SCALABLE AND HIGH-PERFORMANCE MPI DESIGN FOR VERY LARGE 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

Sayantan Sur, B. Tech

* * * * *

The Ohio State University

2007

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. S. Parthasarathy

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

In the past decade, rapid advances have taken place in the field of computer and network design enabling us to connect thousands of computers together to form high-performance clusters. These clusters are used to solve computationally challenging scientific problems. The Message Passing Interface (MPI) is a popular model to write applications for these clusters. There are a vast array of scientific applications which use MPI on clusters. As the applications operate on larger and more complex data, the size of the compute clusters is scaling higher and higher. Thus, in order to enable the best performance to these scientific applications, it is very critical for the design of the MPI libraries be extremely scalable and high-performance.

InfiniBand is a cluster interconnect which is based on open-standards and gaining rapid acceptance. This dissertation presents novel designs based on the new features offered by InfiniBand, in order to design scalable and high-performance MPI libraries for large-scale clusters with tens-of-thousands of nodes. Methods developed in this dissertation have been applied towards reduction in overall resource consumption, increased overlap of computation and communication, improved performance of collective operations and finally designing application-level benchmarks to make efficient use of modern networking technology. Software developed as a part of this dissertation is available in MVAPICH, which is a popular open-source implementation of MPI over InfiniBand and is used by several hundred top computing sites all around the world.

Dedicated to all my family and friends

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my PhD study. I'm thankful for all the efforts he took for my dissertation. I would like to thank him for his friendship and counsel during the past years.

I would like to thank my committee members Prof. P. Sadayappan and Dr. S. Parthasarathy for their valuable guidance and suggestions.

I'm grateful for financial support by National Science Foundation (NSF) and Department of Energy (DOE).

I'm thankful to Dr. Bill Gropp, Dr. Rajeev Thakur and Dr. Bill Magro for their support and guidance during my summer internships.

I'm especially thankful to Dr. Hyun-Wook Jin, who was not only a great mentor, but a close friend. I'm grateful to have had Dr. Darius Buntinas as a mentor during my first year of graduate study.

I would like to thank all my senior Nowlab members for their patience and guidance, Dr. Pavan Balaji, Dr. Jiuxing Liu, Dr. Jiesheng Wu, Dr. Weikuan Yu, Sushmita Kini and Bala. I would also like to thank all my colleagues Karthik Vaidyanathan, Abhinav Vishnu, Amith Mamidala, Sundeep Narravula, Gopal Santhanaraman, Savitha Krishnamoorthy, Weihang Jiang, Wei Huang, Qi Gao, Matt Koop, Lei Chai and Ranjit Noronha. I'm especially grateful to Matt and Lei and I'm lucky to have collaborated closely with them.

During all these years, I met many people at Ohio State, some of whom are very close friends, and I'm thankful for all their love and support: Nawab, Bidisha, Borun, Ashwini and Naveen.

Finally, I would like to thank my family members, Swati (my mom), Santanu (my dad) and Sohini (my sister). I would not have had made it this far without their love and support.

VITA

September 29, 1979	Born - Calcutta, India.
August 1997 - July 2001	B.Tech Electrical and Electronics Engineering, Regional Engineering College, Calicut, India.
August 2001 - July 2002	Member of Technical Staff, Sun Microsystems, India.
August 2002 - June 2003	Graduate Teaching Associate, The Ohio State University.
June 2005 - September 2005	Summer Intern, Intel Corp, Urbana-Champaign, IL.
June 2006 - September 2006	Summer Intern, Argonne National Laboratory, Chicago, IL.
June 2003 - August 2007	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

W. Yu, S. Sur, D. K. Panda, R. T. Aulwes and R. L. Graham, "High Performance Broadcast Support in LA-MPI over Quadrics". *International Journal of High Performance Computer Applications*, Winter 2005.

M. Koop, S. Sur and D. K. Panda, "Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram", IEEE International Conference on Cluster Computing (Cluster 2007), Austin TX.

S. Sur, M. Koop, L. Chai and D. K. Panda, "Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms", 15th Symposium on High-Performance Interconnects (HOTI-15), August 2007.

M. Koop, S. Sur, Q. Gao and D. K. Panda, "High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters", 21st Int'l ACM Conference on Supercomputing, June 2007.

S. Sur, M. Koop and D. K. Panda, "High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-Depth Performance Analysis", SuperComputing (SC), November 11-17, 2006, Tampa, Florida, USA.

S. Sur, L. Chai, H.-W. Jin and D. K. Panda, "Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters", International Parallel and Distributed Processing Symposium (IPDPS 2006), April 25-29, 2006, Rhodes Island, Greece.

S. Sur, H.-W. Jin, L. Chai and D. K. Panda, "RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits", Symposium on Principles and Practice of Parallel Programming (PPOPP 2006), March 29-31, 2006, Manhattan, New York City.

S. Sur, U. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda, "High Performance RDMA Based All-to-all Broadcast for InfiniBand Clusters", International Conference on High Performance Computing (HiPC 2005), December 18-21, 2005, Goa, India.

S. Sur, A. Vishnu, H.-W. Jin, W. Huang and D. K. Panda, "Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems?", Hot Interconnects Symposium, August 17-19, 2005, Stanford University, Palo Alto, California.

H.-W. Jin, S. Sur, L. Chai and D. K. Panda, "LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Clusters", International Conference on Parallel Processing (ICPP-05), June 14-17, 2005, Oslo, Norway.

L. Chai, S. Sur, H.-W. Jin and D. K. Panda, "Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand", Workshop on Communication Architecture for Clusters (CAC 2005); In Conjunction with IPDPS, April 4-8, 2005, Denver, Colorado.

S. Sur, H.-W. Jin, and D.K. Panda, "Efficient and Scalable All-to-All Exchange for InfiniBand-based Clusters", International Conference on Parallel Processing (ICPP-04), Aug. 15-18, 2004, Montreal, Quebec, Canada.

W. Yu, S. Sur, D. K. Panda, R. T. Aulwes and R. L. Graham, "High Performance Broadcast Support in LA-MPI over Quadrics", In Los Alamos Computer Science Institute Symposium, (LACSI'03), Santa Fe, New Mexico, October 2003.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Computer Networks	Prof. D. Xuan
Software Systems	Prof. P. Sadayappan

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Modern day existence is enabled by applications such as weather forecasting, Internet search, e-commerce, drug research, space exploration, data-mining, fluid dynamics simulations, etc. It is almost impossible for us to imagine our daily life without these applications. All these applications depend upon High-Performance Computing (HPC) systems. These systems employ up to several thousand computers connected together by modern networking technologies such as Gigabit Ethernet, Myrinet [7], InfiniBand [17] etc., to execute these applications. Modern HPC systems operate at "Terascale" (10^{12} calculations per second). Although computation power available today seems to be plentiful, it is not enough to satiate the need of demanding applications such as weather forecasting. Due to inadequate computation power, weather cannot be accurately predicted beyond a few days, resulting in delayed advisories or bad predictions. One can expect that tomorrow's applications will be even more challenging, requiring HPC systems to reach "PetaScale" (10^{15} calculations per second).

