups in MPI. This is particularly important for MPI which conceptually groups sets of participating processes in the collective into communicator objects. Thus, to successfully deploy H/W Multicast, a reliable multicast framework is essential requiring new communication protocols. These protocols need to handle reliability, large message transport and also efficiently manage H/W Multicast groups and MPI communicators.

(b) How can efficient Data Transfer Mechanisms eliminating extraneous copy overhead be designed using RDMA and/or System features?

As explained earlier, the overhead of copy in collectives is owing to the following major reasons: a) Extra set of buffers used to stage the data, b) Multiple transfer methods using different set of buffers for the same data and c) Choice of collective algorithm. RDMA provides the capability to directly access the application's data buffer. This provides ample scope for designing Zero-copy protocols. However, Zero-copy is also possible using IBA's Send/Receive. Thus, the advantages of using RDMA need clear study and analysis. Also, RDMA together with System's Shared Memory abstraction can potentially avoid using multiple buffers streamlining data transfers. Integrating these features to design a unified collective substrate presents a challenge. Moreover, matching the underlying algorithms used in the above components to the semantics of the collective greatly enhances the performance of the operation and needs detailed investigation.

(c) What mechanisms can be provided to tolerate Process Skew leveraging Reliable Multicast and RDMA?

Apart from providing good scalability and performance, Reliable Multicast and RDMA can also be used to design skew-tolerant algorithms. As mentioned earlier, collective operations are typically prone to skew when the participating process arrives out of step in a collective operation. Process skew hampers performance especially when the intermediate nodes participating in the algorithm arrives late. In such circumstances, the forwarding of the data is delayed until the particular process arrives at the collective. On the other hand, H/W multicast enables efficient delivery of data MTUs without the involvement of host processes. The challenging task is to effectively leverage this capability to design collective algorithms that can tolerate process skew and also provide high performance using RDMA. The problem poses issues such as how to capture the skew, how to adapt dynamically to changing skew and how to deal with possible race conditions.

(d) What are the benefits of using the existing IBA network primitives together with their associated transports for obtaining good performance vis-a-vis amount of resources consumed and are better IBA primitives required?

Collective operations differ in terms of their communication patterns. This directly relates to the demand placed on the underlying network resources. For example, in MPI_AlltoAll each process communicates directly to all the other processes. This would involve setting up connections to each process if RC transport is used, thus consuming $O(n)$ amount of memory. On the other hand, UD transport consumes only $O(1)$ memory, thus being highly efficient when memory usage is considered.

However, the performance of UD needs to be evaluated with respect to RC as the latter is a connection-oriented transport while the former being connection-less. Hence, it is imperative to carry out an in-depth evaluation of all the different combinations and choose the optimal transport methods and primitives. However, this is not straight forward, as in some cases performance may be penalized trying to optimize memory consumption. Further, certain primitives may be supported only by specific transport modes. This raises the question as to what kind of algorithms need to be designed taking the above issues into consideration and are new IBA primitives required which can provide scalability with good performance?

(e) What Architecture driven optimizations be proposed for emerging clusters featuring Intel and AMD Multicore processors?

Recent designs of multicore processors feature various architectural attributes resulting in several interesting ramifications. One such architectural feature is the design of multi-level cache hierarchies. The two broad strategies deployed in the current day multicores exist in processors from Intel and AMD. Intel processors provide for a shared L2 cache where as AMD multicores deploy HyperTransport links for quick data transfers. However, irrespective of these different hierarchies, both the systems enable very fast sharing of data across the cores. Further, to scale the bandwidth available to the coherency traffic, the cores are either connected via multiple buses as in Intel or by 2-D mesh HyperTransport links. Apart from providing good data movement capabilities across the processing cores, the caching hierarchies are useful in reducing the pressure on the memory bandwidth. This is especially true for scenarios when concurrent network transactions and intra-node communications occur. The effects of these multicore specific characteristics have to be thoroughly understood in order to design optimal collective algorithms and provide good application level performance.

CHAPTER 4

METHODOLOGY

In this chapter, we first explain the broad approaches used to solve the problems mentioned above. We then provide an overview of MVAPICH software architecture to understand the process of incorporating, evaluating and enhancing the designs considered.

4.1 Research Approaches

We now explain the methodology used in the dissertation. Figure 4.1 explains the basic framework for design and optimizing collective communication on InfiniBand multicore clusters. In the following parts of this section, we provide an overview of the research approaches pursued for each of the questions raised in the earlier chapter.

Designing new communication protocols to provide Reliable Multicast:-

Reliability mechanisms can be designed by employing either Acks or Nack-based approaches. In the Ack-based scheme, the receivers of a multicast message send Acks to designated roots. These Acks are then processed and possible re-transmissions dealt with. The Ack based scheme is useful for early detection and re-transmission in case of packet losses. This feature is desirable in cluster

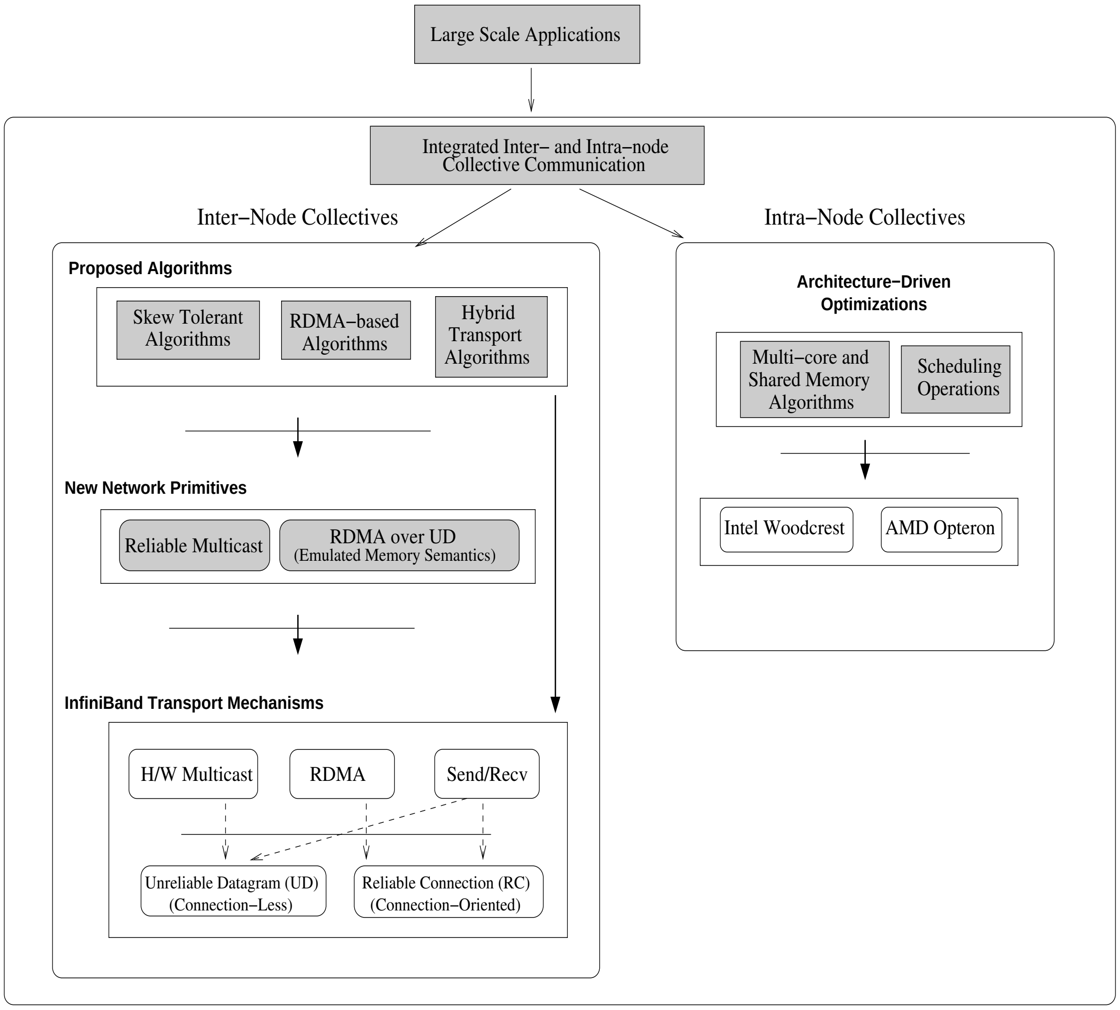

Large Scale Applications
Integrated Inter− and Intra−node Collective Communication
Inter−Node Collectives
Intra−Node Collectives
Proposed Algorithms
Skew Tolerant Algorithms
RDMA−based Algorithms
Hybrid Transport Algorithms
New Network Primitives
Reliable Multicast
RDMA over UD (Emulated Memory Semantics)
InfiniBand Transport Mechanisms
H/W Multicast
RDMA
Send/Recv
Unreliable Datagram (UD) (Connection−Less)
Reliable Connection (RC) (Connection−Oriented)
Architecture−Driven Optimizations
Multi−core and Shared Memory Algorithms
Scheduling Operations
Intel Woodcrest
AMD Opteron

Figure 4.1: Broad Overview

computing where performance is heavily dependent on communication latencies. The main issues which arise in the design are dealing with Ack-implosion and hot-spot effects at the designated roots. Our approach to deal with this problem is to deploy multiple “Co-Roots” aiding the main “Root” in reliability.

The second protocol for doing the H/W Multicast group management can be accomplished by using a dedicated set of clients interacting with IBA Subnet Manager (SM)/Subnet Administrator (SA). As seen from earlier section 2.2, the SM/SA is responsible for calculating the multicast routing information and configuring the routing tables in the fabric appropriately. The H/W multicast can proceed only if these activities are successfully completed. To take into consideration delays due to SM/SA, non-blocking mechanisms of group construction can be employed. In such mechanisms, the creation of multicast groups can be initiated and appropriate checks for the completion of the task can be performed at regular intervals.

The new communication protocols are discussed in detail in Chapter 5.

Providing Fast Data Transfer methods using RDMA and/or System Features:-

RDMA provides mechanisms for a process to directly write into remote process's memory enabling Zero-copy protocols. Further, RDMA capabilities match well with semantics of important collectives such as MPI_AlltoAll leading to improved performance. However, zero copy is also possible using the Send/Recv mode of transport. This can be accomplished by employing separate channels for every message source and preposting the receive descriptor pointing to the correct destination buffer. There are two major drawbacks of using this approach. Firstly, having separate communication channels leads to $O(n)$ utilization of resources limiting the scalability of the system. Mechanisms of having a single reception point for the incoming messages (such as Shared receive Queue Support of IBA) are needed to cater to meet the scalability requirements. However, this imposes ordering for the messages at the receiver due to the preposting of the descriptors. Using RDMA overcomes all these limitations. However, separate synchronization and exchange of application buffers' addresses are needed to design collectives over RDMA.

As the number of cores in each node rises steadily, it becomes particularly important to take into account shared memory as a means of communication within a node. Several design challenges emerge while coupling shared memory based intra-node communication with those of the network primitives discussed above like RDMA and H/W Multicast. Two broad methods of integrating inter- and intra-node collective communications can be investigated. The first method would involve selection of multi-level collective algorithms targeting inter-node communication at the first level followed by memory sharing at the next level and in the multi-core clusters, a third level comprising of cache sharing. The second method would be a more tightly coupled subsystem possible by making the network buffers shareable across the different processes.

Chapter 6 discusses the efficient data transfer techniques utlizing RDMA, H/W Multicast and Shared Memory.

Designing Skew Tolerent Algorithms:-

An effective strategy to minimize the impact of process skew is to make the topology used in the algorithm adapt to the skew pattern. The decision on which topology to choose should be such that the nodes which arrive early do useful work to absorb as much as possible the skew present in the system. Ideally, by the time the last node arrives, most of the collective should have been done and very less time is spent between the last node's arrival and the release of all the nodes. Please refer to Figure 2.2(a) which explains the problem with the current approaches. This topology change can be dynamic where all the required steps to change the topology are performed in one single barrier call as it progresses. Or it could be semi-dynamic where the topology used in one collective is based on the previous one. One drawback of the semi-dynamic scheme is that it assumes that the skew pattern is fixed across the barriers. If this is not the case, it fails to give good performance. The totally dynamic scheme does not rely on any such assumption and is optimal in varying skew conditions. The dynamic change in topologies can be applied to MPI_Barrier, MPI_Allreduce, MPI_Bcast, etc.

As an example, a totally dynamic scheme for MPI_Barrier can be designed using a token-based combining tree approach. The idea of using the token is that it can be progressively passed to the last arriving process in the tree. This process holding the token becomes the root of the new combining tree and can release all the nodes completing the barrier. This approach breaks the dependency on the intermediate node arriving late. The efficiency of such approach needs to be investigated for different scenarios with varying skew patterns.

The new design for skew tolerant collectives utilizing H/W Multicast and RDMA is explained in detail in Chapter 7.

Understanding the benefits of current IBA mechanisms and investigating better primitives:-

As discussed earlier, IBA provides for connection-oriented and connection-less transports. The connection-oriented transport can provide higher bandwidth since the hardware handles message segmentation/re-assembly leading to improved pipelining benefits. The connection-less transport on the other hand lacks in these benefits. However, these transports need to maintain $O(1)$ state compared to connection-oriented transports. This affects the caching achievable at the NIC influencing the performance of the collective operations. All these issues need thorough analysis and careful study. The collective MPI_AlltoAll can be benchmarked keeping in mind all the scenarios discussed above. Also, as a trade-off between performance and memory, a hybrid approach can be taken utilizing both the transports. In these approaches, the amount of connections can be tuned so as to bound the memory consumed. However, in these cases the performance can be penalized.

As discussed in earlier sections, RDMA communication semantics can potentially benefit collective operations. Further, collective operations such as AlltoAll can be "broken down" into a series of one-sided operations to obtain overlap of computation with communication. However, the use of RDMA is restricted by the transport over which it is supported. connection-less UD transport can provide good scalability compared to the connection-oriented RC but lacks in RDMA capabilities. In these scenarios, it becomes important to study the benefits of having RDMA semantics over UD. Since the current specification of IBA does not allow for RDMA over UD, RDMA emulation over UD is necessary to evaluate the merits and demerits of such an approach. The software approach of emulating RDMA over UD can potentially introduce additional overheads. Thus, supporting RDMA over UD in hardware presents a promising scenario.

We present the complete analysis of the different trade-offs involved in using different IBA transport mechanisms in Chapter 8. Further, we also describe the new "RDMA over UD" primitive desired to provide performance and also ensure good resource scalability.

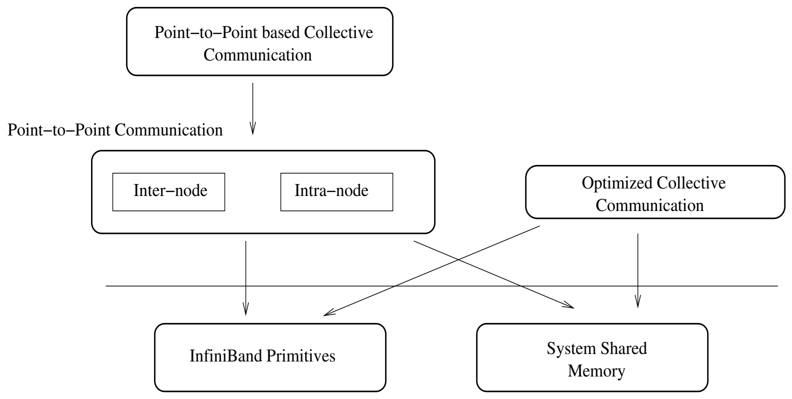
Optimizing Collectives on Multicore platforms:-

As discussed earlier, multicore architecture presents several opportunities and challenges in designing collective operations. Different architectures differ in the manner in which individual cores communicate with each other. Thus, it becomes important to understand these distinctions in detail and optimize the collectives accordingly. The two mainstream multicore platforms are from Intel and AMD. AMD deploys point-to-point Hyper-Transport technology for inter-core communication where as current generation Intel uses Bus-based approaches. Collective algorithms differ in the way optimizations are carried out on these platforms. For example, on the AMD nodes, more distributed approaches perform better because of more data parallelism available unlike Intel. This affects collectives such as MPI_Allreduce which involve both computation and communication and MPI_Alltoall, which performs data transfers. Further, it is interesting to study the interplay of cache-to-memory and memory-to-network data transfers in the context of collective operations.

Chapter 9 discusses all the multicore optimizations for different collectives such as MPI_Allreduce, MPI_Bcast, MPI_Allgather and MPI_Alltoall.

4.2 Collectives in MVAPICH

We now provide a high-level design overview of Point-to-Point and Collective Communication support in the MVAPICH stack. MVAPICH is a popular MPI over InfiniBand used worldwide. As shown in Figure 4.2, the software stack is composed of three main components: a. Point-to-Point operations, b. Point-to-Point based Collective operations and c. Optimized Collective Operations. These are explained briefly below.



Point–to–Point based Collective Communication
Point–to–Point Communication
Inter–node
Intra–node
Optimized Collective Communication
InfiniBand Primitives
System Shared Memory

Figure 4.2: MVAPICH Design Overview

Point-to-Point MPI operations in MVAPICH: The two main protocols used for MPI point-to-point primitives are the eager and rendezvous protocols. In the eager protocol, the message is copied into communication buffers at the sender and destination process before it is copied into the user buffer. These copies are not present if rendezvous protocol is used. However, in this case an extra handshake is required to exchange user buffer information for zero-copy of the message.

For intra-node communication, a separate shared memory channel is used for communication. In MVAPICH, the shared memory channel involves each MPI process on a local node attaching itself to a shared memory region at the initialization phase. This shared memory region can then be used amongst the local processes to exchange messages and other control information. Each pair of the local processes has its own send and receive queues. Small and medium messages are sent eagerly, where as a packetization approach is used for large messages.

Point-to-Point based Collective operations: In MVAPICH, all the collective algorithms discussed above in Section 2.1.3 are implemented over Point-to-Point operations. Most of these implementations are not optimal as the algorithms do not leverage any benefits of group communication. For example, combining and processing of data from the different processes participating in the same collective call can lead to extraneous performance overheads. This is because of the extra amount of copying involved as the Point-to-Point operations are not aware of the collective communication. Also, the network primitives such as H/W Multicast cannot be exposed to the upper layer using Point-to-Point operations.

Optimized Collective Operations: As discussed above, Collective operations implemented directly over IBA and system shared memory can lead to significant performance gains. Some of the collectives already implemented in this fashion are MPI_AlltoAll, MPI_Allgather and MPI_Barrier. The focus of this dissertation is to leverage mechanisms of RDMA, H/W Multicast and system shared memory for optimized collective communication support in MVAPICH.

CHAPTER 5

RELIABLE MULTICAST

In this chapter, we focus on new communication protocols required to design Reliable Multicast. Reliable Multicast can be used directly by MPI_Bcast or as building blocks for other collective operations such as MPI_Allreduce and MPI_Barrier. We first explain how reliability can be provided over the Unreliable H/W Multicast primitive. We then propose mechanisms for efficiently managing the H/W Multicast groups by interacting with the IBA Subnet Manager/Administrator.

5.1 Reliability protocol for IBA's H/W Multicast

There have been many studies about multicast and reliable multicast in the networking area [17, 14, 23]. A majority of the work done in this area focuses on networks based on TCP/IP protocol. Our thesis deals with implementing reliable multicast in InfiniBand. Compared with a general TCP/IP network, InfiniBand offers much higher communication performance and hardware supported multicast. Also, group membership in MPI is much more static than that in the dynamic environment of a TCP/IP network.

In the following, we propose several designs used for providing reliability. We first describe a basic design which is easy to understand and also straightforward to implement. Then we present several new designs which deal with performance and scalability issues of the basic design. We have chosen ACK based approaches, in which delivery is confirmed by acknowledgments and message loss is handled by timeout/retransmission.

5.1.1 Basic Design

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

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

However, there are several major problems in this basic design. First, making the root block for all the ACKs significantly increases the overhead of the reliable multicast call at the root. Second, since all other nodes send back ACKs to a single root node, a hot spot is created at the root, which becomes a performance bottleneck when the total number of nodes in a system is large. This problem is also referred to as *ACK implosion* [34]. In the following subsections, we will address these problems.

5.1.2 Sliding-Window Based Design

Our basic design leads to poor performance because the root has to wait for all the ACKs to be received. In order to alleviate this problem, we propose a solution which makes a copy of the user buffer. After the multicast operation is initiated, we can immediately return without waiting for all the ACKs to be received. To handle multiple outstanding reliable multicasts initiated from a single root, we use a number of pre-allocated buffers at each root. These buffers are organized as a ring. A sliding-window based approach is used to manage these buffers, as shown in Figure 5.1. A buffer is consumed for each new reliable multicast operation. When all ACKs for this operation have arrived, this buffer can be freed and reused for other reliable multicast operations.

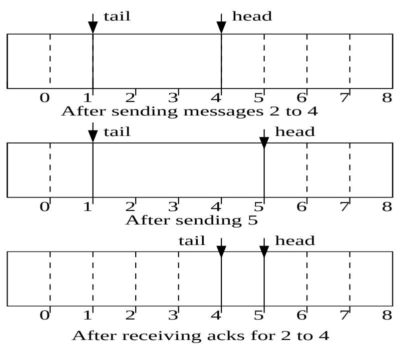


Figure 5.1: Sliding Window Buffer Management

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

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

5.1.3 Avoiding ACK Implosion

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

The drawback of the tree based ACK collection is that it depends on intermediate nodes for ACK processing. Thus, ACK collection time depends on the communication progress of intermediate nodes. (A similar problem has been discussed in [10].) In a polling based MPI implementation such as MPICH, communication progress is only made within MPI function calls. Therefore, if an intermediate node is doing lengthy computation, ACK processing and forwarding could be delayed. The problem becomes even more serious when the tree has multiple levels. As a result, it is very hard to determine the timeout value for retransmission at the root. When ACK processing at intermediate nodes are delayed, the tree based ACK collection is prone to *false retransmission*, which is triggered by delayed ACKs instead of real message loss. To make matters worse, a single delayed ACK will result in the root node retransmitting the message to everyone in the same sub-tree, which can generate a lot of network traffic and increase the overhead of the root node.