One of the popular types of parallel computers is called a "Cluster". Clusters are built out of commodity components taken off-the-shelf. The high volume of commodity components brings down the cost of clusters significantly and boosts the cost-to-performance ratio. In fact, currently 373 of the top 500 most powerful computers in the world [45] are clusters. The

adoption of clusters for very high-end computing has also been encouraged by availability of high-performance interconnects which provide communication support for these clusters. Recent advances in interconnection technology can boost the size of clusters to tens-of-thousands of nodes. In order to scale up HPC systems to PetaScale, one of the major challenges is to provide scalable software environments under which parallel applications can execute. The **M**essage **P**assing **I**nterface (MPI) [30] is one of the most popular software environment for HPC systems, and is used by almost all HPC applications. Thus, it is crucial that the design and implementation of MPI is scalable, so that the applications utilizing MPI can also scale accordingly.

InfiniBand [17] is a cluster interconnect which is based on open standards and is gaining widespread acceptance. It offers several new features which make it desirable for High Performance Computing. MVAPICH [34] is a popular implementation of MPI over InfiniBand which is used by several hundred of the top computing sites all around the world. The basic design of MVAPICH was proposed by J. Liu et al [24]. Since the initial design, the scale of the InfiniBand clusters in production use has grown from a few hundreds-of-nodes to several thousands of nodes. In fact, InfiniBand clusters with tens-of-thousands of nodes are being built for use by the end of this year. As the order-of-magnitude of the clusters has increased, the design of the MPI layer needs to ever more scalable and high-performance. In this dissertation, we present our studies on how to design a high-performance and scalable MPI communication layer while leveraging novel features offered by InfiniBand.

The rest of this Chapter is organized as follows. First we provide an overview of MPI and relevant scalability issues. We then provide an overview of InfiniBand and its modern networking mechanisms. Following that, we present the problem statement and the research approaches. Finally, we provide an overview of this dissertation.

1.1 Overview of MPI

The message passing model of parallel computing requires explicit communication between processes involved in the computation. This model has been thought to be one of the most effective methods to scale a parallel computer up. Message passing was used in early supercomputers in the 1980s. However, during that period, nearly every supercomputer had a different way of doing message passing, and applications were not portable from one parallel computer to the other. To rectify this situation and to pave the way for developing high-performance scientific applications, efforts were started in the early 1990s to standardize the interface which is used by applications to send and receive messages. The result of the standardization process is the Message Passing Interface (MPI) [30]. An extension to the initial specification is also available as MPI2 [31]. MPI provides an application programming interface (API) for the Fortran, C and C++ languages.

MPI has since established itself as the *de-facto* standard of parallel computing. Nearly all scientific computation applications are written using MPI, and many higher-level communication libraries require MPI. MPI is very portable and has been ported to nearly all parallel computer architectures. There is a wide variety of quality MPI implementations available as open-source: MPICH [15], MPICH2 [27], MVAPICH, MVAPICH2 [34], OpenMPI [13]. MPI provides two major modes of communication, point-to-point and collective. In point-to-point communication, individual pairs of processes are involved in sending and receiving messages. In collective communication operations, groups of processes are involved in the data exchange. In this section, we describe the major communication modes offered by MPI in detail, their semantics and issues in designing scalable and high-performance MPI.

1.1.1 Point-to-Point Communication

In an MPI program, two processes can exchange messages using point-to-point communication primitives. The process wishing to send a message, can send it using a function `MPI_Send`. The receiving process may retrieve this message with a matching `MPI_Recv`. Messages are matched by a three-tuple *source*, *tag* and *context*. The "source" indicates the process where the message originated. The "tag" is a user supplied integer value and can be used to separate different messages. The "context" is the group of the processes the sending process belongs to.

`MPI_Send` and `MPI_Recv` are the most commonly used MPI functions. However, there are variations of these calls. `MPI_Send` and `MPI_Recv` are often called as the "blocking" mode calls, i.e. the sending and receiving processes block on these calls until the corresponding operations complete, or the message buffers can be reclaimed by the application. `MPI_Isend` and `MPI_Irecv` are the *asynchronous* versions of the send and receive calls. Using these, the application can initiate send and receive operations while continuing to perform its own computation. The MPI library will attempt to make progress in the meanwhile and can complete these operations. In order to finish the asynchronous operations, the application then needs to call `MPI_Wait`.

The other modes of point-to-point calls are *synchronous*, *buffered* and *ready*. In the synchronous mode, the application is guaranteed that the network/message transfer operations relating to the send/receive are complete before control returns to the application. This mode is activated using `MPI_Ssend` and is primarily used for debugging MPI applications which erroneously have assumed internal MPI buffering. The buffered mode allows applications to provide explicit memory buffers for communication, so that it doesn't have to worry about where the messages may actually be buffered. This is mainly a convenience

function and is activated using `MPI_Bsend`. Finally, in the ready mode may be used when the sending process is sure that a matching receive has been posted. This is an attempt to optimize network protocols associated with that send/receive operation. However, in most MPI implementation, the ready send is just mapped to `MPI_Send` for the sake of convenience.

1.1.2 Collective Communication

In addition to the point-to-point communication primitives, MPI offers collective communication operations. These functions allow a group of processes to perform communication in a coordinated fashion. Based on the physical network and system topology, these operations can be then highly optimized by the MPI library. The application using these MPI functions then need not be aware of specific platform specific parameters in order to optimize these communication patterns. Examples of collective communication are: `MPI_Alltoall`, `MPI_Allgather`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Barrier` etc. Thus, the collective operations not only provide a simple and intuitive interface to application programmers but also give MPI implementors a greater opportunity to optimize them.

1.1.3 MPI Design Issues

In order to design a high-performance and scalable MPI library, there are many different design issues to consider. In this section, we provide and overview of the general issues that need to be dealt with. In the subsequent sections, we will describe them in depth in context with the InfiniBand Architecture.

Communication Buffer Management

MPI assumes a fully connected model. Under this assumption, applications utilizing MPI can send/receive messages from each other at any given moment during the execution.

In order to transmit messages, the MPI library has to ensure that there is some memory space at the receiver where incoming messages land. On the receiving side, the MPI library can then inspect the incoming messages to perform the correct action. The memory space which must be dedicated to receive messages is often referred to as "communication buffers". As the number of processes in the MPI application increases, the amount of buffer space required by the MPI library should not increase dramatically. On the contrary, some limits must be enforced on how much memory is consumed by communication buffers based on how much memory is available at each sending process. Even for process counts in several tens-of-thousands, memory required must be within reasonable limits.

Flow Control

As mentioned in the previous section, MPI assumes a fully connected model with processes sending and receiving messages at will. The MPI library should ensure that incoming messages do not totally overwhelm the receiving process. In order to achieve this, the MPI library should impose directly or indirectly some flow control which limits the rate at which communication buffers are consumed from within the MPI library. As the system size scales, overhead imposed by flow control should be as low as possible. In addition, good flow control mechanisms should allow for as much communication to take place as possible before imposing strict limitations on the rate of incoming messages.