To solve the ACK implosion problem and also to address problems with the tree based scheme, we propose a new ACK collection scheme called the *co-root scheme*. In this scheme, in addition to the root node, we select a subset of other nodes as *co-roots*. The remaining nodes are called *leaf nodes*. Each of the root and the co-roots is responsible for a group of leaf nodes. The basic idea is to guarantee that co-roots can get messages reliably and use them to help ACK processing. The co-root scheme is illustrated in Figure 5.2 and it consists of the following steps:

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

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

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

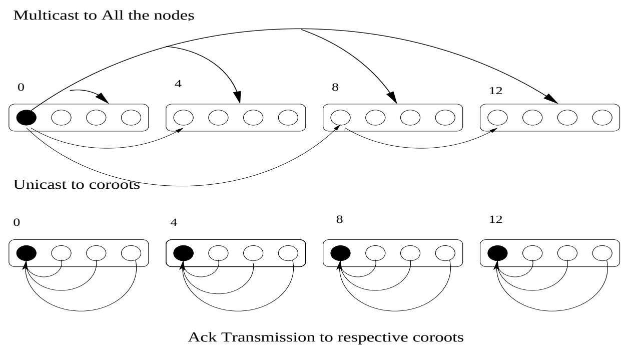

Multicast to All the nodes
0
4
8
12
Unicast to coroots
0
4
8
12
Ack Transmission to respective coroots

Figure 5.2: Co-Root Scheme

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

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

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

5.1.4 Benefits and Usage

Reliable Multicast can be used to design various collective operations. The important operations being MPI_Bcast, MPI_Allreduce, MPI_Barrier and MPI_Allgather.

Designing a low latency and scalalable MPI_Bcast using H/W Multicast is already discussed in [24]. It uses the Reliable Multicast protocol in its underlying design. The paper also discusses the effects of the various parameters in the design of MPI_Bcast, for e.g. the number of co-roots etc. Moreover, one important benefit of using H/W Multicast is tolerance to process skew. Also, as discussed in the paper [24], using Reliable Multicast protocol makes collective resilient to process skews. As we show in the subsequent chapters, Reliable Multicast can be used to design skew tolerant collective algorithms for MPI_Allreduce and MPI_Barrier.

5.2 Managing MPI Communicators and H/W Multicast Groups

In this section, we explain the approach taken for managing multiple MPI communicators and H/W multicast groups. We first explain the steps needed for creating communicators over H/W multicast groups. Then, we describe our approaches of doing these tasks efficiently.

5.2.1 Communicator Creation Mechanism

Though there are two types of communicators *intra* and *inter* defined in MPI, we focus on *intra* communicators. We have implemented all our designs using the MPI_Comm_create function. The inputs to this function are an already existing communicator object, a process group object comprising of a new subset of processes and the final communicator object. MPI_Comm_create is a collective call invoked by all the processes in the existing communicator. In the following discussions, we focus on the communicator creation in the context of mapping these to the hardware multicast groups. All the other steps like the assignment of a unique context and the local ranks have already been done by the time we start constructing the multicast group.

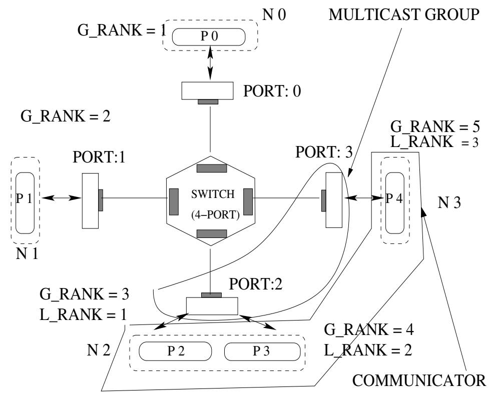


Figure 5.3: Mapping between IBA Multicast Groups and MPI Communicators

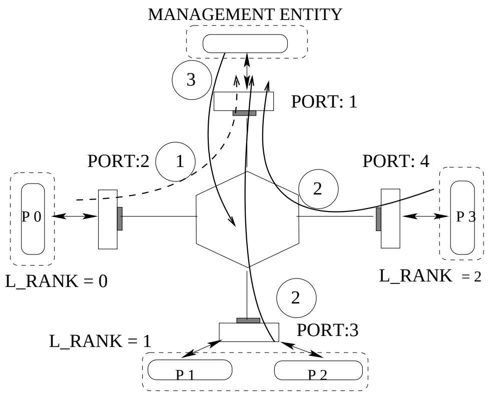


Figure 5.4: Multicast Group Setup Operations

5.2.2 Basic Protocol for H/W Multicast Group Management

The following steps are involved in the basic communicator construction. The mapping between IBA Multicast Groups and MPI Communicators is shown in Figure 5.3. Below, we describe the steps involved in setting up this mapping. All of these are illustrated in Figure 5.4.

Multicast create and join: In this step, the process whose local rank is zero issues a create request to the multicast management entity specifying the Multicast Group IDentifier (MGID)(step 1 in Figure 5.4). The remaining processes then issue join requests to the multicast management entity using the same MGID (step 2 in Figure 5.4). All these requests carry the port identifiers so that the management entity knows which all ports would like to join a multicast group. The multicast management entity after receiving and validating the requests computes a logical spanning tree containing the ports specified in the requests. It then updates all the routing tables of the participating switches in the fabric (step 3 in Figure 5.4). At this point of time, the set up of hardware multicast group is complete.

However, the participating processes have no knowledge of this information. One approach to accomplish this would be to let the multicast management entity notify the MPI application after updating the routing tables. Another approach would be to let the MPI application discover about the completion independently. We have taken the latter approach in all our designs as it does not depend on any particular implementation of the multicast management entity. We refer to this approach as *multicast testing*.

Multicast testing: In this approach, the following algorithm is implemented by all the processes after they finish issuing the requests. Process with rank zero who is the root, posts a multicast *ping* message to the new hardware multicast group and waits for Acks from all the other processes. If the routing has been done, the message is received by all the processes and these processes soon post the Acks to the root. On the other hand, if routing is not complete then the message may not arrive at some of the processes. These processes block waiting for the *ping* message. Meanwhile, the root retransmits the *ping* message after a certain time-out interval. This process repeats until everyone has received the *ping* message.

5.2.3 Lazy Approach for improved overlap

Although the Basic design is good for its simplicity, it is blocking in nature. The application has to wait for the multicast management entity to process the requests and update the routing tables. Until then, all the processes block in the *multicast testing*. Depending on the size of the cluster and the multicast group this can take a considerable amount of time. Instead of doing the *multicast testing* in an eager fashion within the communicator creation call, we do this in a lazy manner by calling this routine every time a collective call is made. We do this until the *multicast testing* phase is over. We accomplish this by making the *multicast testing* as a non-blocking routine.

Asynchronous return: The new *multicast testing* is implemented in the following manner. The root process posts the *ping* message and checks for the arrival of the Acks from the rest of the processes. It does not block for the Acks to arrive. In the subsequent collective calls to this routine, it repeatedly checks for the progress of the Acks. It reposts the *ping* message only if the timeout is exceeded. The root keeps an estimate of the time elapsed by recording the time-stamps in the communicator object. The remaining processes behave in a similar fashion. They check for the *ping* messages in a non-blocking fashion and post the Acks soon after discovering the *ping* message.

Point-to-Point fall back: One important issue requiring detailed attention is the progress of the collective communication call before the communicator is ready for hardware multicast. In our approach, all the collective communication traffic is transmitted via point-to-point messaging until the root discovers that the routing has been done.

This approach overcomes the drawbacks of the Basic design. Due to the asynchronous nature of the *multicast testing* routine, overlap of computation as well as communication is easily achievable.

5.2.4 Multicast Group Pool Based Design

Though the Lazy approach can effectively hide the overhead of hardware multicast group construction in the MPI application, it still has some drawbacks. The benefits of hardware multicast in an application is reduced if the set-up time of the multicast groups is high and the collective communication follows the setting up of these communicators. Using our earlier design, the communication traffic falls back to point-to-point if the multicast groups are not set up. But, this does not improve the performance of the application.

Multicast Group Pool: We overcome the drawback mentioned above using a *complementary* approach of setting up communicators explained as follows. The basic idea in this design is to have a certain pre-defined pool of multicast groups already constructed. These groups contain all the processes to begin with. In the communicator construction routine, instead of participating nodes joining the multicast group, the non-participating nodes leave a multicast group chosen from the pool. There are several advantages of using this approach. First of all, since the multicast groups are already set-up the routing tables in the fabric are in place. So, when the application calls communicator creation function we can use the multicast group directly and we avoid the overhead of the *multicast testing* phase. This approach considerably improves the utility of the hardware multicast groups in an application. Secondly, the multicast pool can be maintained easily as most of the overhead is due to the multicast management entity and can be done in the background. We now explain the steps involved in this design.

When a call to the communicator creation is made, first a multicast group is chosen from the available list of multicast groups already constructed. If this pool is empty we fall back to the Lazy approach explained in the previous section. Once an available multicast group is obtained, the non-participating processes issue leave requests to the management entity. The list of non-participating process can be easily obtained by subtracting the set of the processes involved in the communicator from the global set involving all the processes. This global set is the MPI_GROUP_WORLD process group in MPI. Once a multicast group is consumed from the pool, it is immediately replenished by making all the processes issue requests for group construction. We also need to check for *multicast testing* before including the group in the pool. However, this check is done in the background by the application. The initial pool can be either constructed by the management entity or by the MPI application in the initialization phase. We have taken the latter approach in our implementation.

5.2.5 Performance Evaluation

Each node in our experimental testbed has dual Intel Xeon 2.66 GHz processors, 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. They are equipped with MT23108 InfiniBand HCAs with PCI-X interfaces. An InfiniScale MTS14400 switch is used to connect all the nodes. OpenSM, version 1.7.0, is the multicast management entity used in our tests.

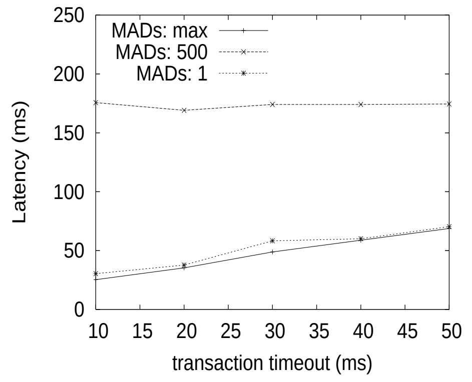


Figure 5.5: Tuning of Multicast Testing

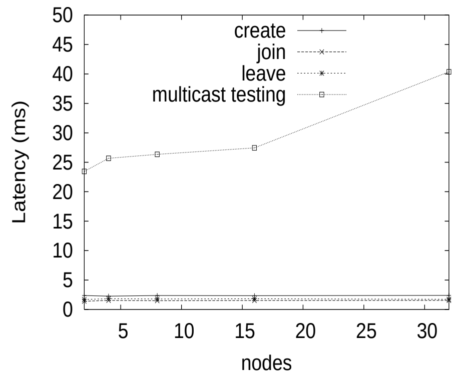


Figure 5.6: Overhead of Basic Multicast Group Operations

Basic Hardware Group Setup Latencies: OpenSM has two parameters which affect the performance of multicast group creation. These are: 1) timeout which is the time for transaction timeouts in milliseconds and 2) maxMADs which is the number of MADs that can be outstanding on the wire at any given point of time. We measure *multicast testing* to tune these parameters as this reflects the time taken by OpenSM to configure routing tables. Figure 5.5 shows these results. From these we have chosen 10 ms for timeout and the number of outstanding MADs is set to maximum for OpenSM to deliver best performance.

Figure 5.6 indicates the results of the basic multicast group operations like create, join and leave. We also present the *multicast testing* time for varying number of nodes. As the figure indicates, *multicast testing* overhead is very high compared to the latencies of issuing create, join or leave requests. This is because as explained in the previous sections, after the requests are issued the management entity has to compute the spanning tree and update routing information of the switches in the fabric.

Effective Latency of Suggested Schemes: To compare the different schemes suggested in the thesis we have measured the effective latency which is the latency of MPI_Bcast operation together with the communicator creation time. We have chosen the size of the message to be 1024 bytes in all our tests. The benchmark is constructed by calling communicator creation followed by the communication calls as many as the number of iterations specified. This is done for communicator sizes of 16 and 32 respectively.

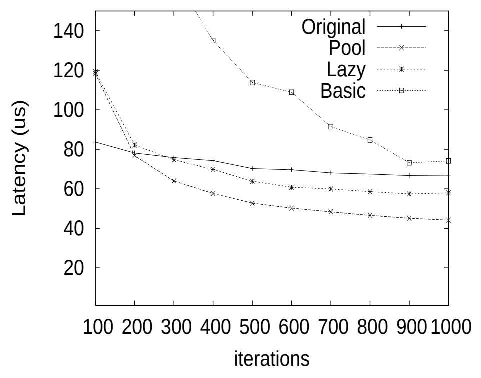


Figure 5.7: Effective Latency with Collectives 16 processes

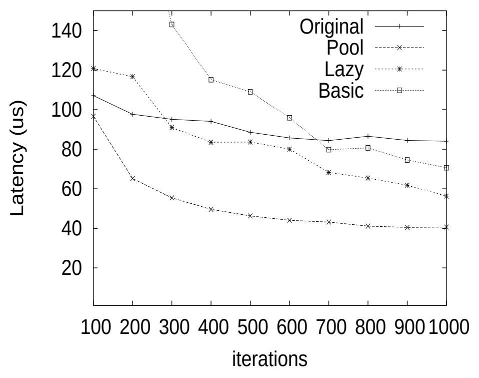


Figure 5.8: Effective Latency with Collectives 32 processes

In Figure 5.7, we measure the effective latencies for varying number of iterations for all the three schemes: Basic, Lazy and Pool. We have also taken the traditional point-to-point collectives as the reference. We refer to this as the Original design in the figures. As shown in the figure, the Pool based design outperforms all the rest. This is because *multicast testing* phase can be fully overlapped with the communicator creation operations and also the multicast group is immediately available. For the Lazy approach, we see the benefits of hardware multicast with the increasing number of iterations. This is because of the increasing percentage of communication using hardware multicast rather than point-to-point. The basic design performs poorly compared to all the designs. This is due to the high overhead associated with the *multicast testing* which is not overlapped with communication. Figure 5.8 shows the same trend for communicator size of 32. Note that the latencies of Pool and Lazy are almost the same for 16 and 32 for higher number of iterations. This is due to the scalability of hardware multicast.

To understand the overlap with computation we have introduced some computation between the communicator creation and the communication in the benchmark used for the above experiments. Figures 5.9 and 5.10 show the trend with increasing computation for sizes 16 and 32 respectively. The Lazy approach due to its asynchronous nature can overlap communicator creation with computation where as the Basic cannot. The Pool based design on the other hand can immediately take the benefits of hardware multicast. However, the initial latencies for size 32 are higher than for size 16 due to the increased overhead of creating larger hardware multicast group. As Figure 5.10 indicates, the Pool based design and the Lazy approaches improve the effective latency by a factor of 4.9 and 3.8, respectively.

5.3 Relation to Existing Work

There have been many studies about multicast and reliable multicast in the networking area [17, 14]. A majority of the work done in this area focuses on networks based on TCP/IP protocol. Our work in this paper deals with implementing MPI_Bcast in InfiniBand. Compared with a general TCP/IP network, InfiniBand offers much higher communication performance and hardware supported multicast.

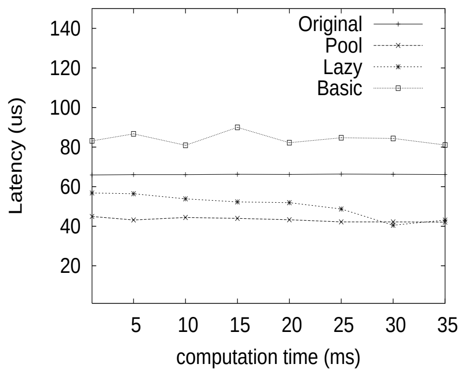


Figure 5.9: Effective Latency with Computation and Collectives 16 processes

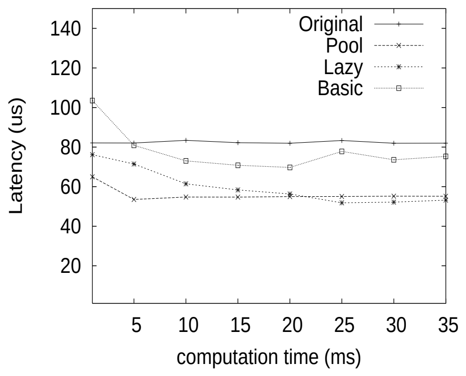


Figure 5.10: Effective Latency with Computation and Collectives 32 processes

Also, group membership in MPI is much more static than that in the dynamic environment of a TCP/IP network. Recently, different collective operations in MPI have been studied on high speed interconnects such as Virtual Interface Architecture (VIA) [16], Quadrics [32], Myrinet [54] and IBM SP [46]. Compared with these interconnects, InfiniBand provides new challenges and opportunities for implementing MPI collective operations.

Various aspects of subnet management like subnet discovery, routing and setting up of forwarding tables have been studied using simulation techniques by the authors in [5] [6] [37]. Paper [11] deals with implementing MPI collective operations using IP multicast over Fast Ethernet. In [55], the authors propose different designs for constructing IP multicast groups. Also, collectives have been implemented using hardware multicast and NACK-based schemes in [1]. Our work differs from these as we provide dynamic schemes of hardware multicast group construction in the context of InfiniBand and we overlap these with the application progress.

5.4 Summary

In this chapter, we described how to take advantage of hardware multicast in InfiniBand to implement Reliable Multicast operation in MPI. We proposed a *Reliability Mechanism* that overcomes the problem of dropping packets by the network. To improve performance of Reliable Multicast, we use sliding window based design which removes much of the processing from communication critical path. To further balance and reduce processing overhead, we proposed techniques such as the *co-root* scheme and *delayed ACK*.

Further, we proposed efficient schemes of dynamically constructing communicators with hardware multicast support in InfiniBand. The basic idea behind the schemes is to overlap the group construction with the progress of the application. The Multicast Pool and the Lazy approaches proposed move most of the overhead of multicast group creation out of the critical path of the application execution. We have evaluated these schemes together with the Basic scheme and found that the Multicast Pool performs the best of all the three followed by the Lazy scheme. Multicast Pool and Lazy schemes improve the Effective Latency by a factor of 4.9 and 3.8 respectively.

CHAPTER 6

EFFICIENT DATA TRANSFER MECHANISMS

In this Chapter, we first present the benefits of using RDMA for collective operations such as MPI_Alltoall. We then describe a combined RDMA and Shared memory algorithm for MPI_Allgather which removes extraneous copy-overhead and improves data transfers via pipelining. Finally, we explain an integrated approach of using H/W Multicast across the nodes and Shared memory within a node for MPI_Bcast.

6.1 High Concurrency RDMA-based *AlltoAll*

We now present the design and performance benefits of employing RDMA for personalized collective operations such as Alltoall.

6.1.1 Detailed Design

AlltoAll, as described in Section 2.1.3 uses a direct algorithm where each process issues simultaneous non-blocking send operations to all the other processes. The current implementations use copy based approaches over Send/Recv for implementing these. These copy costs become significant as the system size increases. This is because the total message size exchanged in Alltoall grows linearly in proportion to the number of participating processes. Thus, we need zero-copy approaches to remove this overhead. There are two alternatives of achieving zero-copy: (i) using Channel semantics or Send/Recv and (ii) using Memory semantics or RDMA.

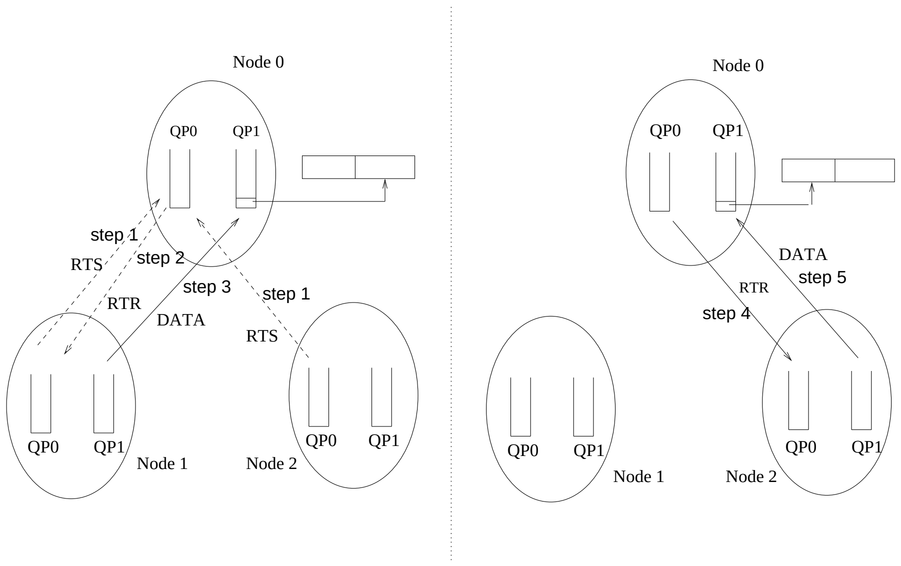


Figure 6.1: Zero Copy over Channel Semantics

Zero-Copy over Channel Semantics:

A zero-copy protocol using channel semantics is outlined in Figure 6.1. The basic idea is to use a handshake to make sure that the remote side has preposted the descriptor pointing to destination buffer. As outlined in the figure, the sender registers its buffer and sends the RTS message to the remote side. The receiver upon receiving the message, pins its buffer and sends the address information in the RTR message to the sender. The sender upon receipt of this message posts the send descriptor. This mechanism applies well when a single pair of process is communicating. However, if more than one process is sending messages to the same destination node, serialization of the messages occurs if a single receive queue is used for the incoming data messages. This is the case with the Shared Receive Queue of IBA. This is because until the message from one process is received, the destination node cannot send RTR message to the other process. This is indicated in Figure 6.1. Though the RTS messages have been concurrently issued by both the processes in the first step, the RTR message arrives only in the fourth step after the data delivery of the previous message is completed.

As illustrated above, achieving zero-copy over Send/Recv leads to serialization of network transactions and our results indicate that copy-based approaches perform better. RDMA on the other hand provides benefits of zero-copy. Utilizing RDMA allows the direct transfer of data from the source buffer to the destination buffer with the ordering taken care of automatically by the network. Most importantly, this also holds true irrespective of the global order in which the messages are injected into the network. This plays a significant role in the AlltoAll personalized operation where messages are sent directly in a non-blocking manner to all the processes. We now present important issues for doing direct *AlltoAll* over RDMA.

Zero Copy over Memory Semantics:

The main idea in our approach is to expose the registered receiver buffer to all the processes participating in the collective operation. Doing so enables direct transfer of data from a given process send buffer to the target buffer as shown in Figure 6.2. This would also require pinning of the send buffers of the *AlltoAll*. Once the framework for zero-copy is ready, the processes issue non-blocking RDMA

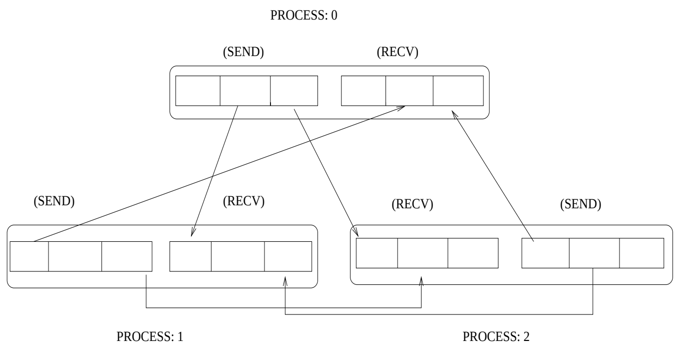


Figure 6.2: High-concurrency AlltoAll

write operations according to the direct algorithm as discussed in Section 2.1.3. All these operations place data directly into the receive buffers thus ensuring zero-copy. The unpinning of the memory buffers can be done in a lazy manner as done in MVAPICH. This cuts down the overhead of pinning and un-pinning when the application re-uses the same buffers.

However, one salient observation to be made in this approach is the need for explicit synchronization before the *AlltoAll* begins. This is because a given process can access a remote process's receive buffer only after the remote process called the *AlltoAll*. Issuing a write operation before that causes incorrect behavior due to two reasons. Firstly, the framework for zero-copy might not be ready for the write to succeed. Secondly, even if the framework is ready the application might still be using the receive buffer. In this case, writing into this buffer is clearly not admissible.

Address Exchange and Completion Semantics: Address exchange is a primary requirement for zero-copy RDMA-based protocol. In our approach, we exchange addresses along with the synchronization phase. This would be similar to the allgather operation of MPI. However, if the application uses the same receive buffer, then this address can be cached so that the address exchange can be eliminated. The decision to use the cached copy or not can be taken during the synchronization phase. If the process needs to use a different buffer, it explicitly notifies this to the other processes during synchronization. We have used a $log(n)$ algorithm similar to the one used in Barrier and Allgather, mentioned in Section 2.1.3.

For tracking completions, we have used RDMA with Immediate of IBA so that a completion entry is generated whenever a RDMA write finishes. All the processes poll for $(n-1)$ completions where n is total number of participating processes. The work requests for using the Immediate mode of RDMA can be preposted on the SRQ.

Relation to existing RDMA-based designs: Collective Algorithms have been studied well in the past. Thakur et al have optimized various collective algorithms over MPICH over Myrinet and IBM SP [42]. However, the focus of our work is on efficient collective and one-sided support over InfiniBand. RDMA collectives like MPI_Alltoall have also been studied over MVAPICH [40, 29]. Our approach is different from these as we focus on high concurrency collective patterns. In this context, the semantics of RDMA match well with direct MPI_Alltoall algorithm.

6.1.2 Performance Evaluation

In this section, we explain the tests conducted and the analysis of the results. We first briefly describe our experimental testbed.

Each node of our testbed has two 3.6 GHz Intel processor and 2 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes are equipped with MT25208 HCAs with PCI Express interfaces. A Flextronics 144-port DDR switch is used to connect all the nodes. The operating system used was RedHat Linux AS4.

We now demonstrate that zero-copy techniques for high-concurrent network transactions over Send/Recv with SRQ is not a good idea. We then present the performance benefits of RDMA compared to the copy-based approach.

Concurrent vs Serial: In this test, several nodes send messages to the root node. The benchmark is implemented in two different ways, with and without copy. The zero copy test follows the protocol explained in Section 6.1.1 and imposes serialization. With copy, the root copies the messages from pinned buffers to the receive buffer but there is concurrency in network operations. As can be seen from Figure 6.3(a), using copy-based or concurrent transactions performs considerably better than zero-copy or "serial" as in the figure.

AlltoAll over RDMA vs Send/Rec: The performance comparison of the zero copy RDMA-based *AlltoAll* vs Copy-based approach is shown in Figure 6.3(b). As can be seen from the figure, the zero-copy *AlltoAll* performs about 38% better for the 32 nodes. Also, for more number of nodes, the performance gains are over 33% for small to medium messages. This demonstrates the impact of using memory semantics vs channel semantics for doing zero-copy high-concurrency collective operations. The copy-based *AlltoAll* is based on SRQ channel semantics incorporated into MVAPICH.

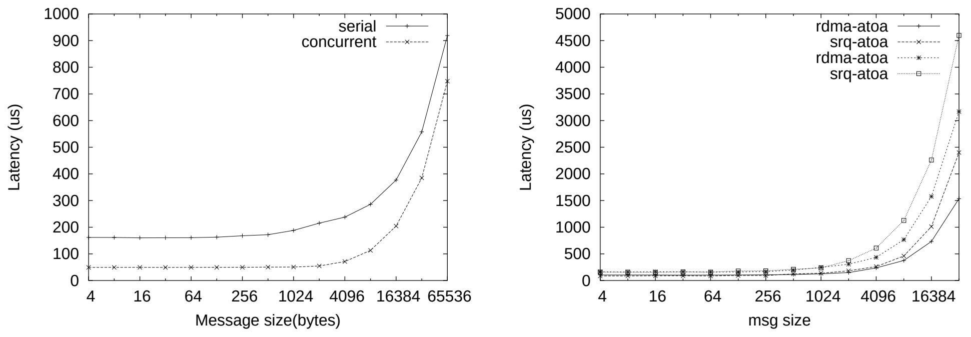


Figure 6.3: (a) Concurrent vs Serial (b) RDMA vs Send/Recv

6.2 RDMA and Shared Memory based Collectives

In this section, we describe our method of using RDMA and Shared Memory for designing efficient collective algorithms. Particularly, we have shown how performance of MPI_Allgather can be improved using this technique.

6.2.1 Design and Implementation

The basic idea used in our approach is to use a common memory segment both for intra and network communication. This memory segment is shared across all the processes local to the node. Further, this segment is pinned so that it can be accessed directly by the NIC for the network operation. We now outline the main steps involved in our approach.

Our Approach: We extend the recursive doubling algorithm discussed earlier to be performed across the nodes rather than across the processes. In this fashion, a single message is exchanged per a pair of nodes irrespective of how many processes are scheduled on a node. This is accomplished by making all the local processes write their data into the shared memory segment in the initial step. This is the step 0 as shown in the Figure 6.4. Once all the processes have written the data into this buffer, the data exchange starts over the network. In the first step, node pairs 0, 1 and 2, 3 exchange the data. Note that the data exchanged in this step is one fourth the size of the total data. After this step, the second step as shown in the Figure 6.5 begins. The size of the data exchanged in this step is doubled as seen from the figure. The pairs which are involved in this exchange are now 0, 2 and 1, 3. Once this step is completed, each node has the data from all the processes. In the final step, which is the step four, the data is copied out of the shared memory segment.

As can be seen from the above example, in our approach the data is exchanged across the nodes in a recursive pair-wise fashion with a single data transfer operation between each pair of nodes. The number of steps would be equal to $log(n)$ where n is the number of nodes involved in the operation. In the example considered, the number of steps is $log(4)$ which is two. Note that by providing a common set of buffers for both network and intra-node data transfers, we eliminate the extra copying that would otherwise occur.

Overlap benefits: The main benefits of having a shared buffer is the potential of overlap between the network operations and the memory copy operations. By referring to the same Figures 6.4 and 6.5, it can be observed that the data arrived in step 1 of the operation can be copied to the processes' buffers concurrently with network operation in step 2. Thus, we need not wait till all the network operations are completed before the data is copied out of the shared memory segment. For a large scale cluster, this benefit is significant as both the size of the data involved is large and also there are more steps involved in the algorithm.

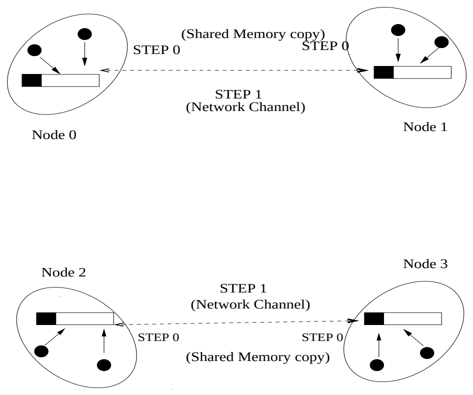


Figure 6.4: Steps 0,1

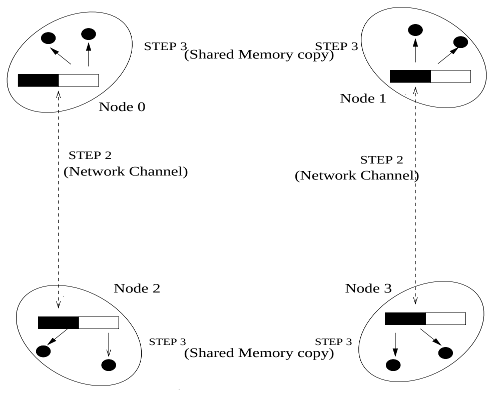


Figure 6.5: Steps 2,3

Implementation Details: The initial implementation step in our approach is creating a shared memory segment per node. This is done by making all the processes local to a node do a mmap of a shared file. After this step, this shared segment is pinned so that data can be accessed directly by the NIC for the network operation. In our design, the shared buffer is pinned by all the processes. This enables all the processes to issue network operations from this memory segment. RDMA is used for network data transfers as it is proven to be an efficient method for inter-node communication. In our implementation, we let one given process issue the network operations from a node. This can be easily accomplished as the processes have local ranks ranging from 0 to p-1 where p is the total number of processes per node. We choose the process with local rank 0 to issue network operations. Note that the addresses of this memory segment are exchanged before the Allgather is initiated. The data notification is done by doing a RDMA write of a one byte flag. These flags are also shared within a node and thus all the processes local to the node can poll for data arrival. This is useful for achieving overlap between network and shared memory copy operations. For synchronizing between the processes within a node another separate set of flags are used.

Differences with existing approaches: Utilizing shared memory for implementing collective communication has been a well studied problem in the past. In [44], the authors proposed to use remote memory operations across the cluster and shared memory within the cluster to develop efficient collective operations. They apply their solutions to Reduce, Bcast and Allreduce operations on IBM SP systems. In our approach we consider a different collective: Allgather which has different communication pattern and present the results on commodity clusters. In [7], the authors implement collective operations over Sun systems. In [52], the authors improve the performance of send and recv operations over shared memory and also apply the techniques for group data movement. Moroever, RDMA collectives have also been studied in MVAPICH e.g. MPI_Barrier, MPI_AlltoAll, MPI_Allgather [20] [36] [40] [39]. These designs are not applicable for the problem discussed in the chapter, as these optimize collectives with one process running per node.

6.2.2 Performance Evaluation

In this section, we compare the performance of the new scheme proposed in the chapter with the already existing approach. The comparison is made by measuring the *Allgather latency* for the two schemes across different message sizes and for two different cluster configurations. The test was conducted for 1000 iterations for each message size. The abbreviations used for the comparison are as follows:

- new: The new shared-memory and RDMA based solution proposed in the chapter.

- original: The original algorithm using MPI point-to-point operations.

Experimental Testbed: We have carried tests on two different clusters:

1) Cluster A: Each node in this testbed has dual Opteron 2.4 GHz processors, 1024 KB L2 cache. They are equipped with MT25204 InfiniBand HCAs with PCI-Express interfaces.

2) Cluster B: Each node in this cluster is a Xeon 2.66 GHz processor with 512 KB L2 cache. Each node is connected with MT23108 InfiniBand HCA with PCI-X interface.

Latency of MPI_Allgather: As the results indicate, our approach outperforms the original approach for the different cluster configurations considered. For Cluster A, we observe benefits upto a factor of 1.47 and 1.39 for 32 and 64 processes as indicated by Figures 6.6 and 6.7 respectively. On cluster B, we observe an improvement by a factor of 1.97 and 1.82 for the considered configurations, 16x2, 32x2. These are shown in Figures 6.8 and 6.9 respectively.

We have also measured the impact of overlap of network operations and shared memory communication on these clusters. The non-overlap approach is implemented by making the processes copy the data from the shared buffers at the end after the network operations are completed. But, for the overlap case the processes copy the data as soon as it arrives and concurrently issue network operations. This is the approach taken in this chapter. With the shared buffer RDMA design proposed the overlap improves the performance of the collective upto 30% for Cluster A and 43% for Cluster B as shown in the Figures 6.10 and 6.11.

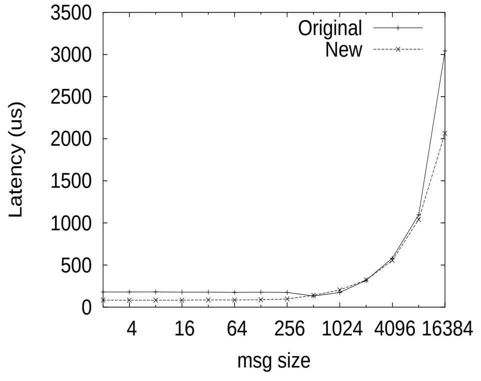


Figure 6.6: Cluster A:(16x2)

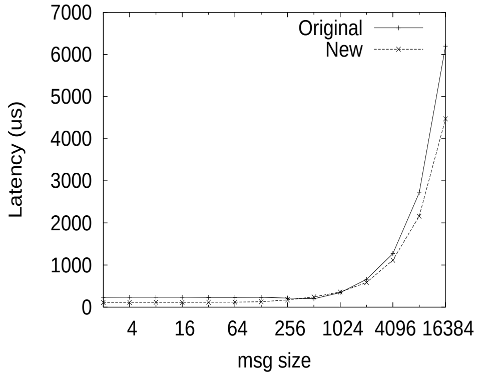


Figure 6.7: Cluster A:(32x2)

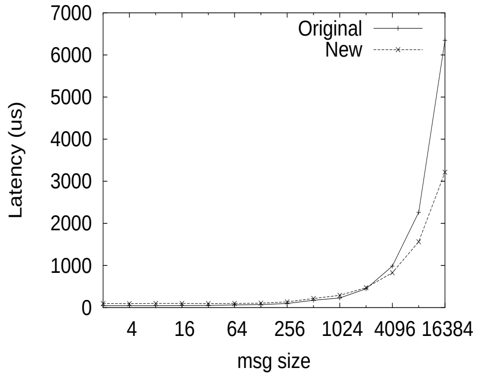


Figure 6.8: Cluster B:(16x2)

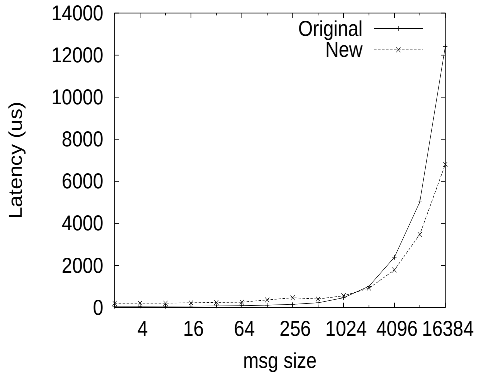


Figure 6.9: Cluster B:(32x2)

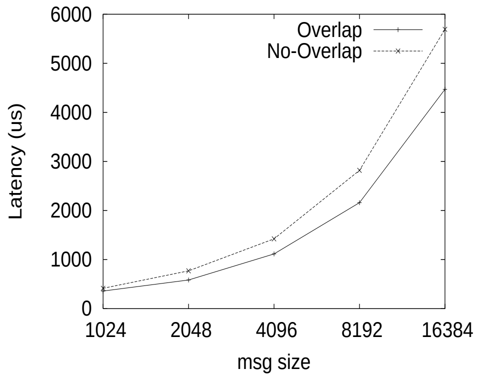


Figure 6.10: Cluster A:(32x2)

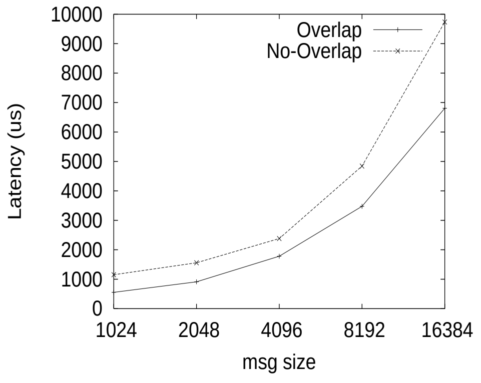


Figure 6.11: Cluster B:(32x2)

6.3 H/W Multicast with Shared Memory

In this section, we present the shared memory collective optimiziations for important collectives like MPI_Bcast. We first present the motivation for using shared memory together with H/W Multicast for designing efficient MPI_Bcast.

6.3.1 Why Hardware Multicast is not enough?

As illustrated in Figure 6.12, when a multicast packet arrives at the NIC, it has to be forwarded to the processes attached to the multicast group specified in the packet header. This process comprises of three steps. First, the NIC has to look up the queue-pairs of the processes which are attached to the multicast group. It then replicates the packets and in the final step DMAs the data to each process's buffer. This cost increases with the increase in the number of processes attached to the multicast group because these DMA operations are sequentialized.

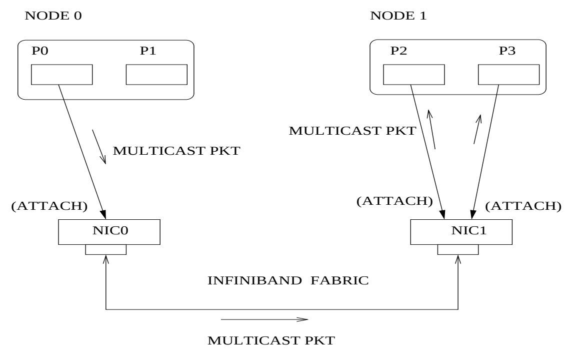


Figure 6.12: Operational principle of IBA Hardware Multicast

Figure 6.13, compares the latency of the MPI_Bcast over hardware multicast for two configurations, 4x2 (meaning two processes per node across four nodes) and

8x1. As shown in the figure, the latency of the former case is significantly higher than the latter. This is because as explained above, in the 4x2 case the DMA of the multicast packets is sequentialized at the NIC resulting in higher latencies. This demonstrates that we need a mechanism to handle multicast based collective efficiently over multi-processor nodes.

Figure 6.14, illustrates the difference between the latency of point-to-point inter-node communication vs the latency of intra-node communication via shared memory channel. As indicated in the figure, the latencies of intra-node channels is an order of magnitude smaller than the inter-node latencies.

These observations motivate us to propose an efficient approach to leverage the shared memory channel for intra-node messages and hardware multicast for inter-node messages to implement MPI_Bcast operation.