Communication Protocol Design and Progress

Applications using MPI may send or receive arbitrary sized messages. Internally, MPI uses two major types of protocols. They are called *Eager* and *Rendezvous* protocols. In the eager mode, usually used for small messages, the sending process simply sends the message over to the remote side, where it is temporarily buffered in the communication buffers. The

Rendezvous mode, usually used for larger messages involves a handshake operation before the actual message is sent. This is done in order to guarantee availability of memory at the receiver for the entire message. The internal design of these protocols is key to achieving high-performance and overlap of computation and communication. The overlap indicates that the MPI application is able to continue computing while the MPI library along with the network-interface take on the responsibility of transferring the message, i.e. continue to make progress.

Collective Communication

Most collective operations are blocking operations, i.e. requiring reception of messages from remote processes. This makes collective operations especially latency sensitive. Accordingly, applications expect that each collective operation be carefully tuned according to the specific platform. While designing collective operations for large system sizes, scalable algorithms and techniques should be employed. There should be as less memory dedicated as possible and it should be reused as much as possible. Finally, for achieving lowest latency, the collective operations can be based on direct network primitives and not on MPI point-to-point operations.

1.2 Overview of InfiniBand

The InfiniBand Architecture [17] (IBA) defines a switched network fabric for interconnecting compute and I/O nodes. In an InfiniBand network, compute and I/O nodes are connected to the fabric using Channel Adapters (CAs). There are two types of CAs: Host Channel Adapters (HCAs) which connect to the compute nodes and Target Channel Adapters (TCAs) which connect to the I/O nodes. IBA describes the service interface between a host channel

adapter and the operating system by a set of semantics called *Verbs*. Verbs describe operations that take place between a CA and its operating system for submitting work requests to the channel adapter and returning completion status. Figure 1.1 depicts the architecture of an InfiniBand network.

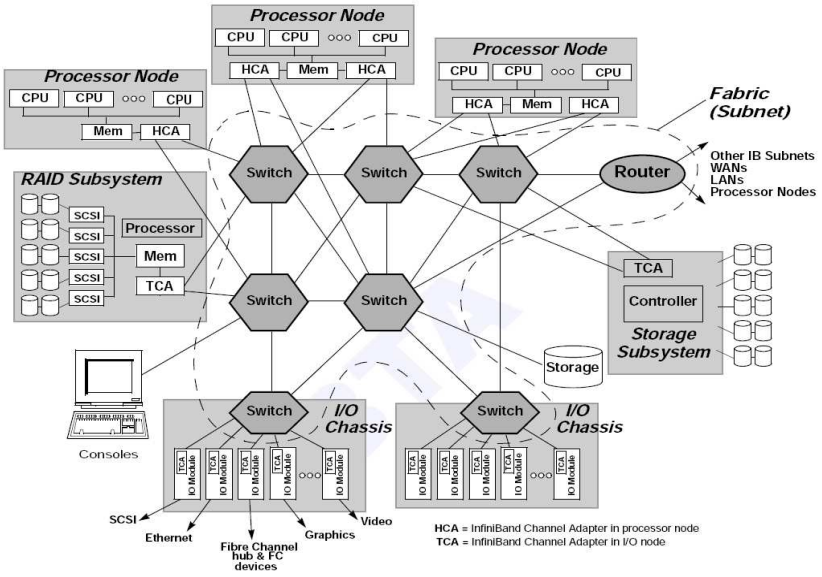


Figure 1.1: InfiniBand Architecture (Courtesy IBTA)

InfiniBand uses a queue based model. A consumer can queue up a set of instructions that the hardware executes. This facility is referred to as a *Work Queue* (WQ). Work queues are always created in pairs, called a *Queue Pair* (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred between the consumer's memory and another consumer's memory, and the receive work queue holds instructions about where to place data that is received from another consumer. The completion of WQRs is reported through Completion Queues (CQ).

Figure 1.2 shows a Queue Pair connecting two consumers and communication through the send and the receive queues.

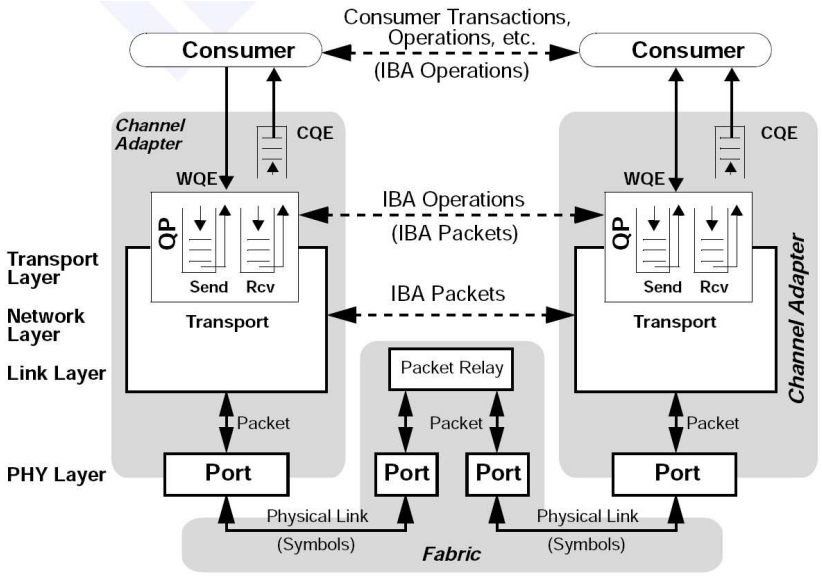


Figure 1.2: IBA Communication Stack (Courtesy IBTA)

1.2.1 Communication Semantics

InfiniBand supports two types of communication semantics. They are called *Channel* and *Memory* semantics. In channel semantics, the sender and the receiver both explicitly place work requests to their QP. After the sender places the send work request, the hardware transfers the data in the corresponding memory area to the receiver end. It is to be noted that the receive work request needs to be present before the sender initiates the data transfer. This restriction is prevalent in most high-performance networks like Myrinet [32], Quadrics [36] etc.

In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send/receive operations. These RDMA operations are one-sided and do not require

any software involvement at the other side. ie. the other side CPU does not have to issue any work request for the data transfer. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand.

1.2.2 Transport Services

The InfiniBand architecture supports multiple classes of transport services. A queue pair can be configured with either of these types of transports:

1. Reliable Connection (RC)
2. Reliable Datagram (RD)
3. Unreliable Connection (UC)
4. Unreliable Datagram (UD)
5. Raw Datagram

Transport services RC and UC are connection oriented. They require a QP to be exclusive to a pair of processes. On the other hand RD and UD are connection-less, i.e. a QP may be used to communicate with as many pairs of processes as possible. Each QP requires a particular set of resources. These resources are mainly to store context information and in case of reliable transports to guarantee reliable, in-order delivery. In general, connection-less transports require lesser resources than connection-oriented transports. The Raw Datagram is used to provide compatibility with other types of networks. For example, IPv6 packets may be tunneled over InfiniBand using this type of transport. The Raw Datagram is out of scope for this proposal.