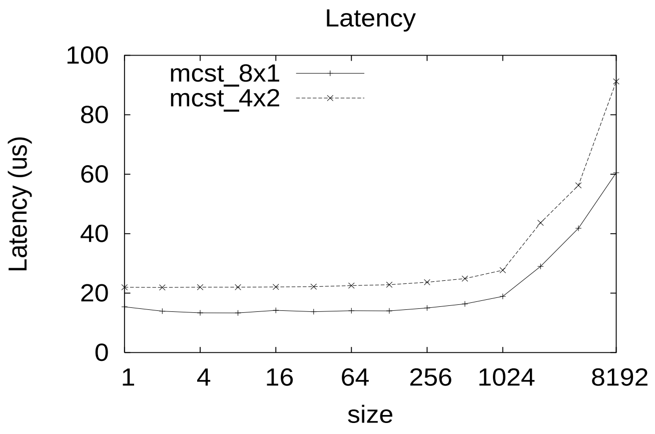


Figure 6.13: MPI_Bcast Latency with IBA hardware multicast on two system configurations of 8 nodes

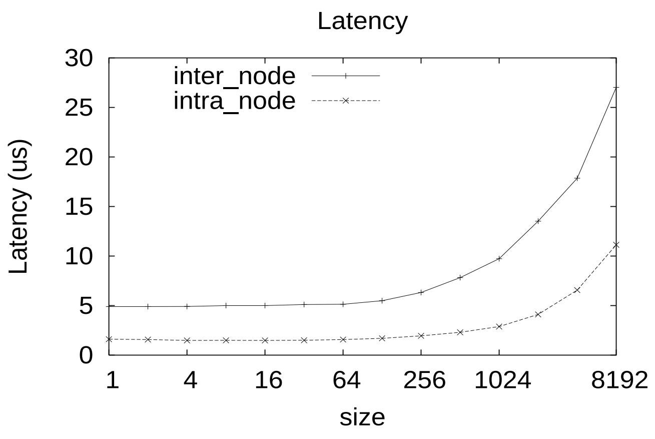


Figure 6.14: Comparison between inter-node and intra-node Point_to_Point Latency

6.3.2 Shared Memory based MPI_Bcast

There are two important design alternatives for implementing collectives utilizing both shared memory channel and hardware multicast.

- Direct multicast to shared memory: In this approach, the multicast packets can be received directly into the shared memory regions across the nodes. After detecting the multicast message arrival, the local processes can copy the message into its own buffer. Though this approach looks promising, detecting the arrival of the multicast packet is a tricky and complex operation. This is because the arrival of the message is notified only to the processes which are attached to the multicast group. As discussed in the earlier section, this approach does not scale well. Another approach would be to write a separate flag following the message. This solution also does not guarantee correctness as there is no ordering guarantee between the UD-based multicast messages and UD or RC based point-to-point flag messages.

- Leader-based Approach: In this approach, a designated process receives hardware multicast messages. It can then distribute the message to the remaining nodes.

We have taken the latter approach in the chapter which is described in the following section.

Leader-based Approach:

In this approach, as indicated in Figure 6.15, the broadcast operation occurs in a hierarchical two-step manner:

1. The Root process posts a multicast message to the multicast group. A set of Leader processes is identified, one Leader per node which receives the multicast packets. These leaders have to be attached to the multicast group at the NIC to receive the packet.

2. The Root after posting the multicast message delivers the message to the participating local processes via shared memory channel. The Leaders on each node do the same after receiving the multicast packet from the Root.

The leaders are identified using the local IDs which are initialized during the initialization of the shared memory channel. In our implementation, the process with local ID zero is chosen as the Leader which receives the hardware multicast packets. It forwards the packets to the other processes by indexing into the local-to-global rank mapping table to determine the other local processes running on

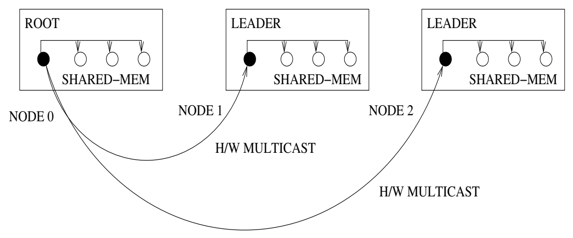


Figure 6.15: Leader-based design

the node. This table is also set up during the initialization phase of the shared memory channel.

It is to be noted that hardware multicast is unreliable in InfiniBand. In [25] we have already proposed Ack based reliability schemes to address the problem. In these schemes, the processes have to send Acks to confirm the receipt of the message. In the approach described here, only the Leaders are required to send the Acks as the other processes receive intra-node messages over shared memory channel.

Dynamic Attach Policy:

The basic design indicated above does not always deliver good performance. This situation occurs when both the hardware multicast packet and a non-Leader process have arrived at the collective call but the Leader process did not arrive. In this case, the non-Leader process has to wait for the Leader to forward the message. This can hamper the performance of the application especially if the chosen Leader always arrive late.

In this section, we describe a Dynamic Attach policy wherein a non-Leader process attaches to the multicast group based on certain conditions. The non-Leader process uses the average wait-time computed across different broadcasts to

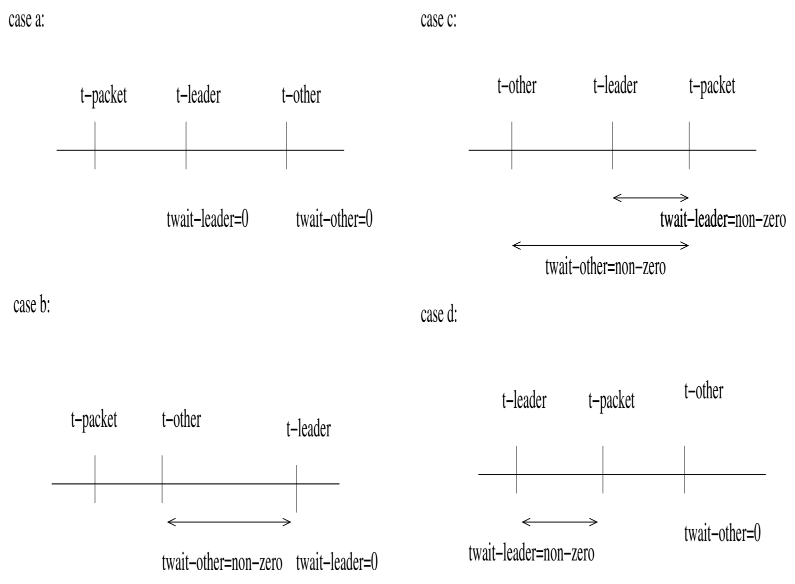


Figure 6.16: Dynamic Attach Policy

make a decision. The average wait-time after a certain number of broadcasts is the total wait-time accumulated so far divided by the number of broadcast operations completed. The wait-time can be easily computed by computing the difference between the time when the message is received and the time when the broadcast is invoked.

Figure 6.16 illustrates the possible cases of the arrival of the packet, the Leader and the other process which is not the leader. We make an assumption that the wait times are much higher than the intra-node latencies. Under this condition, the wait-times for both the Leader and the other process are zero when the packet is unexpected (i.e. arrives before the receiver) as shown in case a. The wait time for the non-Leader process is zero or less than that of the Leader in case d. The only cases for which it has a wait time greater than that of the Leader is in cases b and c.

If we can discount case c, then we can safely say that if the average wait times of the non-Leader process are higher than the Leader process, then case b is the most frequently happening case. In this case, the non-Leader process can attach itself to the multicast group rather than waiting for the Leader process to forward the messages. However, if case c happens more frequently then this assumption would not hold true as in this case both the leader and the other process are waiting for the packet to arrive.

We can overcome the above problem by making the non-Leader process compute the average only when the packet is expected to the Leader, as is the case b. This can be easily implemented by the Leader setting a flag in the shared memory buffer. The average wait time of the Leader process is also stored in a shared memory region. The non-Leader process computes the wait time by starting the timer when it enters the collective and stopping it after receiving the packet. It recomputes the average wait time when the flag is set. It chooses to attach to the multicast group when the difference between its average wait time and that of the Leader's wait time crosses a threshold value. Conversely, if this difference decreases below the threshold, these non-Leader processes detach from the multicast group. To avoid repeated attach and detach overhead, the non-Leader processes detach in a lazy manner i.e. they wait for some number of iterations before detaching. Also, the proposed policy comes into effect only after the number of broadcasts executed cross a threshold. This is to decrease error margins which can occur if the average wait time is based on too few broadcasts.

6.3.3 Performance Evaluation

In this section, we compare the performance of the new scheme proposed in the chapter with the already existing approaches. The comparison is made by running the *broadcast latency* micro-benchmark for all the schemes across different message sizes. To show the benefits of the dynamic attach policy, we have modified the *broadcast latency* micro-benchmark to add skew within the node.

All the different schemes considered and their abbreviations are as follows:

- smp_mcst: The new SMP-Aware solution proposed in the chapter.
- nosmp_mcst: The original solution employing hardware multicast but no shared memory channel as proposed in [25].
- ori_bcst: The point-to-point implementation utilizing network for inter-node communication and shared memory for intra-node communication.

To compare the benefits of the dynamic attach policy, we have run the modified *broadcast latency* program explained above with and without the dynamic attach support.

Another important consideration while running the multiple processes per node is how they are distributed across the nodes in a cluster. In the *cyclic* distribution, consecutive processes are assigned different processors while in *block* distribution they are not assigned different processors until the node has reached its capacity with repect to number of processors.

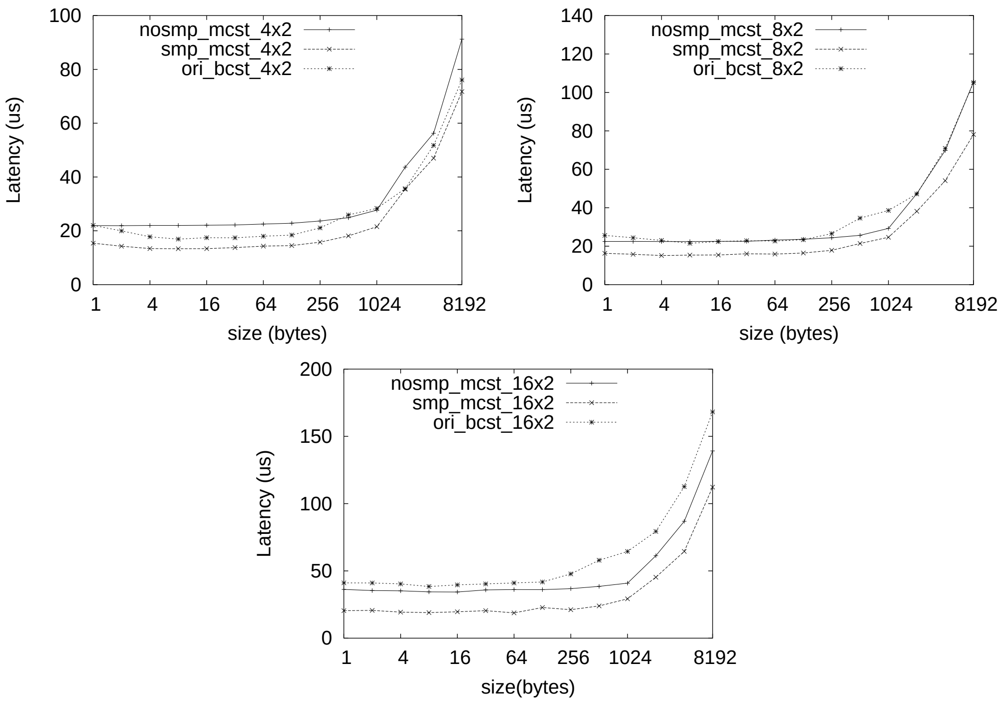


Figure 6.17: Broadcast Latency, Cluster A: (a) 4x2 (b) 8x2 (c) 16x2

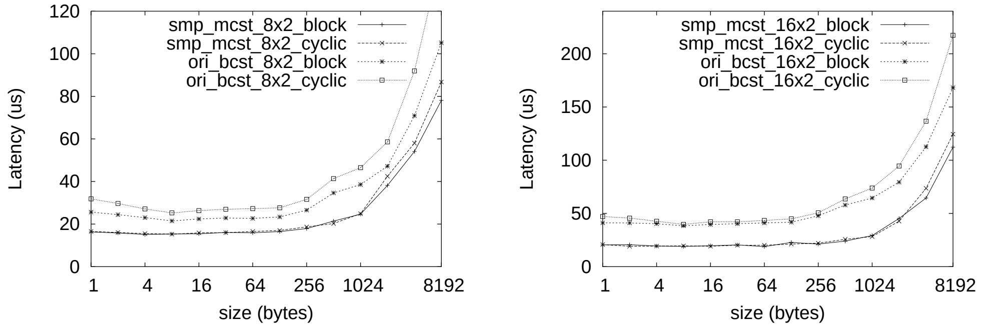


Figure 6.18: Broadcast Latency, Cluster A: (a) 8x2 (b) 16x2

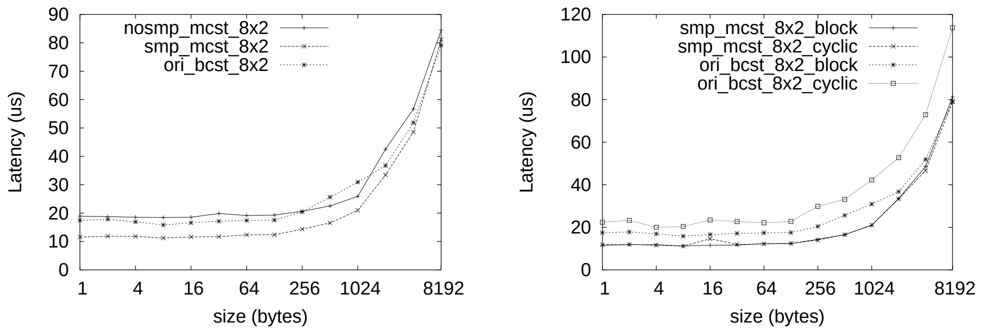


Figure 6.19: Broadcast Latency, Cluster B: (a) 8x2 (b) 16 processes

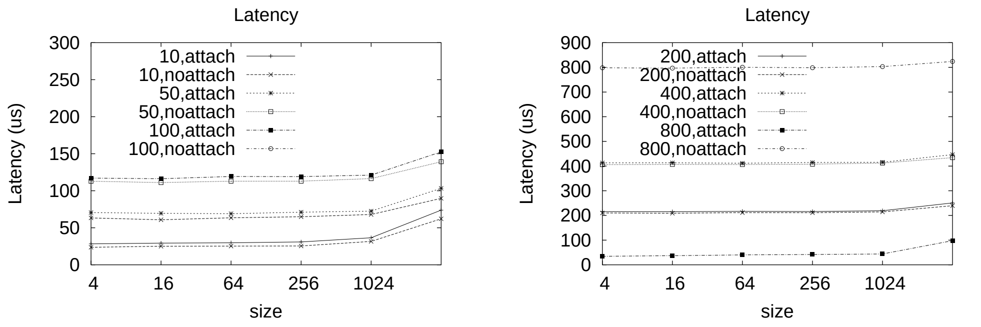


Figure 6.20: Impact of Dynamic Attach on Broadcast Latency when Leaders arrives late

Experimental Testbed:

Cluster A: Testbed cluster consists of 16 SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets. Eight of these are Intel Xeon 3.0 GHz processors and the other are Intel Xeon 2.4 Ghz with 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. We have used InfiniHost MT23108 DualPort 4x HCAs from Mellanox. All nodes are connected to a single Mellanox InfiniScale 24 port switch MTS 2400, which supports all 24 ports running at full 4x speed. The kernel version we used is Linux 2.6. The Front Side Bus (FSB) of each node runs at 533MHz. The compilers we used were GNU GCC.

Cluster B: Testbed cluster consisting of 8 dual Intel Xeon 3.2GHz EM64T systems. Each node is equipped with 512MB of DDR memory and PCI-Express Interface. These nodes have MT25128 Mellanox HCAs with firmware version 5.1.0. The nodes are connected by an 8-port Mellanox InfiniBand switch.

Broadcast Latency:

Broadcast latency is the time taken for a broadcast message to reach every receiver. The test consists of a loop, in which an MPI_Bcast is issued from a root node and the receivers take turns to send back an acknowledgment using MPI_Send. The broadcast latency is derived from time to finish each iteration and the MPI point-to-point latency.

Figure 6.17(a), Figure 6.17(b) and Figure 6.17(c) show the latency of the broadcast operation for 8, 16 and 32 processes, respectively. The number of processes per node is equal to two and we use a block distribution to scatter the processes. The cyclic distribution is considered separately. As indicated in the figure, the performance of nosmp_mcst is worse than the ori_bcst for 8 processes and comparable to ori_bcst for 16 and 32 processes. However, in both cases the smp_mcst performs well delivering performance improvement by a factor of 2.18 and 1.8 when compared to ori_bcst and nosmp_mcst respectively for 32 processes. Similar trends are also observed on Cluster B, Figure 6.19(a).

In Figure 6.18(a) and Figure 6.18(b), we compare the *broadcast latency* for the cyclic and block distributions. The new design proposed in the chapter does equally well both with cyclic and block distributions. However, the ori_bcst scheme performs better in the block distribution vs cyclic distribution. This is because though the messages within a node are transferred over shared memory, the MPI implementation has no global knowledge of local and remote processes. As a result, in the cyclic distribution case the intra-node messages are first delivered and then the inter-node messages. This weakens the hierarchical design which is essential for maximum overlap between the inter and intra node communication. The same results are also obtained on Cluster B, as indicated in Figure 6.19(b).

In the final tests, we have added skew to the *broadcast latency* test by delaying the leaders by variable amounts. We compare the new design proposed in the chapter together with the Dynamic Attach enhancement. The threshold selected was equal to 500 us which is the result of multiplying processors per node and the attach latency which was measured to be around 250 us. As shown in Figure 6.20(a) and Figure 6.20(b), the latency of the operation increases as the delay is being added to the leader. The legend indicates the delay added followed by the scheme selected. As the delay is increased, the original scheme with no dynamic attach keeps incurring the cost but where as the scheme with dynamic attach drops down after crossing the threshold. This occurs for the delay value of 800 us for the leader.

6.4 Summary

In this chapter, we first investigated the semantic advantages of mapping collectives memory semantics of RDMA. We have shown that using RDMA is better compared to using channel semantics of IBA. We have taken AlltoAll as a case study to demonstrate the benefits of RDMA over Send/Recv. We have shown that using RDMA zero-copy can be achieved and it performs about 38% better for the 32 nodes. Also, for more number of nodes, the performance gains are over 33% for small to medium messages

We then described a RDMA and Shared Memory algorithm for MPI_Allgather. MPI_Allgather is an important collective operation which is used in applications such as matrix multiplication and in basic linear algebra operations. The next generation systems feature multi-core architecture enabling a high process count per node. The traditional implementations of Allgather use two separate channels, namely network channel for communication across the nodes and shared memory channel for intra-node communication. Since there is no buffer sharing across these channels, the performance achieved is sub-optimal due to the extra copying of data within a node. This is true especially for a collective involving large number of processes with a high process density per node. In the approach proposed in this chapter, we eliminate the extra copy costs by sharing the communication buffers for both intra and inter node communication. Also, we optimize the performance by allowing overlap of network operations with intra-node shared memory copies.

On a 32, 2-way node cluster, we observe an improvement upto a factor of two for MPI_Allgather compared to the original implementation. We also observe overlap benefits upto 43% for 32x2 process configuration.

Finally, we have designed an efficient integrated mechanism of using IBA's H/W Multicast together with Shared Memory. We proposed a Leader-based mechanism to couple InfiniBand's hardware multicast communication with the shared memory channel to deliver optimal performance to the MPI collectives. This is especially true for the modern systems which feature multi-way SMPs allowing more than one process to run on a single node. Our results show that the scheme proposed in the chapter delivers a performance improvement by a factor of as much as 2.3 and 1.8 when compared to the point-to-point and original solution employing only hardware multicast. Also, on a 4-way NUMA system we observed a performance gain of 1.7 with our designs. We also propose a Dynamic Attach policy to alleviate the performance bottlenecks caused due to process skew.

CHAPTER 7

SKEW TOLERANT ALGORITHMS

In this chapter, we describe how certain collective operations can tolerate process skew present in the application. We describe algorithms that can detect skew behavior and adopt accordingly. The ideas discussed are applied to important collective operations such as MPI_Allreduce and MPI_Barrier.

7.1 Design of the Adaptive Algorithm

In this section we discuss in detail the various design issues involved in the adaptive scheme.

7.1.1 Basic Idea

In the standard gather-broadcast algorithm, the topology of the combining tree used is fixed. As previously discussed, the problem with using a static tree topology is that though the root arrived very early, it has to wait for acks from all of its children. Consider example 1, Figure 7.1 where all the nodes have arrived except for the last node. Further, all the nodes have sent acks to the root and are waiting for the release message from the root. The release message is posted only when the root has received the acks from both of its children. As shown in the figure, it takes two more steps after the last node arrives before the root releases all the nodes.

Now take example 2 of the same figure. The root node now has an extra token which it keeps with itself until one of its children arrived. Soon after receiving the ack from node 1, the root passes on the token to its other child, node 2. Since node 2 has got hold of the token it is the new root and the tree topology changes as shown in Figure 7.1(e) of the figure. As node 5 has already sent its ack, the token now moves on to the last node which has not arrived yet. As soon as this node arrives, it finds out that it has the token and releases everybody. Note that when the last node arrives, the tree looks as shown in *f*.

Using a token we were able to cut down the synchronization delay by two hops. Consider a cluster with large number of nodes where there are multiple levels. In this case, we would cut down the delay by the number of hops equal to the height of the tree. We use hardware multicast of InfiniBand for sending the release message. This enables us to achieve constant synchronization delay for clusters with different node sizes.

The only difference between our scheme and the standard scheme is the use of an extra token apart from the usual acks. However, unlike the earlier scheme where there is a fixed root, the node holding the token becomes the root of the tree in our approach. In the example we have assumed a binary tree topology. We can easily extend this idea to any kind of tree topology. Also, it is possible that any node, irrespective of its position in the tree, can become the owner of the token and hence the root of the tree.

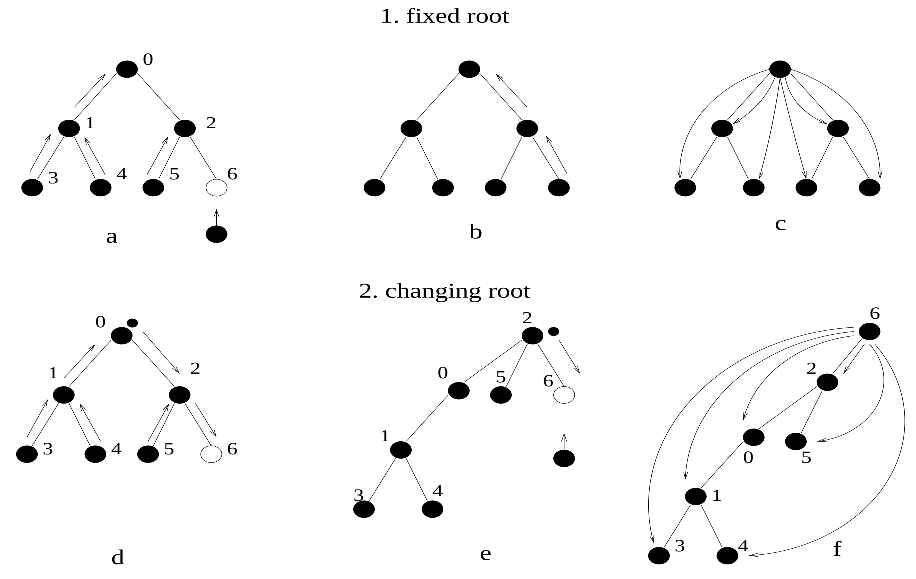


Figure 7.1: Adaptive vs Nonadaptive algorithm

7.1.2 Deciding the root of the tree

We now explain how the token is passed from the current node holding the token to one of its children. The current owner of the token after entering the collective starts polling for all its children. If all of them arrive, then all that it has to do is to post a release message. If one of them has not arrived, then it passes the token to the child who has not arrived yet. To begin with, the token is present with node 0 which is the root of the tree. Note that the token is always passed from the node at a higher level to one of its children in the lower level. In the case of barrier, the token is an opaque object while in the case of allreduce, it carries data. We talk more about the detailed design issues in the following subsections.

The nodes that are not involved in the token transfer behave the same way as in the standard combining tree algorithm. They ack the root as soon as they arrive. The intermediary nodes do the same once they collect the acks from the children. However, all these nodes have to wait for a token or a release message.

Other issues to be resolved are: how does the node know that it holds the token and how is it passed from one node to the other. Another problem faced by this scheme is that of race conditions. Consider this scenario where both the node holding the token and the last of its children have arrived at the same time. In this case, the token from the parent and the ack from the child are exchanged simultaneously. This leads to race conditions which have to be avoided to make the implementation foolproof. Another issue to be discussed is that the hardware multicast in InfiniBand is unreliable. The release message may not reach some of the nodes in which case they are kept waiting. We discuss all these issues in the following detailed design section.

7.1.3 RDMA approach to handle the Token and Acks

We use the remote direct memory access (RDMA) operation of InfiniBand to do the token transfer. In our approach, we have used a one byte flag for both the token and ack in the case of barrier. For allreduce, the token as well as the ack additionally carry a data vector apart from the flag.

By using the RDMA operation the transfer of the token becomes straight forward. All we have to do is to RDMA write into an appropriate location in the remote memory buffer when we want to transfer the token or an ack. But prior to this, we have to exchange the necessary information about the buffers among all the participating nodes. Each node polls on the local memory to check for the token or the acks.

In the allreduce case, each node apart from the leaves has to do some computation before it can pass along the token or the ack. The leaf only includes its data vector in the ack and passes it to its parent. However, if an intermediary node has to pass an ack to its parent or a token to its child, it first computes a data vector from the acks received from the children. It then includes it in an ack or token and passes it to the appropriate node.

7.1.4 Avoiding Race conditions

Race conditions are possible in the implementation of the token-based approach. This is because many combinations of the token-ack transfers are possible based on the skew of the system. Take for example a simple case where both the node holding the token and its last child have arrived at the same time. In this case, the parent passes the token to its child and similarly the child acks the parent. If the child arrived a little late, it would have got the token and it need not have acked the root. The same is the case with the parent.

We take care of this race condition by using a simple technique. We make the child poll both for the token and the release message at the same time even though it has passed its ack to its parent. The parent on the other hand ignores the ack from its child and waits only for the release message. The child after it finds out that it has the token posts a release message immediately.

7.1.5 Reliability

Reliability is another issue which has to be dealt with as we are using hardware multicast of InfiniBand to post the release message in our implementation. We cannot be sure that all the nodes have received the message unless we add some mechanism to make the multicast reliable.

Reliability can be added by making use of a timeout and retransmission mechanism. The root stores all the posted release messages indexed by the count of the barrier or allreduce operation. It stores them until it receives an ack from all the other nodes. A sliding window mechanism can be employed at the root so that it doesn't block waiting for the acks. For more details, please refer to chapter 5.

7.1.6 Flow control

Since we use RDMA based approach to transfer the acks and the token, we need not post any extra descriptors on the remote node in this case. However hardware multicast requires that descriptors be preposted on all the nodes for the message to be received.

The way we do this is initially we post a certain number of descriptors. After every barrier operation we post an extra descriptor to make sure that there are sufficient number of descriptors available always.

7.1.7 Related Work

Work in [13] deals with designing software barriers when there is skew in the system caused by the load imbalances. It explains why such load imbalances occur in a system and how it leads to processes being skewed. In this paper, the authors have come up with a semi-adaptive approach where the nodes which arrive late are placed closer to the root. This approach is different from our design which is totally dynamic. In their approach they use a prediction based scheme based on the recent history of the barriers done. This scheme has certain drawbacks as it works well when the skew pattern remains the same across the barriers. The approach we have taken in this thesis is to use a dynamic tree topology which adjusts itself to the skew. Changing the topology of the tree has also been discussed in [27]. But they have used a counter based combining trees for SMP systems. The scheme discussed in this chapter is targeted for large scale clusters which use InfiniBand as the interconnect. The standard algorithms like pair-wise exchange are explained in [21]. Other algorithms like the gather-broadcast and dissemination are explained in [20]. We have shown in this chapter the limitations of using such algorithms in systems having lot of skew. More details about the hardware multicast are found at [18],[26]. Other interconnects like Quadrics provide hardware multicast[31]. In [53] this feature is used to implement broadcast. NIC-level multicast in Myrinet/GM is studied in [54]. This is different from the hardware multicast of InfiniBand. In NIC-level multicast, the broadcast operation is handled by the NIC instead of the host.

7.2 Performance Evaluation

In this section, we evaluate the different designs of doing barrier and allreduce. We compare the results of our adaptive scheme with the existing approaches for barrier and allreduce. We have used synchronization delay as an important metric in most of our micro-benchmark tests.

The different schemes considered for barrier are as follows:

- adaptive: Our implementation of an adaptive barrier using a combining tree of degree 8. Please refer to the Figure 7.2.

- nonadaptive: a nonadaptive barrier using a combining tree of degree 8.

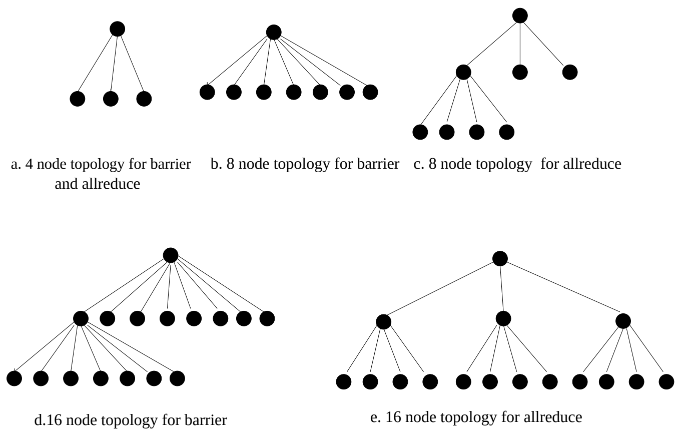

a. 4 node topology for barrier and allreduce
b. 8 node topology for barrier
c. 8 node topology for allreduce
d.16 node topology for barrier
e. 16 node topology for allreduce

Figure 7.2: Combining trees for barrier and allreduce

- dissemination: a dissemination based barrier.

The different schemes considered for allreduce are as follows:

- adaptive: Our implementation of an adaptive allreduce. The exact topology used is as shown in the Figure 7.2.
- nonadaptive: a nonadaptive using the same combining tree as above.
- pair-wise: allreduce based on the pair-wise exchange algorithm.

Experimental Testbed: Our testbed cluster consists of 16 SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets.Eight of these are Intel Xeon 3.0 GHz processors and the other are Intel Xeon 2.4 Ghz with 512 KB L2 cache and PCI-X 64-bit 133 MHz bus. We have used InfiniHost MT23108 DualPort 4x HCAs from Mellanox. All nodes are connected to a single Mellanox InfiniScale 24 port switch MTS 2400, which supports all 24 ports running at full 4x speed. The kernel version we used is Linux 2.4.22smp. The InfiniHost SDK version is

3.0.1 and HCA firmware version is 3.0.1. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM. The compilers we used were GNU GCC 2.96 and GNU FORTRAN 0.5.26.

In the first subsection we use the standard micro-benchmark, *Average latency* to compare the different schemes used to implement the collective operation. Note that measuring the average latency of a collective is meaningful in situations where the skew is negligible. In the following subsection we use the *Synchronization Delay* as a metric to evaluate the above schemes in the presence of varying skew conditions.

Average latency for Barrier and Allreduce: To obtain the average latency of a barrier operation we measure the time taken for each node to perform barrier for a large number of iterations. We compute the average across all the nodes and iterations to obtain the average barrier latency. Average latency of allreduce is computed in the same manner.

In the latency test for the dissemination and pair-wise exchange cases, we can safely assume that all the nodes call barrier or allreduce at same time and also exit the collective call at the same time. However in the tree-based schemes, there is one node which exits earlier than the rest. This could result in skew, but on large clusters this is very small since the latency is averaged across all the nodes.

Figure 7.3(a) shows the average latency results for the barrier operation. As seen from the figure, the adaptive scheme latencies are slightly higher than non-adaptive ones though both schemes use the same tree topology. This is because we pay a penalty of one extra rdma write operation for some iterations. This case happens if the root node has passed the token to the last arriving child but it receives its ack immediately after. This results in one extra hop towards the child before it posts the release message. But, on large number of nodes, this penalty will be too small to be observed.

Though the tree based schemes perform badly for small system sizes, they perform better as the size increases. The reason is the use of a higher degree fan-in for the combining tree and use of hardware multicast which scales very well over large number of nodes.

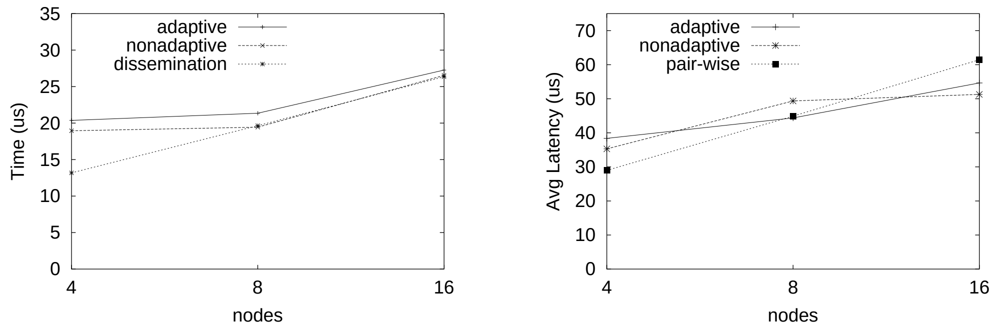


Figure 7.3: Average Latency: (a) MPI_Barrier (b) MPI_Allreduce

Figure 7.3(b) shows the average latency results for allreduce.

Notice that for node sizes of four and 16, the adaptive scheme gives higher latencies. This is because for these topologies, the tree is totally balanced unlike the case with size eight. In balanced tree topologies, it is usually the case that we pay one extra hop of the token as the penalty. This is not the usual case in the unbalanced topology with eight nodes.

Notice that for large number of nodes, both the tree based algorithms perform better or comparably to the pair-wise algorithm in allreduce and dissemination in the case of barrier. More importantly, in the case of barrier, the adaptive scheme performs comparably to the nonadaptive one. In the case of allreduce, for an unbalanced topology the adaptive allreduce fares better.

Synchronization Delay: In this section, we focus on the evaluation of different schemes where unlike the previous case, all the nodes do not arrive at the same time. We artificially delay some of the nodes by making these *target* nodes loop for the specified amount of time. We outline below our approach of measuring *Synchronization Delay* which we use to benchmark different schemes. Following that we present the results of different test cases showing the performance of the schemes under the presence of skew.

Measuring Synchronization Delay Synchronization delay is the time difference between the last node entering the collective to the last node leaving the collective. The test to measure this delay consists of a loop where each of the nodes take turns to send back an acknowledgment using MPI_Send to the node arriving last. Each of the nodes can determine who is the last arriving node from the input parameters given to the test program. The last node starts a timer before calling the operation and stops it as soon as it receives the ack. It computes this time for all the nodes and computes the maximum. This maximum minus the latency of the MPI_Send gives the Synchronization delay for the test.

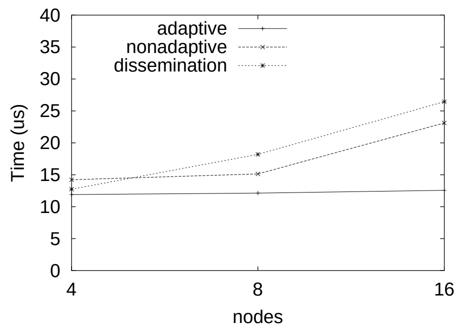
We now give an outline of the different kinds of tests considered in this section. In the first test, we show the scalability of adaptive design compared to the other approaches. In the second test, we consider the scenario where one node always arrived late at the collective. In the third test, we extend this to multiple nodes arriving late. In the fourth and final test, we show instances of skew where the adaptive design does not give optimal performance and explain the approach to solve the problem.

Scalability study: To show the scalability of the adaptive design, we choose one node to be the *target* node and delay this node by the amount equal to twice the average latency of the collective considered. We perform this test for different system sizes. Also, we have chosen the highest rank node as the *target* node.

Figure 7.4(a) shows the results for barrier. From the figure we observe that the synchronization delay is constant across different node sizes for adaptive scheme. For nonadaptive case we see every extra level added to the tree increases this delay by one hop. This is because in the former scheme, the token arrives at the *target* node by the time it called barrier. The combining of acks would have been done and the node only has to post the release message. Since we are using hardware multicast, this phase of the barrier is a fast and scalable operation. In the case of barrier with dissemination, the synchronization delay is proportional to $log(n)$ where n is the number of nodes involved in the barrier.

Figure 7.4(b) is the graph for allreduce. The same reason described for the barrier applies to allreduce as well. However, the synchronization delay, though remaining constant is higher. This is because, the release message now carries the final result vector of the allreduce operation and hence is greater in size. Moreover, the synchronization delay also includes costs of copying and the computation of the final data vector. We have used a vector of size 128 and of type MPI_DOUBLE in all our test cases. The operator used was MPI_PROD.

Single node arriving late: In this test, the *target* node is delayed by different amounts and the synchronization delay is calculated. All the tests are run on



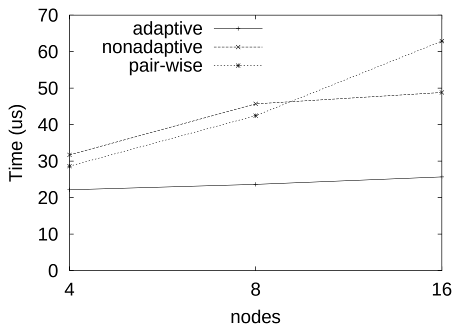
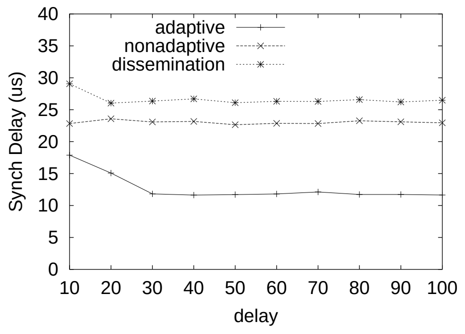


Figure 7.4: Synchronization Delay: (a) MPI_Barrier (b) MPI_Allreduce

16 nodes. Once again we choose rank 15 as the target node. This helps us to understand the behavior of adaptive scheme better.

Figure 7.5(a) shows the results for Barrier. In the adaptive case, as the delay is increased, we obtain smaller synchronization delays up to a point after which it remains constant. This is because initially, the token is not reaching the last node as the delay is not sufficiently high. However, we perform better than the nonadaptive based scheme even in such cases. This is because, in the adaptive design an intermediate node receives both the token from its parent and acks from its children. Thus, less time is spent by the ack of the *target* node to reach the node holding token unlike the nonadaptive design, where it has to travel all the way to the top. For larger delay we observe that the other schemes have the synchronization delay equal to the average latency. For the adaptive design, it is equal to the latency of hardware multicast.

Figure 7.5(b) shows the results for allreduce. The same trend is also observed in the case of allreduce too. However, the values we obtain are higher than in the case of barrier. We have already explained this observation in the previous test case.



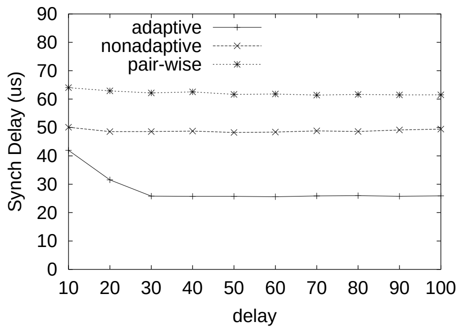


Figure 7.5: Single node arrives late: (a) MPI_Barrier (b) MPI_Allreduce

Multiple nodes arrive late: In this section, we concentrate on evaluating the different schemes when we use multiple *target* nodes. For a cluster of large size, there are many combinations of *target* nodes possible. Also, even after fixing the target nodes, these nodes may arrive in any order. It is not feasible to evaluate the schemes for all possible cases. However, in the case of barrier on 16 nodes we come with three representative cases which include all the skew patterns possible.

Observe that all the children under a given root are identical. That is, no matter what the pattern of arrival of these children is, the passing of the token is dependent upon the last two arriving children. The skew between these nodes decides who gets hold of the token irrespective of the arrival pattern of the other nodes. Thus we can choose two *target* nodes which are the last and second last arriving nodes.

In the topology used in barrier (Figure 7.2) we have two subtrees. Thus, we have three cases to consider. Both the *target* nodes belong to the upper subtree, both belong to the lower subtree, one belongs to upper subtree and the other to the lower subtree.

Figure 7.5(a) can be used to demonstrate the second and the third cases. The way we obtained the results earlier was by skewing only one *target* node where as all the others arrived at the same time. The second *target* node can be assumed to be any one of these other nodes. However, in this case the second *target* node always arrives early. We consider the possibility of the second *target* node arriving early separately. From the figure, we conclude that if there is a reasonable amount of difference between the arrival times of the two *target* nodes, the adaptive barrier does significantly better than the other schemes.

Now, we consider Figure 7.7 to show the case where both the *target* nodes belong to the upper subtree. This also includes the missed case explained above. Note that in this case, the margin of improvement between the nonadaptive based scheme and the adaptive scheme has reduced. This is because the target nodes are now closer to the root than the earlier case.

The allreduce case is more complicated as there are more subtrees to be considered. In this thesis, we show only one set of combinations which we can conclude from Figure 7.5(b). It is the combination in which one of the target nodes is a bottom node and the secondary node is any of the nodes in the tree.

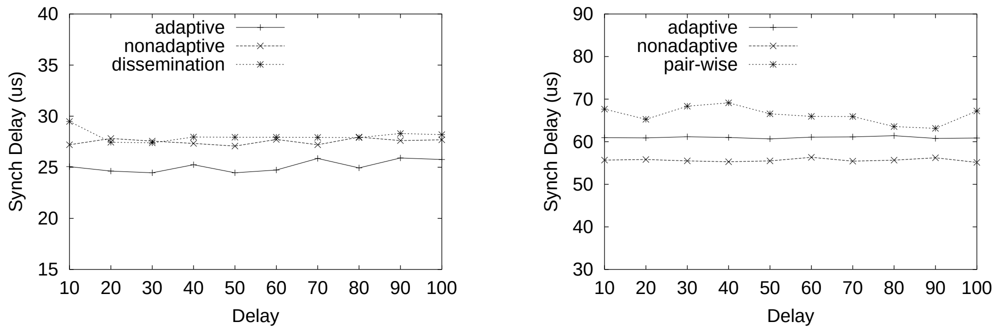


Figure 7.6: Two nodes arrives late: (a) MPI_Barrier (b) MPI_Allreduce

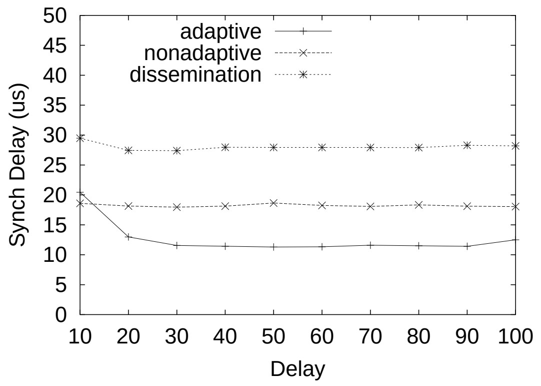


Figure 7.7: Single node arrives late, Barrier

Two nodes arriving equally late:

We have taken this as a special case to show that the adaptive scheme does not give optimal performance under this situation. We also show how this can be improved.