Table 1.1 compares the various transports provided by IBA. We note that for m processes on n nodes and all processes connected, the RC and UC transports require the most number

Attribute	RC	RD	UC	UD	Raw Datagram
Scalability (Number of QPs)	m^2n	m	m^2n	m	1
Corrupt Data Detected	Yes	Yes	Yes	Yes	Yes
Delivery Guarantee	Yes	Yes	No	No	No
Data Loss Detection	Yes	Yes	No	Yes	No
Error Recovery	Reliable	Reliable	Unreliable	Unreliable	Unreliable

Table 1.1: Comparison of IBA Transport Types

of QPs. On the other hand RD and UD require much lesser QPs. However, to the best of our knowledge, no InfiniBand hardware currently implements RD. The RC transport provides reliable in order delivery and detection and is suitable for programming models such as MPI which require all processes to be logically connected and provide reliable data transfer.

1.2.3 Shared Receive Queue

InfiniBand provides channel communication semantics (as described in Section 1.2.1) in the form of send and receive operations on a QP. In order to use these operations, it is necessary that the receive work requests are placed before the send operations are issued. This ensures that the CA has enough resources to place the data which it receives. However, this presents a scalability issue when there are multiple QPs communicating. Resources made available to one QP are wasted if there is no more communication with that remote process. To handle such a situation, the IBTA specification version 1.2 introduced a new software service called the *Shared Receive Queue*. This allows the CA to share receive requests for several QPs into one FIFO queue. In addition to providing a scalable mechanism to share receive requests for multiple connections, the SRQ also provides a "Low-watermark" asynchronous service. If the shared receive requests drop below a preset threshold, then the

application (in our case MPI library) may be notified of this event. This event may allow applications to perform flow-control or other operations based on their requirement.

1.2.4 Memory Registration

InfiniBand requires that all memory that is used for communication be "registered" before any data is sent or received into it. Registration is a two phase operation in which the pages are marked unswappable (ie. these will no longer be paged out to disk) and the virtual addresses of the pages in concern will be sent to the CA. The reason for this requirement is that when the CA actually performs the communication operation, the data should be present in the RAM and the CA should know its address.

Registration is usually a high-latency blocking operation. In addition, since the memory pages registered cannot be swapped out, the application (running on top of MPI) has lesser physical memory available.

1.2.5 Completion and Event Handling Mechanisms

In InfiniBand, the Completion Queue (CQ) provides an efficient and scalable mechanism to report completion events to the application. The CQ can provide completion notifications for both send and receive events as well as many asynchronous events. It supports two modes of usage: i) Polling ii) Asynchronous. In the Polling mode, the application uses an InfiniBand verb to poll the memory locations associated with the completion queue. One or many completion entries may be returned at one go. In the Asynchronous mode, the application need to continuously poll the CQ to look for completions. The CQ will generate an interrupt when a completion event is generated. Further, IBA provides a mechanism by which only "solicited events" may cause interrupts. In this mode, the application can poll the CQ, however on selected types of completions, an interrupt is generated. This mechanism allows

interrupt suppression and thus avoid unnecessary costs (like context-switch) associated with interrupts.

1.3 Problem Statement

Cluster computing has become mainstream HPC (High-Performance Computing) now, with 75% of the most powerful parallel computers being clusters. These modern clusters are equipped with powerful interconnects, such as InfiniBand [17]. Scientific applications executing on these clusters use primarily MPI as their programming model. As the size of these clusters continues to increase, it is crucial to design MPI libraries in the most efficient and scalable manner.

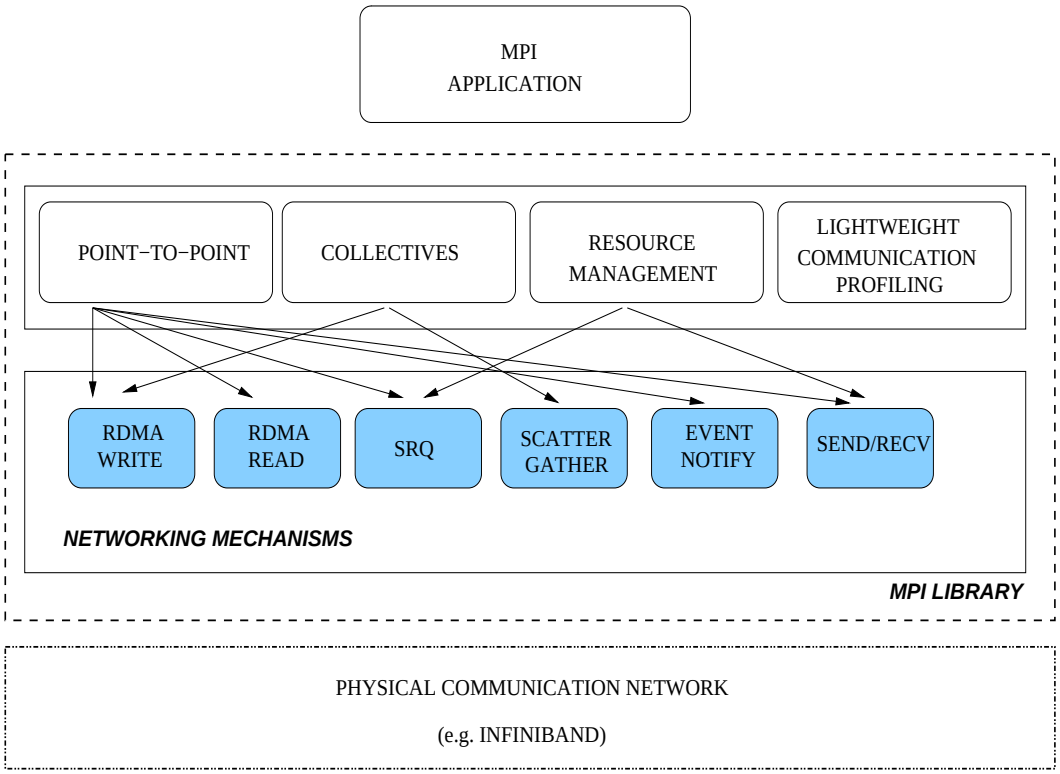


Figure 1.3: Problem Space for this Dissertation

Figure 1.3 shows the scope of this dissertation. In short, we aim to design the point-to-point and collective components of the MPI libraries to make use of novel network primitives in order to make better utilization of resources and be most efficient. We intend to understand the resource usage characteristics and communication patterns of MPI applications not only for optimizing MPI libraries, but also to optimize the end applications/benchmarks themselves. We present the problem statement in details as follows:

- Can we design point-to-point operations in a manner such as to improve computation and communication overlap ratio? – The computation and communication overlap ratio is an indication of the amount of computation an MPI application can perform while communication operations are pending. Current generation MPI libraries depend upon the MPI application to invoke communication progress by using MPI routines. This means that computation and communication cannot proceed in parallel. However, independent progress of communication operations is critical to achieving good overall application performance. As far as possible, the performance of the MPI library should not depend on the application calling MPI routines to make progress. With system sizes scaling to the tens-of-thousands of processors, application developers are forced to tackle many complex optimizations. The MPI library should take charge of making sure communication operations can proceed without direct application involvement. Enabling independent communication is a tough challenge, as it involves asynchronous actions to take place while the CPU is busy in computation routines.