Figure 7.6(a) shows the results obtained for the barrier where the two nodes come late by the same amount of time. Also, the two nodes are the bottom most ones of the tree and belong to separate subtrees under the main root. We used nodes with ranks 1 and 15 as the *target* nodes for this case. The x axis shows the delay of both nodes. As shown in the figure, the synchronization delay remains the same in spite of the increase in the delays of these nodes unlike the earlier figures where it drops down to the hardware multicast latency. The similar behavior is observed in allreduce case (Figure 7.6(b)).

This behavior is because the token is immobile at the root when two of the nodes are arriving late. The root cannot decide to which branch it should pass along the token unless the second last child arrives. But in this case considered, the second last child arrives at the same time as the last child. So, the root is unable to make that decision. We can overcome this problem in our design, by introducing an extra token. Now the root can pass two tokens, one to the second last child and the other to the last child. This idea can be generalized to include multiple tokens. However, introducing too many tokens increases both the complexity of the design and adds more traffic into the system.

7.3 Summary

The standard algorithms like pair-wise exchange, dissemination and gather broadcast do not perform optimally when there is skew in the system. This is because the nodes participating in these algorithms are tightly coupled with each other in all the steps of the algorithms. The design presented in this chapter removes this limitation by making the tree topology adapt dynamically to the changing skew scenarios. We have used an adaptive root mechanism where the last arriving node becomes the root of the tree if the skew is sufficiently large. We have used hardware multicast in the release phase of our algorithm. From the results we have shown that the design presented in this chapter scales very well as the number of nodes increases. We obtained a synchronization delay of 12 us in the case of Barrier which is close to the hardware multicast latency. This number is constant for varying system sizes. Using our scheme we reduce the synchronization delay by a factor of 2.28 for Barrier and by a factor of 2.18 for allreduce. We also show that the adaptive design performs comparably when there is no skew. Different skew scenarios were discussed in the chapter and we show that our adaptive design either performs better or comparable to the existing skew conditions.

CHAPTER 8

PERFORMANCE AND MEMORY SCALING

In this chapter, we explain the benefits and drawbacks of using the different transport methods of IBA. To elucidate these distinctions, we use MPI_Alltoall as the case study. We describe the different approaches of implementing Alltoall over IBA transports and use these designs to benchmark performance and memory usage. Subsequently, we illustrate why a new IBA primitive is desired and explain its benefits.

8.1 Designing Alltoall over UD

As explained in Section 2.2, RC and UD transports exhibit different performance and scalability trends. To completely evaluate the trade-offs between these two modes of transport, we choose *AlltoAll* as the benchmark to reflect the benefits of choosing RC vs UD. *AlltoAll* over RC is already integrated into MVAPICH [29]. We now present two new designs of *AlltoAll* over UD. The first one is the direct algorithm as explained in Section 2.1. This algorithm exclusively uses UD for communication. The second one explained below uses both forms of transport RC and UD for data transfer.

Hybrid algorithm This algorithm is the variant of the pair-wise exchange algorithm which is used for large messages. As discussed in Section 2.1, this algorithm consists of (n-1) steps where in each step a given process exchanges messages with a different peer process. This algorithm can be modified such that from a given number of steps, k steps can use RC transport for messaging and the remaining (n-1-k) steps use UD. In our implementation, the parameter k is the same across all the processes to ensure homogeneity.

We now outline the implementation issues common to both the algorithms.

Communication over UD: As explained in earlier sections, UD allows for MTU-size packet transfer between any two processes. All the information the processes need to know are the QP specific information such as QP numbers, Qkeys and Address-vectors for routing. The first step required in the communication of a message is the fragmentation into MTU-sized chunks. These chunks are then posted to the UD-QP after filling in the descriptors with the address information of the destination process. To receive the message the reverse process is required, assembling of the chunks of the data into the receive buffer. Also, note that since UD does not guarantee order of delivery of the data, each packet is marked with a sequence number so that the data is placed in the correct manner.

Reliability mechanism: Since UD is unreliable, we need host-level reliability to ensure retransmission if any packet loss is encountered. We use a Nack based approach to tackle the issue. A Nack based approach is used because the probability of packet drop in a cluster is relatively very low. Infact, we observed no packet drop in our experiments. However, we need to buffer the packets in case the packet loss is experienced. Owing to the property of *AlltoAll* that at any given time only two operations can be outstanding, reliability protocol is simplified. Thus, at any point a maximum of two messages are buffered.

8.2 Evaluating Performance and Memory Trade-offs

In this section, we explain the tests conducted and the analysis of the results. We first briefly describe our experimental testbed.

Each node of our testbed has two 3.6 GHz Intel processor and 2 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes are equipped with MT25208 HCAs with PCI Express interfaces. A Flextronics 144-port DDR switch is used to connect all the nodes. The operating system used was RedHat Linux AS4.

Performance and Memory Trade-Offs: We now present the benchmark-level evaluation and analysis of the various designs and performance trade-offs studied.

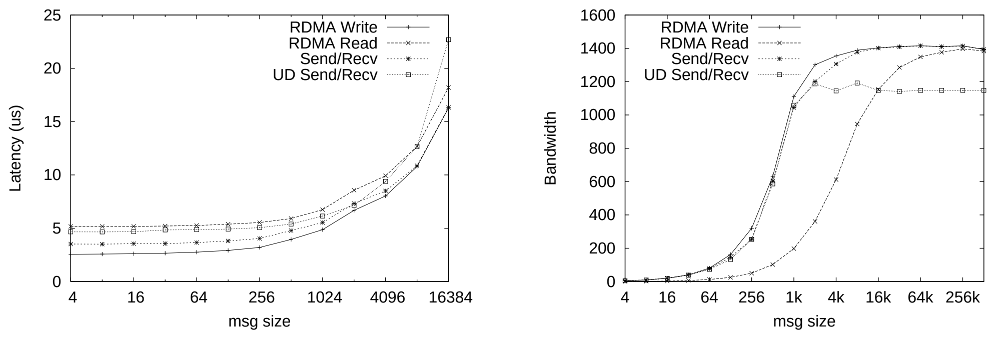
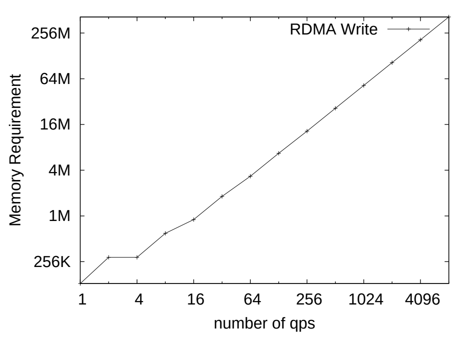


Figure 8.1: Basic Performance (a) Latency (b) Bandwidth



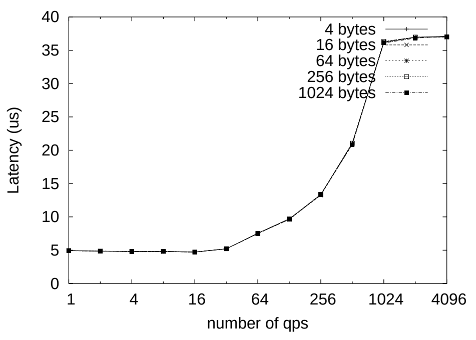


Figure 8.2: (a) Queue Pair Memory requirements, (b) RC Multiple QPs

We first do a micro-benchmark evaluation and then choose *AlltoAll* as the Benchmark to compare RC and UD.

Micro-Benchmark Level Evaluation:

Basic Evaluation of IBA Transports: Figures 8.1 (a) and (b) illustrate the performance of RC and UD in latency and bandwidth. As indicated in the figures, RC performs better than UD, both in latency and bandwidth. However, for MTU messages, both perform almost equally well for send/recv case. But, for large messages, UD performance drops compared to RC. This can be seen in the latencies of messages over MTU and in the bandwidth numbers. This can be attributed to the per-MTU overhead and lack of efficient pipelining.

Memory Scaling of RC: Figure 8.2(b) explains the principal drawback of using RC i.e. growing memory requirements with increase in number of connections (QPs). As can be seen from the figure, there is an upward linear trend of memory usage with the number of connections. We assign one QP per connection.

NIC caching effects: To explain the caching effects, we constructed a simple ping-pong latency benchmark with multiple connections present between the two nodes. Messages are exchanged in a round-robin manner and their latencies measured. As indicated in Figure 8.2(b), the latency of messages start increasing from number of connections (QPs) equal to 32. This is because the NIC has to DMA the QP context information every time this is flushed from the cache.

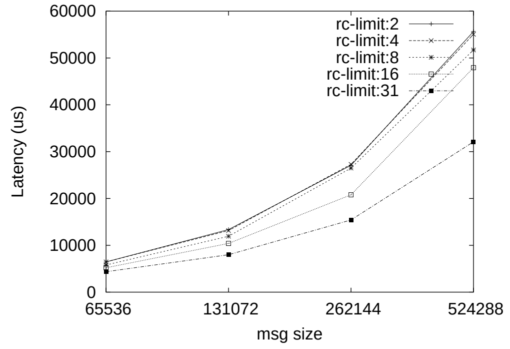


Figure 8.3: (a)Hybrid algorithm

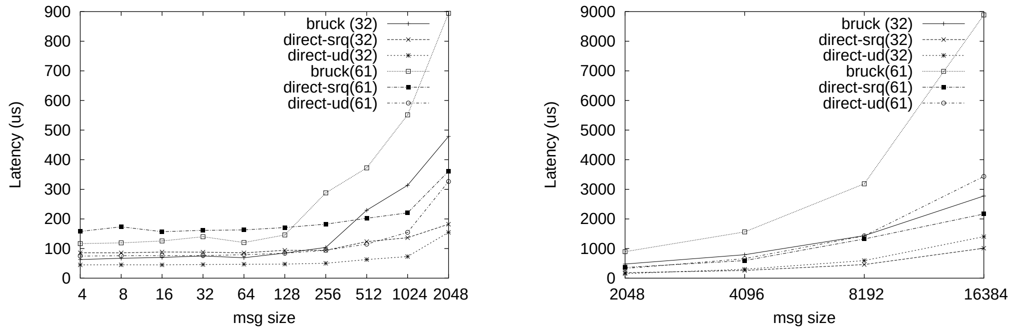


Figure 8.4: UD Collectives (a) Small messages (b) Medium messages

Benchmark Level Evaluation (AlltoAll): In this section, we first evaluate the performance of the *AlltoAll* latency for three different algorithms : *AlltoAll* over UD as discussed in this chapter, *AlltoAll* over SRQ in MVAPICH, *AlltoAll* using Bruck's Algorithm in MVAPICH. The legends in the graph being direct-UD, direct-SRQ and Bruck respectively. We then show the trade-offs in performance vs resource utilization using the hybrid algorithm proposed in this chapter.

Comparison of RC and UD: As shown in Figure 8.4 for short messages UD performs better than RC. This is in tune with the micro-benchmark evaluation above. However, as the message size increases, the performance of RC is better than UD. This is because of the better bandwidth capabilities of RC. Also, UD performs better 37% better than Bruck's which is the currently used algorithm for *AlltoAll* short messages.

Also, as seen from the figures the latency of small messages of direct-UD is almost double than that of direct-SRQ for both 32 and 61 processes. This can be mainly attributed to the NIC caching overhead. As Figure 8.2 indicates, the caching overhead comes into play after 16 connections have been established. Since every message incurs this overhead, the total overhead increases linearly with the size of the process group. Thus, if 1 us overhead is incurred per message, for 61 processes it would be around 60 us. For UD, this overhead is not present because only one QP is used which can be easily cached. Further, our approach performs better by around 36% compared to Bruck.

Hybrid Algorithm - Choosing Performance 'vs' Memory: We now present the results from hybrid algorithm proposed in this chapter which allows choosing varying number of connections for the *AlltoAll* operation. The important point to note is that using this algorithm the upper layer can configure the maximum number of connections to use depending on the resource availability on the system. Figure 8.3 which shows the performance of *AlltoAll* with varying number of RC connections used and UD. The "rc-limit" corresponds to the number of RC connections used in the algorithm. As shown in the graph, increasing the number of RC connections betters performance but increases resources as shown in Figure 8.2 (a).

8.3 New IBA primitives

As described in earlier Chapter 6, RDMA semantics closely match with that of many collective operations, especially MPI_Alltoall. Also, RDMA is required to support one-sided operations. But, RDMA support is provided for IBA: RC, UC and IBA: RD. No support is provided for UD transport. This makes scaling RDMA impractical on ultra-scale clusters. Moreover, implementing RD is a very hard problem and no vendor has provided support for it in the current implementations. Moreover, the semantics of RD restricting that only one message can be outstanding between two end points is not suitable for MPI. Therefore, we need RDMA support over UD. In this section, we first describe how such support can be added via software. We demonstrate that though this approach provides the necessary functionality, better performance can be obtained by supported this primitive in hardware.

8.3.1 RDMA Emulation over UD

There are two broad design choices of emulating RDMA over UD, using zero-copy or copy-based. However, using zero-copy over UD becomes difficult to accomplish as UD does not guarantee in-order delivery of packets. Also, as shown in earlier sections, zero-copy over channel semantics serializes network transactions. Due to these two reasons, we follow a copy based approach for our emulation. However, a major drawback of this approach is cache pollution which is absent in zero-copy protocols.

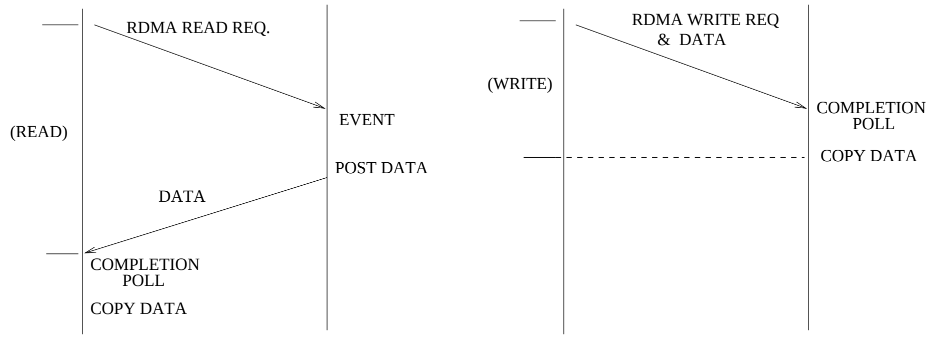


Figure 8.5: Emulating RDMA over UD

We now describe the protocols used for copy-based RDMA over UD emulation.

RDMA Read emulation over UD: As Figure 8.5 indicates, designing RDMA Read requires three steps. In the first step, the issuer sends the RDMA Read Request together with the address of the remote buffer. Once the remote side receives the request, it posts the data corresponding to the address back to the process issuing the request. The issuing process receives the data and copies into the correct destination buffer.

There are two important issues with this basic protocol. The first one pertains to the asynchronous progress of the operation. A RDMA Read request arriving at the remote end necessitates the early service of this request to achieve lowest latency possible. In the case of the native RDMA support over RC, this is taken care of by a dedicated processor on the NIC. To achieve this over UD, we need a separate host-level service thread which manages these tasks. The service thread can either poll for the incoming requests or block until a request arrives. The polling mode is not suitable because the service thread would contend with the application threads for cpu. To make the communication progress least obtrusive to the application's computation, we need to use the blocking mode.

Another issue is that the service thread can execute in the user address space or as a part of the kernel in the form of kernel-level thread. However, in either cases, blocking mode needs to be used to avoid contention of CPU resources. Our implementation is based on the user-level implementation. The application is responsible for spawning its own service thread.

IBA allows for the blocking mode or the event notification mechanism. The RDMA request message has a special bit set. This bit called as the solicitation flag triggers a completion notification event once the message arrives at the other end. After the notification is generated, the service thread is woken which follows-up on the remaining procedure. In our case, this would be posting the data corresponding to the address specified in the request packet. If the address region is not-pinned, a registration operation is required on the appropriate buffers before the descriptor is posted. Also, note that a UD operation supports transfer of only one MTU of data per descriptor. Thus, a chain of descriptors is required in case the data exceeds one MTU size. As explained in the next section, the event notification overhead can be avoided for RDMA Write operations.

RDMA Write emulation over UD: We now consider the operation of a Write over UD. As shown in the Figure, in the first step the issuing process posts the RDMA write request to the remote process. The request in this case carries both the remote address and also the data payload. Once the request reaches the remote end, the remote process copies the data into the address specified by the issuer.

Please note that another important issue which needs to be addressed in detail is that of security. The integrity of the process memory has to be carefully safeguarded so that only authorized processes can access the memory window. Since our focus is on the performance aspects of the emulation, we assume security as not an issue in our study. As opposed to a Read operation, Write operation can be optimized not to have an event generated. This is because the host process can do a lazy copy of the data into its memory window whenever it checks for the completion of the operation. In our case, an operation is complete if the associated IBA completion entries have been successfully polled from the completion queue.

RDMA and One-sided operations in other Protocol stacks: Using RDMA for One-sided operations have been studied in depth. Various designs for one-sided operations have been proposed in [50, 51, 35, 47]. However, these operations have been designed over RC support of IBA which poses scalability problem for large cluster sizes. Yelick et al have studied one-sided operations in the context of UPC and Titanium [45, 48, 12, 49]. In [4], the authors propose techniques to achieve overlap of computation and communication using one-sided operations. However, our focus is on addressing the scalability issues of RDMA one-sided network primitives for clusters comprising of large number of nodes. The benefits of RDMA over UD have been explored in [43]. The authors explore the potential of using RDMA over UD for increased performance, reduction in hot-spot effects and caching problems. Our focus is on studying the benefits of emulating RDMA over UD for one-sided operations.

8.3.2 Performance Evaluation

In this section, we explain the tests conducted and the analysis of the results. We first briefly describe our experimental testbed.

Each node of our testbed has two 3.6 GHz Intel processor and 2 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes are equipped with MT25208 HCAs with PCI Express interfaces. A Flextronics 144-port DDR switch is used to connect all the nodes. The operating system used was RedHat Linux AS4.

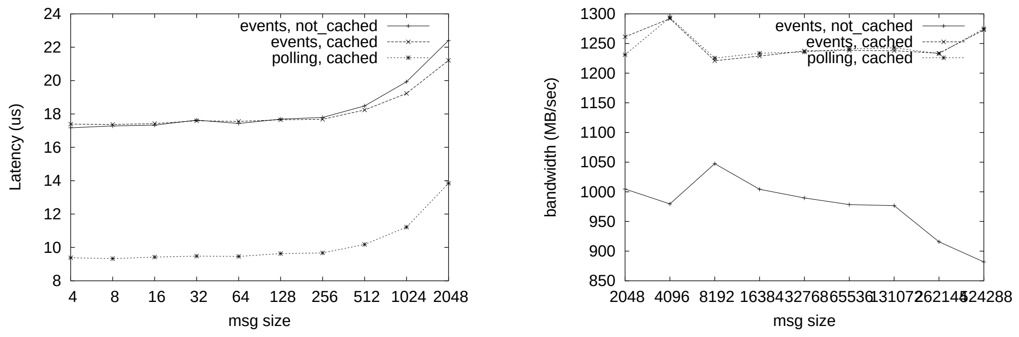


Figure 8.6: Emulating RDMA Read over UD (a) Latency (b) Bandwidth

RDMA Emulation over UD: We now present the results of the emulated Read and Write operations over UD. Figure 8.6 explains the results of the Read latency tests. We explore three cases, when the destination buffer is in the cache, out of the cache and when polling is used instead of events. Our motivation of

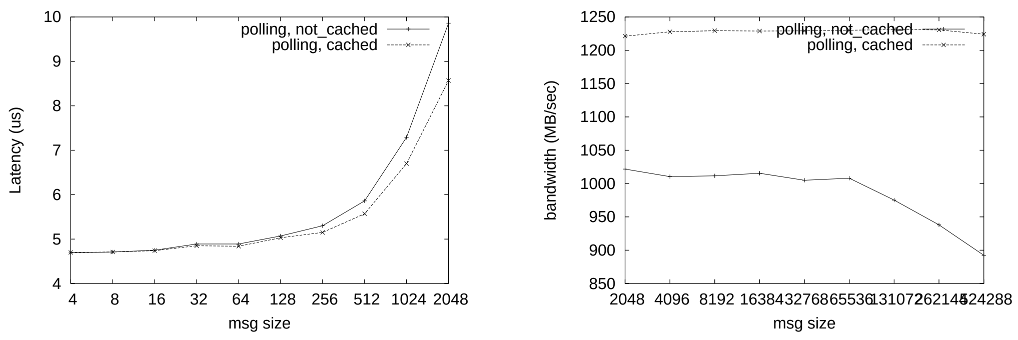


Figure 8.7: Emulating RDMA Write over UD (a) Latency (b) Bandwidth

taking into caching is because the objective of one-sided operations is to overlap computation and communication. In such scenarios it is very likely that the communication buffer might not be cached while the application is computing on a different chunk of data. As the results indicate caching determines the performance of both the latency and also bandwidth. For applications which have buffers un-cached most of the time, this would lead to performance degradation and most importantly cache pollution. Also, as the figure indicates there is around 8-9 us of event overhead compared to polling. Write operations, as indicated in Figure 8.7, are relatively less costly as there is no event overhead. But, like Read operations, caching plays a dominant role.

Also compared to RC performance (Figure 8.1), RDMA emulation performs poorly. This is especially true for one MTU message, RDMA Read which has a factor of more than three performance degradation. In the light of above observations, we conclude that emulating RDMA over UD poses performance limitations and the native support of RDMA over UD is desired.

8.4 Summary

As clusters increase in size, the performance and scalability of the communication subsystem becomes the key requirement for achieving overall scalability of the system. In this context, the efficiency of collective operations is especially important as they are the widely used communication operations in different programming models like UPC, MPI-2, etc. Thus, they have to be designed harnessing the capabilities and features exposed by the underlying networks. In some cases, there is a direct match between the semantics of the operations and the underlying network primitives. InfiniBand provides two transport modes: (i) Connection-oriented *Reliable connection (RC)* supporting Memory and Channel semantics and (ii) Connection-less *Unreliable Datagram (UD)* supporting Channel semantics. Achieving good performance and scalability needs careful analysis and designing of communication operations based on these options.

In this chapter, we evaluated the scalability and performance trade-offs between RC and UD transport modes. We have taken AlltoAll as a case study to demonstrate the benefits of RDMA over Send/Recv and shown the performance/memory trade-offs over IB transports. Our experimental results show that the UD-based AlltoAll performs 38% better than Bruck's algorithm for short messages and up to two times better than the direct AlltoAll over RC. Since IBA does not provide RDMA over UD in hardware, we emulated the same in our study. Our results show a performance dip of up to a factor of three for emulated RDMA Read latency as compared to RC, highlighting the need for hardware based RDMA operations over UD. We thus emphasize the need for extending the IBA specification to allow for support of RDMA over UD.

CHAPTER 9

ARCHITECTURE DRIVEN OPTIMIZATIONS

In this chapter, we focus on designing/optimizing collective primitives on the Multicore architectures. The collectives focused in this chapter are MPI_Allgather, MPI_Bcast, MPI_Allreduce and MPI_Alltoall. The main objective is to understand the effects of the multicore specific characteristics in different collective operations and use the insights obtained to guide the design process for achieving optimal performance.

9.1 Communication in Multicores

The advent of multicore processors presents several opportunities and challenges for designing efficient collective operations. For efficient design of these operations, it is important to first understand the communication trends over the multicores. In this section, we conduct different experiments involving both inter- and intra-node communication followed by those involving intra-node communication. Our rationale to focus also on intra-node optimizations is that the core count is expected to increase rapidly in the coming years. Thus, depending on the applications' characteristics, significant amount of communication time can be spent in exchanges messages within the node. Our first test measures the impact of L2 cache sharing in Intel Clovertown architecture.

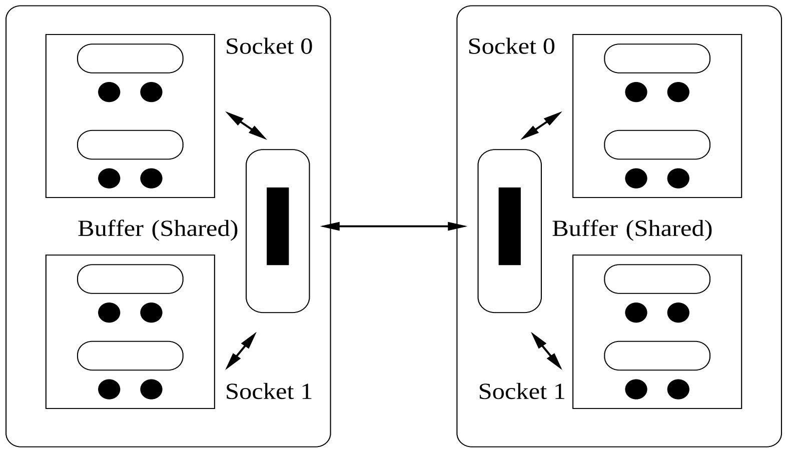


Figure 9.1: (a)Send-Recv latency test

L2 cache sharing in Intel Clovertown: To elucidate the impact of L2 cache sharing, we perform a simple micro-benchmark evaluation. In this test, a message of size 16384 bytes is exchanged between a set of nodes as shown in Figure 9.1(a). The message buffers on each node are shared across all the cores. In this way, one process per node initiates the message transfer and all the other processes on the nodes copy data directly from this buffer. The test is run by varying the number of processes on each node to a maximum of eight processes per node. The results of this benchmark are shown in Figure 9.2(a). As seen from the graph, latency remains constant for processes upto four on each node and then steps to a higher level. This corresponds to core count of eight in the Figure. In our test all the four processes are scheduled on the same socket. This is clearly due to the impact of L2 cache sharing and faster cache-coherency protocol within each

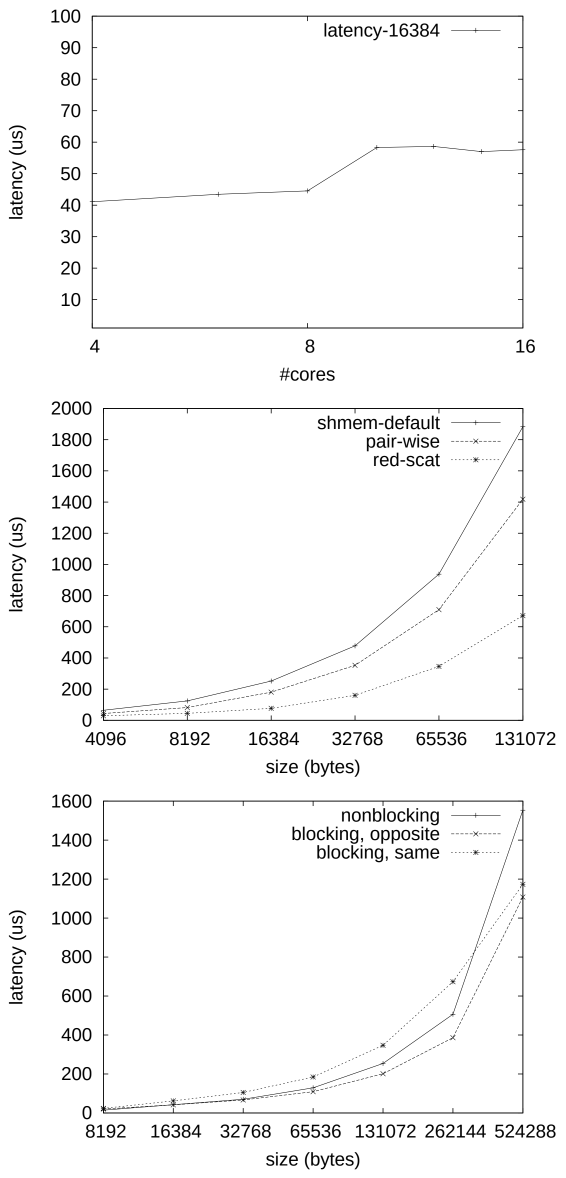


Figure 9.2: (a)Effects of L2 cache sharing (b)Intra-node Reductions (c)Scheduling communication over HyperTransport

socket of the Clovertown platform. This attribute potentially benefits MPI level broadcast operations where the same data is sent to multiple processes.

Reduction operations within a node: One of the primary methods of doing reduction within a node has been to use shared memory segments to copy the data into and do the necessary arithmetic. For example, in MPI_Allreduce all the processes copy the data into a shared buffer, followed by reduction by one of the process. After the reduction, the data is copied back into the respective receive buffers. Though this technique is optimal for short messages, it does not scale for large messages. This is shown in the Figure 9.2(b) where a more distributed approach of point-to-point reduce-scatter followed by allgather performs the best. This approach fares better than the pair-wise exchange and the basic shared memory method though there are extra memory copies involved. Thus, more optimized shared memory approaches need to be investigated. Further, the cache and memory hierarchies play a crucial role in determining the optimality of these approaches.

Scheduling data movement over HyperTransport: In this test, we conduct different tests across two pairs of communicating processes. Two of the processes are running on one socket while the other two are on a different socket. Each process in a pair sends and receives message from the other process separated by one hop distance over the HyperTransport. However, there are three different methods of exchanging these messages. The first method is to use non-blocking send/receive functions provided in MPI. The other method is to use the blocking modes of these function. Even with this case, the sends can be issued by both the pairs in one direction over HyperTransport followed by those from other direction.

Or, to utilize the full bidirectional bandwidth of HyperTransport, sends can be issued in opposite directions. We test all these cases and notice that the last method which utilizes the full bidirectional bandwidth performs the best, Figure 9.2(c).

In the remaining sections of the chapter, we explore all these issues and use the insights obtained to optimize different collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall.

9.2 Multicore Optimizations

In this section, we discuss our proposed optimizations for collective functions: MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. For these collective operations, we focus on large message optimizations in this chapter.

9.2.1 Optimized MPI_Bcast and MPI_Allgather

As discussed in earlier sections, the multicore clusters provide for faster sharing of data within a node. Further, data transfer operations can be overlapped within a node and across the nodes. These useful insights can be channeled to develop efficient MPI_Bcast and MPI_Allgather collective operations.

From section 2.1, the default algorithm used to design MPI_Bcast for large messages is a scatter operation followed by an all-to-all broadcast or MPI_Allgather of the data. This algorithm performs optimally for clusters having one process per node. However, when multiple processes are launched on the same node this algorithm needs to be modified.

The basic building block of our re-designed broadcast algorithm is the shared memory based all-to-all broadcast of the data using a ring topology. In this design, we allow for a single process per node to perform all the inter-node collective communication while the remaining processes are involved in intra-node communication. The process which does all the inter-node communication is called the *leader*. In the new design for MPI_Bcast, the collective communication can be broken down into three steps. In the first step the root process copies the message to the shared buffer and the leader of the node scatters the message to all the other leaders. A binomial recursive halving algorithm is used to scatter the data. In the second step all the leaders are involved in inter-node ring communication as shown in Figure 9.3. This step corresponds to the MPI_Allgather of the data. The number of steps involved would be n-1 where n is the number of leaders. The third step involves a copy of the data from the network buffer to the user buffers. This step can be done after the entire network communication is done or can be overlapped. We have taken the latter approach by overlapping network transfers with shared memory copying of data. In our approach the network buffers are shared across all the processes to facilitate direct copy of data.

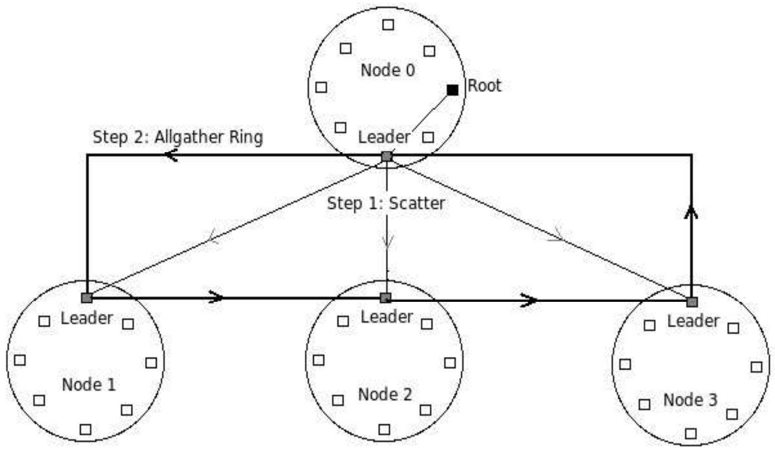


Figure 9.3: Broadcast

Impact of L2 cache sharing of Intel Clovertown: In this section, we try to understand the behavior of the above mentioned algorithm on an Intel Clovertown multicore cluster. As discussed above, after each communication step of the ring, the local processes copy the data from the shared buffer to the respective receive buffers. Since the processes share L2 Caches and they copy the same chunk of data from shared memory after each step, several interesting observations can be made about the behavior of the collective algorithm. Firstly, the intra-node communication overhead would be greatly reduced due to the high bandwidth of L2 caches, provided the communication buffers are appropriately cached. Secondly, with the increase in the number of cores participating in the collective, we do not expect the latency of the collective to change till the intra-node copy time exceeds the inter-node latency. For example, if the extra cores participating in the operation belong to the same socket, the latency should marginally increase as the caches with in the socket are efficiently shared. Only when the cores added are derived from two different sockets do we expect an appreciable jump to occur in the latencies. Thus, we observe that the shared L2 Cache architecture effectively reduces the contention on the memory especially when concurrent network transactions occur together with intra-node message copying. We experimentally evaluate these effects in the performance evaluation section of the chapter.

9.2.2 Optimized MPI_Allreduce

In the default shared memory algorithm for allreduce, only a single core does the reduce, while other cores are idle during that time. In this section, we propose two algorithms for allreduce collective that utilize the computational power of multiple cores inside a node. The first algorithm is a very basic approach of parallelizing the computation. In the second algorithm, we propose mechanisms to improve the performance even further by leveraging the architecture of multicores.

In the basic approach of doing MPI_Allreduce, the computation is delegated to all the cores in a straight forward manner. We illustrate this with an example consisting of four processes within a node, Figure 9.4. As described in the figure, each process copies its data into different blocks of the shared memory region based on their respective ranks(step 0 of the figure). Please note that after this step the respective data blocks will be located closer to the process either in the cache in the case of Intel Clovertown and AMD Opteron or also in memory as in the case with NUMA Opterons. In the next step, each block of shared memory is divided into four sub-blocks corresponding to the number of processes. Now, every process operates on one sub-block from every block of the shared region. For example process 0 reduces sub-blocks 0a, 0b, 0c and 0d to the memory space of 0a and the result is sub-block 0R. Similarly process 1 reduces sub-blocks 1a, 1b, 1c and 1d resulting in sub-block 1R and so on. In step 3, each process copies the entire zero block, containing sub-blocks 0R, 1R, 2R, 3R, to their own local memories.

Now, a close observation leads to the fact that the last algorithm can suffer from cache/memory access contention. This is because all the processes follow the same order of accessing the blocks in the shared memory region. Thus, we design another allreduce algorithm that keeps the parallelism of the previous one, but in addition utilizes a cyclic approach to eliminate the drawback of the previous algorithm. In this approach, steps 0 and 1 are same as above except that the reduction and gathering of the data is done in a cyclical manner. In the example considered, process 0 reduces in order 0a, 0b, 0c, 0d whereas process 1 reduces in order 1b, 1c, 1d, 1a. Similarly, in step 3, each process copies the sub-blocks of the result into its local memory in a cyclic fashion. This important optimization is expected to yield significant benefits in modern multicore machines having distributed cache and memory hierarchies.

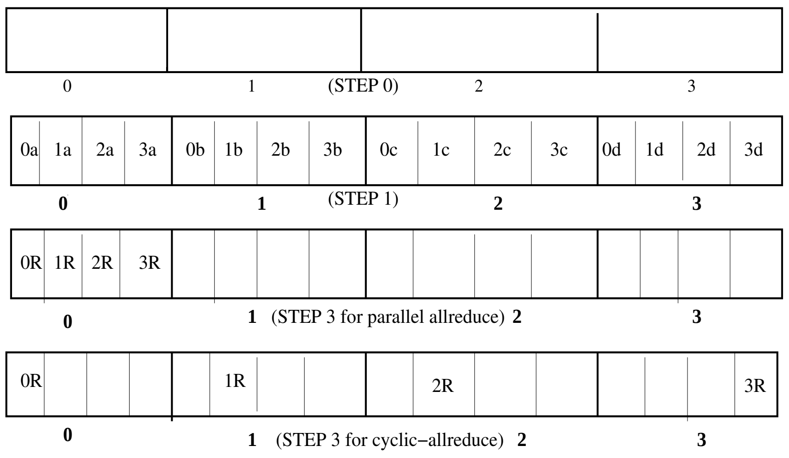


Figure 9.4: Optimized shared memory allreduce schemes

9.2.3 Optimized Shared Memory MPI_Alltoall

In this section, we explain the scheduling of the operations for MPI_Alltoall over the HyperTransport links of Opteron multicore architecture. The basic idea behind our approach is to completely utilize the bidirectional bandwidth of the links interconnecting the different cache and memory modules of the Opteron architecture.

As explained in the earlier section 9.1, we convert all the non-blocking operations into their blocking counterparts. Since MPI_Alltoall uses a pair-wise algorithm with every process talking to every other process, there would be $n(n-1)$ pairs in total. For each of the pairs, communication schedule is constructed in such a manner that the data transfer happens in both directions at any given point in time. This schedule would keep the links busy through out the whole operation.

9.2.4 Related Work

Utilizing shared memory for implementing collective communication has been a well studied problem in the past. In [44], the authors propose using remote memory operations across the cluster and shared memory within the cluster to develop efficient collective operations. They apply their solutions to Reduce, Bcast and Allreduce operation. The authors evaluate their approach on IBM-SP systems. In this chapter, we evaluated the SMP-based collectives taking into account the multicore aspect of the clusters. Specifically, we developed algorithms and gained insights into their performance on the latest platforms from Intel and AMD. Optimizing collectives on hierarchical architectures have also been studied by Bull [9] in the context of Itanium based NUMA machines. In our work, we focused on the Intel and AMD based multicore systems.

In [7, 38], the authors implement collective operations over Sun systems. However, in their design and evaluation they do not take into account any cache effects. Instead, their designs focus on optimal utilization of memory bandwidth by taking advantage of multiple memory banks. In [52], the authors improve the performance of send and recv operations over shared memory and also apply the techniques for group data movement.

9.3 Performance Evaluation

In this section, we compare the performance of the new designs of MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. We first explain the testbed used in our evaluations.

We have conducted our tests on an Intel cluster of 64 nodes for inter- and intra-node collectives. Each node is a dual Intel Xeon Clovertown processor with quad core possessing a shared 4MB cache for two cores. The nodes are interconnected by MT25208 HCAs in DDR mode. The operating system used is Redhat Linux AS4. For opteron, we did not have access to a large scale cluster. For intra-node collectives, we evaluated our designs on a quad-socket, dual core opteron. Each of the cores contains a 1MB L2 cache. We have evaluated our designs on varying configurations such as 32X4 (i.e. 32 nodes with 4 process on each node), 32X8, 64X4 and 64X8.

9.3.1 MPI_Bcast Latency

We measure the MPI_Bcast Latencies and the results are as shown in Figures 9.5(a) and 9.5(b) for 64 nodes of the cluster. In each of the graph we compare our new design with the original design. As discussed earlier the original design employs a point-to-point reduce-scatter followed by allgather without taking into account the locality of the processes. The results for original design show different performance when the processes are scheduled in a block and cyclic fashion. As can be seen the new design proposed in this paper gives better performance for any distribution of processes. We obtain speed-ups of upto 1.78 and 1.90 for the two configurations.

In the second experiment, we measure MPI_Bcast latency on four Intel nodes by gradually increasing the cores participating in MPI_Bcast from each node. We go from one to eight processes per node since each node has 8 cores. The results can be seen in the figure 9.5(c). In each of the configuration, two experiments were conducted with processes being scheduled in a node in socket-block or socket-cyclic manner. As can be seen from the figure 9.5(c), initially for 4X1 both the tests show the same latency since both the configurations are same. As we increase the processes to 4X2 the socket-cyclic method of scheduling causes greater latency than socket-block. This is because in socket-block method the processes share L2 Cache and hence the copy time is negligible which is not the case in socket-cyclic. This continues till 4X4 configuration and then after that both the methods produce the same latency as both the methods have processes on each of the sockets. Please note that in figure 9.5(a), the new scheme performs poorly for message less than 1 MB. This is because by default the processes are distributed in socket-cyclic manner in our test cases.

9.3.2 MPI_Allgather Latency

Figure 9.7 shows the results for MPI_Allgather latency on 64 node Intel Clovertown cluster. Depending on the message size, the new design outperforms the current one by a factor of by a factor of upto four on 64x8 configurations.