- Can the collective operations be designed to leverage modern networking mechanisms and achieve improved latency and scalability? – MPI applications often utilize collective communication routines in order to enable the MPI library to optimize performance based on the system architecture/topology. However, MPI libraries often implement

collective operations on top of the point-to-point operations (e.g. `MPI_Send`, `MPI_Recv`, etc.), adding software overhead and losing the collective operation semantics which leads to lack of optimization opportunities. Designing collective operations directly over raw network primitives is a promising approach to removing software overheads. However, it is a challenge to identify and design the right algorithms and design them in a scalable and high-performance manner.

- Can we leverage newer features of InfiniBand and design newer communication buffer organization techniques which have scalable resource usage? – MPI specifies a fully connected model. Under such a model any process can send or receive from any other process. The MPI library needs to reserve some memory to handle the exchange of messages. The amount of communication buffers the MPI library needs should not rapidly increase with the number of processes in the application. Current generation MPI implementations often allocate communication buffer resources on a *per remote process* basis in order to optimize point-to-point communication performance. Although these approaches provide good performance for small scale clusters, they are simply not viable for larger scale clusters as they require several GigaBytes of memory per process. The challenge is to devise a mechanism which not only allows high-performance for small scale clusters, but scales well for very large clusters with tens-of-thousands of nodes.

- Can the new scalable communication buffer organization achieve similar or better performance than previous designs for a wide variety of MPI applications? – MPI applications exhibit a wide variety of communication patterns. For every different communication, buffers may be consumed in different ways. The communication buffer

mechanism must be able to achieve high-performance under most of the highly likely communication patterns, while providing scalable buffer usage. A very detailed study of a variety of MPI applications is required in order to understand the performance characteristics of the buffer management mechanisms. There are no existing profiling tools that offer such detailed information, and in order to study these parameters, a light-weight profiling layer is required inside the MPI library.

- Can we achieve a significantly better understanding of application/benchmark characteristics and requirements, and redesign them according to the strengths of modern interconnects? – As the system sizes increase to tens-of-thousands of nodes, the scope to perform optimization is greatest in the MPI application. Many of the existing MPI codes have been written several years back when the networks didn't offer novel features. Thus, the MPI applications/benchmarks may not be best suited for the modern generation of networks. It is therefore crucial to understand the behavior of these applications and benchmarks and provide insights to real application developers about scalable techniques to employ while writing PetaScale applications.

1.4 Research Approaches

In this section we present our general approaches to the above mentioned issues.

1. **Designing rendezvous protocol leveraging RDMA Read to enhance computation and communication overlap** – We have designed a novel rendezvous protocol which achieves nearly complete computation and communication overlap. We make use the RDMA Read feature offered by InfiniBand and couple it with selective interrupts for enabling the MPI library to make progress even when the application cannot call the MPI library.

2. **Designing scalable collective operations leveraging lower level RDMA and non-contiguous operations** – We have designed RDMA based collective operations for two of the commonly used MPI collectives: `MPI_Alltoall` and `MPI_Allgather`. Our approach cuts down on software overheads bypassing several layers directly to the InfiniBand communication layer and leveraging RDMA and native support for non-contiguous communication.

3. **Designing scalable communication buffer management techniques** – We have designed a scalable technique to manage communication buffers utilizing Shared Receive Queues. Using our method, communication buffers need not be dedicated per process, rather, they are used in a FIFO order from a shared pool. We have also designed an associated flow control method with this management scheme.

4. **In-Depth performance analysis of MPI library with reduced memory usage utilizing internal profiling layers** – We have analyzed in detail the performance characteristics of our overall designs for a wide variety of end MPI applications. We developed an internal profiling layer to collect information on lower layer events, memory usage to validate design decisions.

5. **Redesigning application level benchmarks to leverage modern networking capabilities** – We have redesigned two widely used HPCC [18] benchmarks, High-Performance Linpack and RandomAccess, to better utilize modern network and benefit from advances in MPI design as per the above mentioned work.

1.5 Dissertation Overview

We have presented our research over the next several chapters.

In Chapter 2 we take on the challenge of redesigning the Rendezvous Protocol in order to improve the computation and communication overlap. We leverage the RDMA Read semantics to reduce the number of intermediate messages required to transmit an MPI level message. In addition, we utilize the *selective interrupt* mechanism to insert progress calls inside the MPI library even though the MPI application may be busy in computation. Using our design, the MPI library is able to offer almost complete computation and communication overlap.

In Chapters 3 and 4, we present new designs to take advantage of the advanced features offered by InfiniBand in order to achieve scalable and efficient implementation of the `MPI_Alltoall` and `MPI_Allgather` collectives. We proposed that the implementation of collectives be done directly on the InfiniBand Verbs Interface rather than using MPI level point-to-point functions. We evaluate our proposed designs in detail Our experimental results and analytical models enable us to conclude that our new designs can be more scalable and efficient than current approaches.

In Chapter 5, we propose a novel Shared Receive Queue based Scalable MPI design. Our design uses low-watermark interrupts to achieve efficient flow control and utilizes the memory available to the fullest extent, thus dramatically improving the system scalability. In addition, we also proposed an analytical model to predict the memory requirement by the MPI library on very large clusters (to the tune of tens-of-thousands of nodes).

As InfiniBand gains popularity and is included in increasingly larger clusters, having a scalable MPI library is imperative. Through our evaluation of the NAS Parallel Benchmarks, SuperLU, NAMD, and HPL in Chapter 6, we explore the impact of reduction of communication memory on the performance. Our evaluation shows that the latest SRQ design

of MVAPICH is able to use a constant amount of internal memory per process with optimal performance, regardless of the number of processes, an order of magnitude lesser than other Eager protocol designs of MVAPICH. In our experiments, only 5-10MB of communication memory was required by the SRQ design to attain the best recorded performance level achievable with MVAPICH.

In Chapter 7, we demonstrate that by revisiting the design of end MPI applications, we can gain significant performance improvement. The communication patterns of these applications and benchmarks need to be studied and modified to take the most advantage out of modern networks and their capabilities. The MPI design parameters can have a significant impact on the performance characteristics of end applications. With the coupling of the application modifications with optimized MPI library design, we can improve overall performance significantly.