The benefits of MPI_Allgather can also be seen from the Matrix Multiplication Kernel which uses MPI_Allgather collective communication. The application is

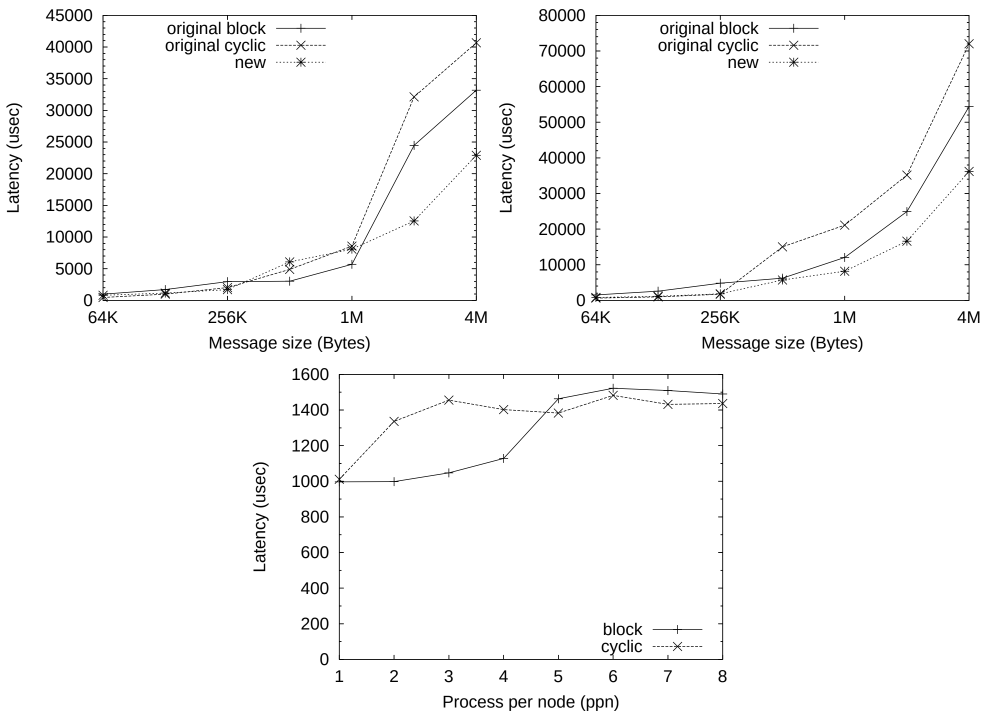


Figure 9.5: Bcast Latency:(a) 64X4 (b) 64X8 (c) Effect of Cache on 4 nodes

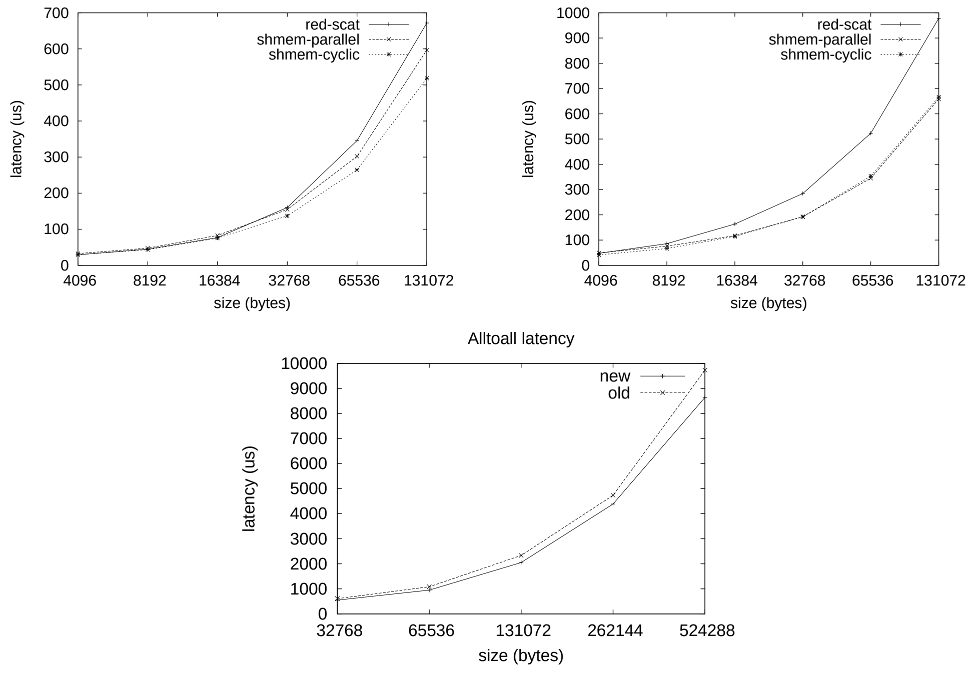


Figure 9.6: (a)Allreduce:Opteron, NUMA (b)Allreduce:Intel Clovertown (c)Scheduling in Alltoall

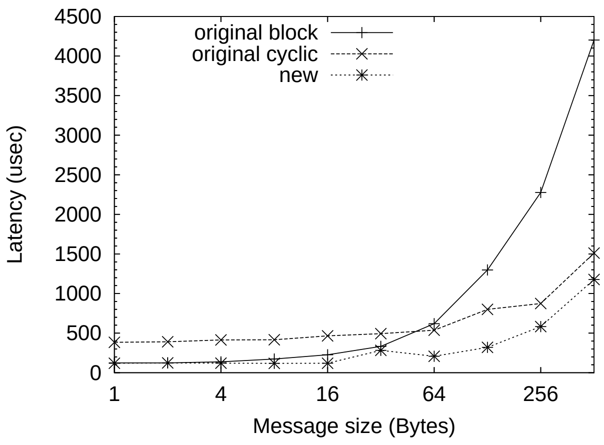


Figure 9.7: Allgather Latency

run with different matrix sizes (double data type matrix elements) starting from 16x16 and upto 512x512. Figure 9.8 shows the performance gain with increasing message sizes. For 512x512 sizes, we improve the performance by as much as three times compared to the default implementation.

9.3.3 MPI_Allreduce Latency

We have evaluated the performance of the newly proposed parallel algorithms with the existing allreduce algorithms for the two different multicore architectures. Figures 9.6(a) and 9.6(b) show the performance of existing and the two new allreduce algorithms for the AMD Opteron and Intel Clovertown architecture respectively. The legend “shmem-parallel” corresponds to the basic algorithm where all processes accesses different blocks in the same order. The “shmem-cyclic” legend refers to the optimized parallel algorithm where the processes access the different blocks in a cyclical manner. From the figures it can be seen that the shmem-cyclic

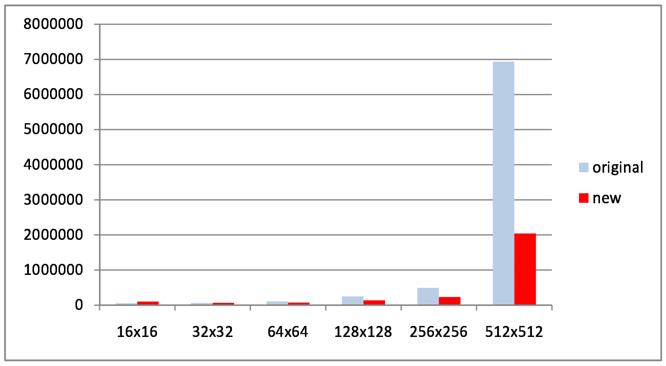


Figure 9.8: Matrix Multiplication

performs the best of all the three. We improve the performance by 23% on NUMA opteron and by 32% on Intel Clovertown. One interesting observation to make is that on Intel Clovertown, both shmem-cyclic and shmem-parallel perform identically where as on NUMA Opteron, shmem-cyclic performs the best. Owing to the NUMA and the different HyperTransport links connecting the different cache and memory modules, AMD Opteron fares better due to increased level of parallelism and lower contention.

9.3.4 MPI_Alltoall Latency

In this section, we compare the proposed optimization of scheduling the send/receive operations over the bi-directional HyperTransport links connecting different sockets in AMD Opteron. The results are described in Figure 9.6(c). As shown in the figure, the new optimization improves the latency of MPI_Alltoall by around 15% compared to the original scheme. We expect to see much higher performance gains as the core counts increase per socket.

9.4 Summary

Optimizing MPI collective communication on emerging multicore clusters is the key to obtaining good performance speed-ups for many parallel applications. However, designing these operations on the modern multicores is a non-trivial task. On the other hand, modern multicores such as Intel's Clovertown and AMD's Opteron feature various architectural attributes resulting in interesting ramifications. For example, Clovertown deploys shared L2 caches for a pair of cores where as in Opteron, L2 caches are exclusive to a core. Understanding the impact of these architectures on communication performance is crucial to designing efficient collective algorithms. In this chapter, we have systematically evaluated these architectures and used the insights to develop efficient collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. Further, we characterized the behavior of these collective algorithms on multicores especially when concurrent network and intra-node communications occur. We also evaluated the benefits of multicore Opteron's architecture for intra-node MPI_Allreduce compared to Intel's Clovertown. The optimizations proposed in this chapter reduced the latency of MPI_Bcast and MPI_Allgather by 1.9 and 2.5 times respectively. For MPI_Allreduce, our optimizations improve the performance by as much as 33% on the multicores. Further, we see performance gains upto three times for the matrix multiplication benchmark on 512 cores.

CHAPTER 10

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

Our work described in the dissertation has been incorporated into several releases of MVAPICH1 and MVAPICH2 distribution. Since its first public open source release in 2002, more than 680 organizations have downloaded the software. The current version of MVAPICH1/MVAPICH2 is 1.0.

MVAPICH supports a wide class of collective algorithms over different interfaces such as OpenFabrics Gen2, uDAPL. It supports both InfiniBand and RDMA interconnects such as iWARP. The support for InfiniBand includes multiple vendors: Mellanox, Qlogic. Recently, it has successfully scaled to over 30,000 cores on Ranger Cluster at Texas Advanced Centre for Computation in Austin. It currently powers the third fastest supercomputer at the SGI/New Mexico Computing Applications Center (NMCAC).

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

In this thesis, we addressed the problem of providing a Scalable and High Performance Communication Subsystem for MPI Collectives over InfiniBand Multicore Clusters. Several key issues addressed in the proposal were a) Communication Protocols, b) Data Transfer Methods, c) Skew Tolerance, d) Performance and Memory Scaling and e) Multicore Optimizations. The basic approach to resolve and handle these issues was to understand the capabilities of InfiniBand Network and gain insights into Multicore architectures. InfiniBand provides advanced network primitives such as H/W Multicast and RDMA over different Transport Methods. The designs proposed in this thesis leverage these primitives to provide efficient communication mechanisms. Moreover, collective optimizations proposed in the thesis take into account the underlying architecture of the current/future generation Multicores. The summary of the research contributions is explained in the following sections of the chapter.

11.1 Research Summary

The research conducted in this dissertation is summarized below.

11.1.1 New Communication Protocols

In Chapter 5, we described how to take advantage of hardware multicast in InfiniBand to implement Reliable Multicast operation in MPI. We proposed a *Reliability Mechanism* that overcomes the problem of dropping packets by the network. To improve performance of Reliable Multicast, we have used a sliding window based design removing much of the processing from communication critical path. To further balance and reduce processing overhead, we proposed techniques such as the *co-root* scheme and *delayed ACK*.

The chapter also discusses efficient schemes of dynamically constructing communicators with hardware multicast support in InfiniBand. The basic idea used in the schemes is to overlap the group construction with the progress of the application. The Multicast Pool and the Lazy approaches proposed in the chapter move most of the overhead of multicast group creation out of the critical path of the application execution. We have evaluated these schemes together with the Basic scheme and found that the Multicast Pool performs the best of all the three followed by the Lazy scheme. Multicast Pool and Lazy schemes improve the Effective Latency by a factor of 4.9 and 3.8 respectively.

11.1.2 Efficient Data Transfer Methods

Chapter 6 investigated the semantic advantages of mapping collectives memory semantics of RDMA. We have shown that using RDMA is better compared to using channel semantics of IBA. AlltoAll is taken as a case study to demonstrate the benefits of RDMA over Send/Recv. We have shown that RDMA can achieve zero-copy and it performs about 38% better for the 32 nodes. Also, for more number of nodes, the performance gains are over 33% for small to medium messages.

The chapter also describes a RDMA and Shared Memory algorithm for MPI_Allgather. The traditional implementations of Allgather use two separate channels, namely network channel for communication across the nodes and shared memory channel for intra-node communication. Since there is no buffer sharing across these channels, the performance achieved is sub-optimal due to the extra copying of data within a node. This is true especially for a collective involving large number of processes with a high process density per node. In the approach proposed in the chapter, the extra copy costs are eliminated by sharing the communication buffers for both intra and inter node communication. Also, further optimizations are done by allowing overlap of network operations with intra-node shared memory copies. On a 32, 2-way node cluster, we observe an improvement upto a factor of two for MPI_Allgather compared to the original implementation. We also observe overlap benefits upto 43% for 32x2 process configuration.

In Chapter 6, efficient integrated mechanism of using IBA's H/W Multicast together with Shared Memory is described. We proposed a Leader-based mechanism to couple InfiniBand's hardware multicast communication with the shared memory channel to deliver optimal performance to the MPI collectives. Our results show that the scheme proposed in the chapter delivers a performance improvement by a factor of as much as 2.3 and 1.8 when compared to the point-to-point and original solution employing only hardware multicast. Also, on a 4-way NUMA system we observed a performance gain of 1.7 with our designs. We also propose a Dynamic Attach policy to alleviate the performance bottlenecks caused due to process skew.

11.1.3 Achieving Skew Tolerance

The standard algorithms like pair-wise exchange, dissemination and gather broadcast do not perform optimally when there is skew in the system. This is because the nodes participating in these algorithms are tightly coupled with each other in all the steps of the algorithms. The designs presented in Chapter 7 removes this limitation by making the tree topology adapt dynamically to the changing skew scenarios. We have used an adaptive root mechanism where the last arriving node becomes the root of the tree if the skew is sufficiently large. The algorithm uses hardware multicast in the release phase of our algorithm. From the results we have shown that the design presented in this chapter scales very well as the number of nodes increases. We obtained a synchronization delay of 12 us in the case of Barrier which is close to the hardware multicast latency. This number is constant for varying system sizes. The proposed scheme reduces the synchronization delay by a factor of 2.28 for Barrier and by a factor of 2.18 for allreduce. We also show that the adaptive design performs comparably when there is no skew. Different skew scenarios were discussed in the chapter and we show that our adaptive design either performs better or comparable to the existing skew conditions.

11.1.4 Issues in Performance and Memory Scaling

In Chapter 8, we evaluated the scalability and performance trade-offs between RC and UD transport modes. We have taken AlltoAll as a case study to demonstrate the benefits of RDMA over Send/Recv and shown the performance/memory trade-offs over IB transports. Our experimental results show that the UD-based AlltoAll performs 38% better than Bruck's algorithm for short messages and up to two times better than the direct AlltoAll over RC. Since IBA does not provide RDMA over UD in hardware, we emulated the same in our study. Our results show a performance dip of up to a factor of three for emulated RDMA Read latency as compared to RC, highlighting the need for hardware based RDMA operations over UD. We thus emphasize the need for extending the IBA specification to allow for support of RDMA over UD.

11.1.5 Architecture Driven Optimizations for Multicores

Modern multicores such as Intel's Clovertown and AMD's Opteron feature various architectural attributes resulting in interesting ramifications. For example, Clovertown deploys shared L2 caches for a pair of cores where as in Opteron, L2 caches are exclusive to a core. Understanding the impact of these architectures on communication performance is crucial to designing efficient collective algorithms. In Chapter 9, we have systematically evaluated these architectures and used the insights to develop efficient collective operations such as MPI_Bcast, MPI_Allgather, MPI_Allreduce and MPI_Alltoall. Further, we characterized the behavior of these collective algorithms on multicores especially when concurrent network and intra-node communications occur. We also evaluated the benefits of multicore Opteron's architecture for intra-node MPI_Allreduce compared to Intel's Clovertown. The optimizations proposed reduced the latency of MPI_Bcast and MPI_Allgather by 1.9 and 2.5 times respectively. For MPI_Allreduce, our optimizations improve the performance by as much as 33% on the multicores. Further, we see performance gains upto three times for the matrix multiplication benchmark on 512 cores.

11.2 Future Work

The following are some of the future ideas that can be pursued along the research presented in this thesis.

- **Multicore based Collectives:** The research presented in this thesis along Multicore optimizations is a starting point. With the increasing core count on a node, future Multicore architectures present several challenges. Especially, the topics of interest would be optimizing collective communication over on-chip interconnection fabric. Since, the architectural trend is shifting towards using point-to-point links rather than shared buses, several shared memory collective algorithms need to be re-visited and optimized accordingly.
- **Asynchronous Progress for Collectives:** We plan to evaluate the impact of skew in applications and propose mechanisms to alleviate such effects. One strategy would to use one of the cores for collective offload. This srategy is effective for single root collectives such as MPI_Bcast, MPI_Reduce.
- **Non-Blocking Collectives:** With the increasing scale of the next generation clusters, process skew is a critical factor affecting the performance of parallel applications. Strategies proposed in this thesis address the problem while using Blocking collectives. However, efficient schemes can be designed by changing the semantics of MPI Collectives to Non-Blocking. Apart from

changing the present day algorithms, this would entail modifying the applications to leverage the new semantics.

- **Framework for Tuning and Optimization:** As discussed in this thesis, designing a collective algorithm over a given architecture is a non-trivial problem. An equally challenging task is to automatically choose the correct algorithm for the provided platform. Automatic tuning and optimization of collective algorithms is an open area of research.

- **Integrated Evaluation:** The emerging clusters pose several challenges with respect to varying communication topologies, differing network architectures etc. This opens up research avenues along conducting comprehensive evaluation of applications over these cluster architectures. Applications such as POP [22], an ocean modelling application, extensively uses the MPI_Allreduce operations. Another application of interest is the FFT kernel [3]. FFT uses MPI_Alltoall for carrying out transpose operations.

BIBLIOGRAPHY

[1] Multicast collectives. http://vmi.ncsa.uiuc.edu.

[2] AMD. http://www.AMD.com/opteron.

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

[4] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap Support. In *IPDPS*, 2006.

[5] A. Bermudez, R. Casado, F. J. Quiles, T. M. Pinkston, and J. Duato. Evaluation of a Subnet Management Mechanism for InfiniBand Networks. In *Proceedings of ICPP*, 2003.

[6] A. Bermudez, R. Casado, F. J. Quiles, T. M. Pinkston, and J. Duato. On the InfiniBand Subnet Discovery Process. In *Proceedings of Cluster Computing*, 2003.

[7] M Bernaschi and G Richelli. Mpi collective communication operations on large shared memory systems. In *Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop*, 2001.

[8] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions in Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[9] BULL. http://www.bull.com/hpc/.

[10] D. Buntinas, D. K. Panda, and R. Brightwell. Application-bypass broadcast in mpich over gm. In *International Symposium on Cluster Computing and the Grid (CCGRID '03)*, May 2003.

[11] H. A. Chen, Y. O. Carrasco, and A. W. Apon. MPI Collective Operations over IP Multicast. In *Workshop PC-NOW 2000*, 2000.

[12] Costin Iancu and Parry Husbands and Paul Hargrove. Hunting the overlap. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005*, 2005.

[13] Alexandre E. Eichenberger and Santosh G. Abraham. Impact of load imbalance on the design of software barriers. In *Proceedings of ICPP*, pages 63–72, 1995.

[14] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[16] Rinku Gupta, Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

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

[18] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.1. http://www.infinibandta.org, November 2002.

[19] Intel. http://www.intel.com.

[20] Sushmitha P. Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *EuroPVM/MPI*, Oct. 2003.

[21] Vipin Kumar, Ananth Grama, Anshul Gupta, and George karypis.

[22] LANL. The Parallel Ocean Program (POP). http://climate.lanl.gov/Models/POP.

[23] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, March 1996.

[24] J. Liu, A. Mamidala, and D. K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2004.

[25] Jiuxing Liu, Amith R.Mamidala, and Dhabaleswar K. panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Proceedings of IPDPS*, 2004.

[26] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. http://www.mellanox.com, July 2002.

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

[28] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[29] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html, January 2003.

[30] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Int'l Symposium on Computer Architecture(ISCA '84)*, 1984.

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

[32] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, February 2002.

[33] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputin*, 2003.

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

[35] Rajeev Thakur and William Gropp and Brian Toonen. Minimizing synchronization overhead in the implementation of mpi one-sided communication. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004)*, 2004.

[36] Amith R.Mamidala, Jiuxing Liu, and Dhabaleswar K. panda. Efficient Barrier and Allreduce InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms . In *Proceedings of Cluster Computing*, 2004.

[37] J. C. Sancho, A. Robles, and J. Duato. Effective Strategy to Compute Forwarding Tables for InfiniBand Networks. In *Proceedings of ICPP*, 2001.

[38] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large-scale SMP's. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[39] S. Sur, U.K.R. Bondhugula, A.R. Mamidala, H.-W. Jin, and D. K. Panda. High performance RDMA based All-to-All Broadcast for InfiniBand Clusters. In *(HiPC)*, 2005.

[40] S. Sur, H.-W. Jin, and D. K. Panda. Efficient and Scalable All-to-All Exchange for InfiniBand-based Clusters. In *International Conference on Parallel Processing (ICPP)*, 2004.

[41] TACC. http://www.tacc.utexas.edu/resources/hpcsystems/.

[42] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In *Euro PVM/MPI*, 2003.

[43] Breaking the connection: RDMA deconstructed. Rajeev Sivaram and Govindaraju, R.K. and Hochschild, P. and Blackmore, R. and Piyush Chaudhary. In *HOTI*, 2005.

[44] V Tipparaju, J Nieplocha, and D K Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *International Parallel and Distributed Processing Symposium, 2003*, 2003.

[45] UC Berkeley/LBNL. Berkeley upc - unified parallel c. http://upc.lbl.gov/.

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

[47] A. Vishnu, G. Santhanaraman, W. Huang, H.-W. Jin, and D. K. Panda. Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits. In *HiPC*, 2005.

[48] Wei-Yu Chen and Costin Iancu and Katherine Yelick. Communication optimizations for fine-grained upc applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005*, 2005.

[49] Wei-Yu Chen and Dan Bonachea and Jason Duell and Parry Husbands and Costin Iancu and Katherine Yelick. A performance analysis of the berkeley upc compiler. In *17th Annual International Conference on Supercomputing (ICS)*, 2003.

[50] Weihang Jiang and Jiuxing Liu and Hyun-Wook Jin and Dhabaleswar K. Panda and Darius Buntinas and Rajeev Thakur and William Gropp. Efficient implementation of mpi-2 passive one-sided communication on infiniband clusters. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004)*, 2004.

[51] Weihang Jiang and Jiuxing Liu and Hyun-Wook Jin and Dhabaleswar K. Panda and Darius Buntinas and Rajeev Thakur and William Gropp. High performance mpi-2 one-sided communication over infiniband. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2004.

[52] Meng-Shiou Wu, R A Kendall, and K Wright. Optimizing collective communications on smp clusters. In *ICPP 2005*, 2005.

[53] W.Yu, S.Sur, D.K.Panda, R.T.Aulwes, and R.L.Graham. High Performance Broadcast Support in LA-MPI over Quadrics. In *Las Alamos Computer Science Institure Symposiun,(LACSI'03)*, Oct 2003.

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

[55] X. Yuan, S. Daniels, A. Faraj, and A. Karwande. Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast. In *The 6th International Conference on Computer Science and Informatics, Durham, NC*, March 8-14 2002.