CHAPTER 2

IMPROVING COMPUTATION AND COMMUNICATION OVERLAP

MPI provides both blocking and non-blocking semantics of point-to-point communication. Of these, it is widely accepted that non-blocking semantics offer better performance to end-applications by allowing overlap of computation and communication. Applications can use `MPI_Isend`, `MPI_Irecv` to initiate the communication operations and return to computing. When the application needs the messages, they can call `MPI_Wait`. Most high-performance MPI implementations are based on polling progress engines, i.e. the sender and receiver processes must periodically call MPI functions to ensure communication progress. However, due to certain MPI internal protocols (such as Rendezvous protocol), overlap of computation and communication may be hampered. If progress calls are not triggered for a long time, messages may be severely delayed.

In this Chapter, we take on the challenge of redesigning the Rendezvous Protocol in order to improve the computation and communication overlap. We leverage the RDMA Read semantics to reduce the number of intermediate messages required to transmit an MPI level message. In addition, we utilize the *selective interrupt* mechanism to insert progress calls inside the MPI library even though the MPI application may be busy in computation. Using

our design, the MPI library is able to offer almost complete computation and communication overlap. Application wait times can be reduced by 30%.

The rest of the chapter is organized as follows. In Section 2.1 we provide necessary background information for this work. In Section 2.2 we describe current approaches and their limitations. In Section 2.3, we discuss the design alternatives and the final design approach. In Section 2.4, we evaluate our design and provide experimental results. Finally, in Section 2.5 we summarize the results and impact of this work.

2.1 Background

In this section, we provide the necessary background details for this work. First, we describe the Rendezvous Protocol, then we describe the RDMA Write and RDMA Read mechanisms of InfiniBand.

2.1.1 Overview of Rendezvous Protocol

The Rendezvous Protocol negotiates the buffer availability at the receiver side before the message is actually transferred. This protocol is used for transferring large messages when the sender is not sure whether the receiver actually has the buffer space to hold the entire message. MVAPICH [34], along with MPICH-GM [33], MPICH-Quadrics [38], utilize RDMA Write to totally eliminate intermediate message copies and efficiently transfer large messages.

In this protocol, the sending process sends a `START` message to the receiver along with the message envelope (usually, MPI message matching tags). Upon receipt of this message, the the envelope is buffered. When the matching receive is posted by the receiving MPI application, the receiver side sends a `REPLY` message with the location of the receiver user buffer. Upon discovery of the `REPLY` message, the sender sends the actual data `DATA` message

to the receiver. This is followed by a `FIN` message from the sender indicating end of all data. Thus, the message can be transferred to the receiver. As the data transfer doesn't start until matching receives are posted, only buffering of message envelopes are required.

2.1.2 Overview of InfiniBand RDMA-Write and RDMA-Read

InfiniBand offers two types of memory access semantics, RDMA-Write and RDMA-Read. RDMA stands for *Remote Direct Memory Access*. Using RDMA, a network-interface can access the memory of a remote node transparent to the CPU at the remote node. The CPU is involved only in granting access privileges to the network-interface. Once the access control is set up, the DMA engines at the network-interface directly access memory without any further intervention. This enables the CPU to perform more useful computation work, while leaving networking responsibilities to the NIC. This also aids in reducing the amount of cache pollution. Since memory need no longer be touched by the CPU, it need not be brought back into the various levels of caching hierarchy.

In the RDMA-Write mechanism, the sending process is aware of the remote memory location and has a message to send which fits in the amount of memory available in the remote memory window. The network-interface takes the message contents from memory and places them directly in the memory of the remote process. In RDMA-Read, the originating process can read the remote memory contents with aid from the network-interface at the remote side.

2.2 Current Approaches and their Limitations

The RDMA Write based protocol is illustrated in Figure 2.1(a). The sending process first sends a control message to the receiver (`RNDZ_START`). The receiver replies to the sender

using another control message (RNDZ_REPLY). This reply message contains the receiving application's buffer information along with the remote key to access that memory region. The sending process then sends the large message directly to the receiver's application buffer by using RDMA Write (DATA). Finally, the sending process issues another control message (FIN) which indicates to the receiver that the message has been placed in the application buffer.

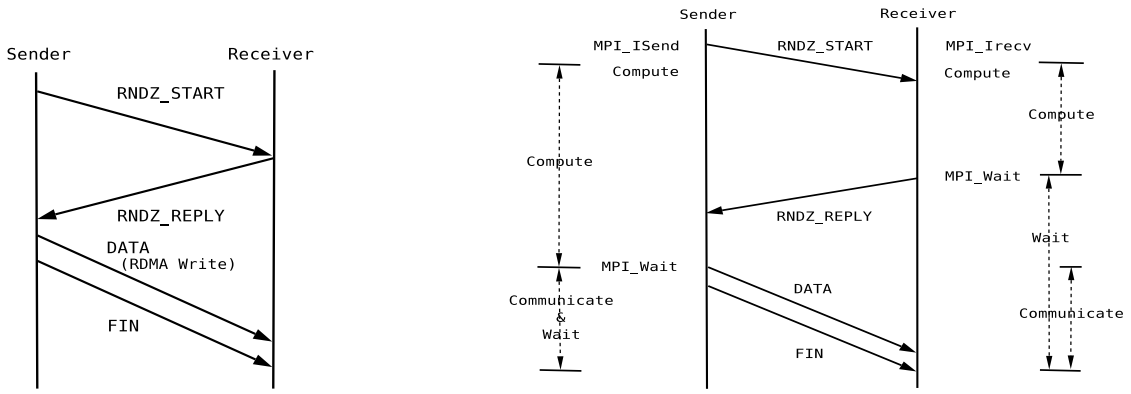
MVAPICH uses a progress engine to discover incoming messages and to make progress on outstanding sends. To achieve low latency, the progress engine senses incoming messages by polling various memory locations. As can be seen in Figure 2.1(a), the RDMA Write based Rendezvous Protocol generates multiple control messages which have to be discovered by the progress engine. Since the progress engine is polling based, it requires the application to call into the MVAPICH library.

However, the MPI applications might be busy doing some computational work or I/O. In this case the applications cannot make any call into the MPI library. As a result, the message transfer has to simply wait until the control messages are discovered. This scenario is illustrated in Figure 2.1(b). The delayed discovery of important control messages leads to serialization of the computation and communication operations. As a result, the overlap potential of computation and communication is severely hampered as shown.

2.3 Design Alternatives and Challenges

In this section, we compare RDMA Read and Write as design alternatives and pick the best one of them. We will compare the two based on parameters like: communication progress, computation/communication overlap, number of I/O bus transactions, etc.

Typically, small messages are sent over Eager Protocol (which is copy-based) and larger messages are set over Rendezvous Protocol. According to the MPI specification, only the



(a) Rendezvous Protocol

(b) Rendezvous Protocol Limitations

Figure 2.1: MVAPICH Rendezvous Protocol and its Limitations

sender can choose the actual protocol efficiently. Particularly, the MPI Specification [30] states that: *"The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer. If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified."* According to the requirements imposed by MPI semantics, the receiver may post a much larger buffer than what the sender chooses to send. Since, the choice of size of the message actually sent (not posted size), lies with the sender, the sender can efficiently make a choice of which protocol to use (Eager or Rendezvous).

Now, we consider the case in which the sender decides to use the Rendezvous Protocol for the message transfer. The operation of a RDMA Write based protocol is shown in Figure 2.1(a) and that based on a RDMA Read protocol is shown in Figure 2.2(a). Based on program execution and timing, there can be three cases.

- **Sender arrives first:** If the sender arrives first at the send call, it can send the `RNDZ_START` message immediately. Inside the `RNDZ_START` message, it can also embed the virtual address and memory handle information about the buffer to be sent. It is to be noted that upon the receipt of this `RNDZ_START` message, all the information about the application buffer is available to the receiving process. Clearly, the receiving process does not need to send a `RNDZ_REPLY` message any more. It can simply perform a RDMA Read from the application buffer location of the sending process.

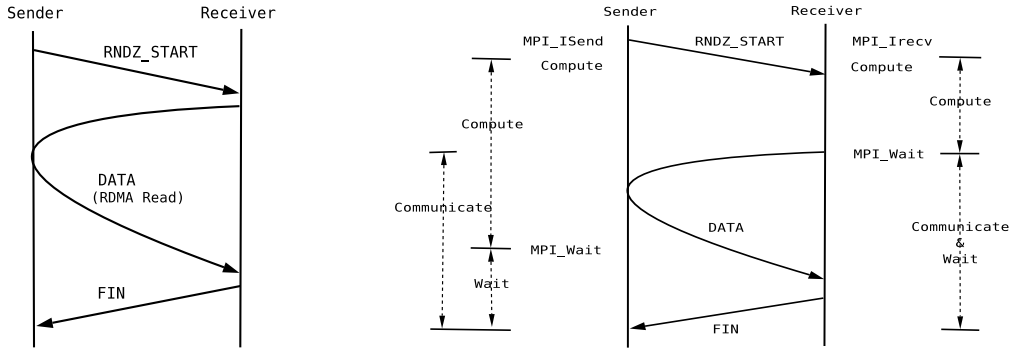
- **Receiver arrives first:** Even if the receiver arrives first at the receive call, it cannot choose which protocol the message will be actually sent over. So, it must wait for the sender's choice of protocol. The receiver waits for the `RNDZ_START` message from the sender. However, once the receiver gets the `RNDZ_START` message, it can perform the RDMA Read directly from the sender buffer, without sending any more `RNDZ_REPLY` message.

- **Sender and receiver arrive at the same time:** In this case, the sender and the receiver arrive concurrently. However, neither the sender or the receiver knows whether the other process has arrived. Hence, in this case, the receiver must wait for the protocol choice from sender (as stated before), and the sender must assume that it has arrived first. Hence, again in this case, the optimal choice would be to have the sender send a `RNDZ_START` message to the receiver. As stated above, the receiving process can simply perform a RDMA Read from the sender buffer directly.

As per the above three cases, RDMA Read is chosen to reduce the number of control messages. Since the number of control messages is reduced, the total number of I/O bus transactions are reduced too. In addition, since the receiver can progress independently of the

sender (once the RNDZ_START message is sent), we can enhance the communication progress. Further, even if the sender does not call any MPI progress, the data transfer can proceed over RDMA Read. This leads to much better overlap of computation with communication, if RDMA Read is used.

Thus, we conclude from the above that: the optimal choice of data transfer semantics is RDMA Read in all possible combinations of sender or receiver arriving at the communication point.



(a) RDMA Read Protocol Operation

(b) RDMA Read Computation Overlap

Figure 2.2: RDMA Read Based Rendezvous Protocol

2.3.1 RDMA Read with Interrupt Based Rendezvous Protocol

In this section we describe the design of Rendezvous Protocol using RDMA Read with interrupt. As we described earlier in this section, RDMA Read is the best data transfer mechanism when the sender arrives first. However, if the receiver arrives first, it still needs to wait for the RNDZ_START message from the sender. In the meantime, the receiver might be busy computing. The discovery of this RNDZ_START message is critical to achieving good

overlap between computation and communication. Since this control message is critical, we can generate an interrupt on its arrival. This message should be handled by an *asynchronous completion handler*. The basic protocol is illustrated in Figure 2.3(a).

Selective Interrupt: Interrupts are usually associated with various overheads. Causing too many interrupts can harm the overall application performance. We devise a method by which we can cause a selective interrupt only on the arrival of `RNDZ_START` message and completion of RDMA Read `DATA` message. In order to have selective interrupts, two things must be done. First, the sender has to set a solicit bit in the descriptor (`solicit_event`) of the message which is intended to cause the interrupt. Secondly, the receiver must request for interrupts from the completion queue by setting `VAPI_SOLIC_COMP` prior to the arrival of the message.

Interrupt Suppression: Even though we have a selective interrupt scheme, back-to-back `RNDZ_START` messages should not generate multiple interrupts. This will harm the overall application performance. For designing this scheme, we disable any interrupts on the completion queue automatically after the asynchronous event handler is invoked. The event handler then keeps on polling the completion queue until there are no more completion descriptors. Thus, in this design even though back-to-back `RNDZ_START` messages might arrive, only one interrupt is generated. Finally, when there are no more completion descriptors left, the asynchronous event handler resets the request for interrupts before exiting.

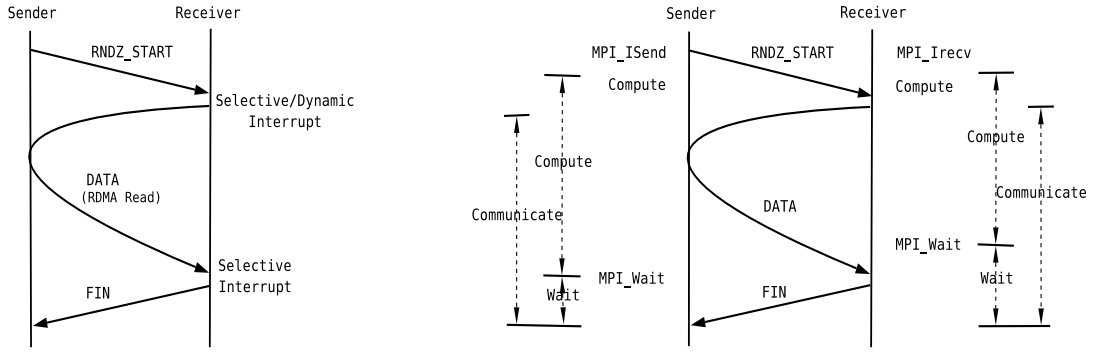
Dynamic Interrupt Requests: The approximate cost of an interrupt is 18 μs on our experimental platform. However, the cost of the receiver requesting an interrupt and clearing it is only 7 μs. Our design of RDMA Read with Interrupt, has such a *dynamic* scheme, in which the receiving process requests for interrupts only when pending receives are posted. If no receives are pending, then the request for interrupts is turned off, and the MPI goes

into polling based progress. Whenever the interrupt is set, an internal flag indicates this status. On posting of subsequent receives, this interrupt does not need to be re-requested. Similarly, when the interrupt is cleared, an internal flag indicates that status too. This dynamic scheme can reduce the number of interrupts in the case where the sender arrives first, but the receive application hasn't posted the receive as yet.

Hybrid Communication Progress: In this new design, our asynchronous event handler is invoked by an interrupt. It executes as a separate thread to the MPI program. Many MPI implementations are based on a polling progress engine, including MVAPICH. This means that whenever a MPI call is issued by the application, the MPI implementation checks all communication channels for incoming messages and makes progress on pending sends. Hence, we can potentially have two threads of the progress engine (one polling and the other handling the event) active at the same time. Thus, we need to provide a thread safe mechanism to implement this hybrid progress engine. At the same time as providing thread safety, it should also provide high performance. If there are no interrupts caused, the overhead imposed by this thread safety mechanism should be minimal. Figure 2.3(b) shows the computation/communication at both the sender and receiver side. In this figure, the `RNDZ_START` message causes an interrupt at the receiver. The RDMA Read `DATA` message is issued immediately. Hence, the computation and communication can be overlapped at both sender and receiver.

2.4 Performance Evaluation

In this section we will present the results we obtained with our proposed RDMA Read based Rendezvous protocol. We compare three schemes, the first one being the RDMA Write (MVAPICH version 0.9.5) [34], the second one being the RDMA Read and the third one



(a) RDMA Read with Interrupt Protocol Operation

(b) RDMA Read with Interrupt Computation Overlap

Figure 2.3: RDMA Read with Interrupt based Rendezvous Protocol

being RDMA Read with Interrupt based Rendezvous Protocol. Our evaluation platforms used were of two types:

- **Cluster A**: 8 SuperMicro SUPER X5DL8-GG nodes with dual Intel Xeon 3.0 GHz processors. Each node has 512KB L2 cache and 2GB of main memory. The nodes are connected to the InfiniBand fabric with 64-bit, 133 MHz PCI-X interface.

- **Cluster B**: 32 nodes, dual Intel Xeon 2.66 GHz processors. Each node has 512KB L2 cache and 2GB of main memory. The nodes are connected to InfiniBand fabric with 64-bit, 133 MHz PCI-X interface.

2.4.1 Computation and Communication Overlap Performance

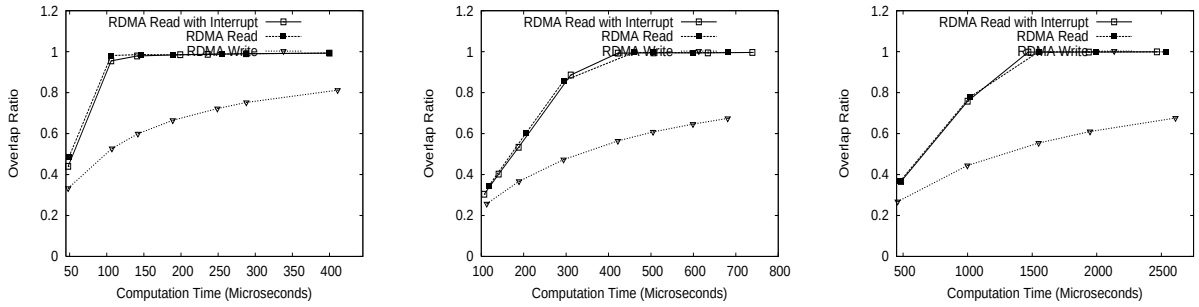
In this section we evaluate the ability of our designed schemes to effectively overlap computation and communication. We designed two micro-benchmarks and carried out the evaluation on Cluster A.

Sender Overlap: In this experiment, we evaluate how well the sending process is able to overlap computation with communication. The sender initiates communication using `MPI_Isend`, then computes for W μs. At the same time, the receiver is just blocking on a `MPI_Recv`. After the sender has finished computing, it checks for completion of the pending sends. The entire operation is timed at the sender. If the entire operation lasted for T μs, then the computation to communication overlap ratio is W/T.

Figure 2.4 shows this ratio versus the computation time. We can see that for the RDMA Write scheme, the overlap ratio is quite low. This is because the sender process is unable to receive the `RNDZ_REPLY` message due to the computation. On the other hand, the RDMA Read and RDMA Read with Interrupt schemes show nearly complete overlap. It is to be noted that for low values of computation time (W), the value of the ratio is low, since in this case, the time for communication is dominant.

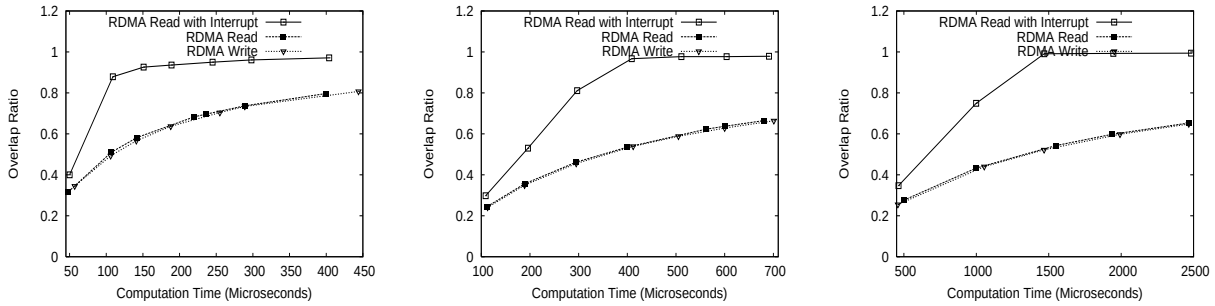
Receiver Overlap: In this experiment, we evaluate how well the receiving process is able to overlap computation with communication. This experiment is similar in nature with the sender overlap experiment. In this experiment, the receiver posts a receive using `MPI_Irecv` and computes for W μs, while the sender blocks on a `MPI_Send`. After the computation, the receiver waits for the communication to complete. The entire time is marked as T. The computation to communication ratio is W/T.

Figure 2.5 shows this ratio versus the computation time. We can see that for the RDMA Write and the RDMA Read schemes, the overlap ratio is quite poor. This is because the receiving process is unable to issue the `RNDZ_REPLY` or `DATA` message due to the computation. On the other hand, the RDMA Read with Interrupt scheme show nearly complete overlap, since the arrival of the `RNDZ_START` message generates an interrupt and the receiving process



(a) Sender Overlap Performance (64KB) (b) Sender Overlap Performance (256KB) (c) Sender Overlap Performance (1MB)

Figure 2.4: Sender Communication and Computation Overlap Performance



(a) Receiver Overlap Performance (64KB) (b) Receiver Overlap Performance (256KB) (c) Receiver Overlap Performance (1MB)

Figure 2.5: Receiver Communication and Computation Overlap Performance

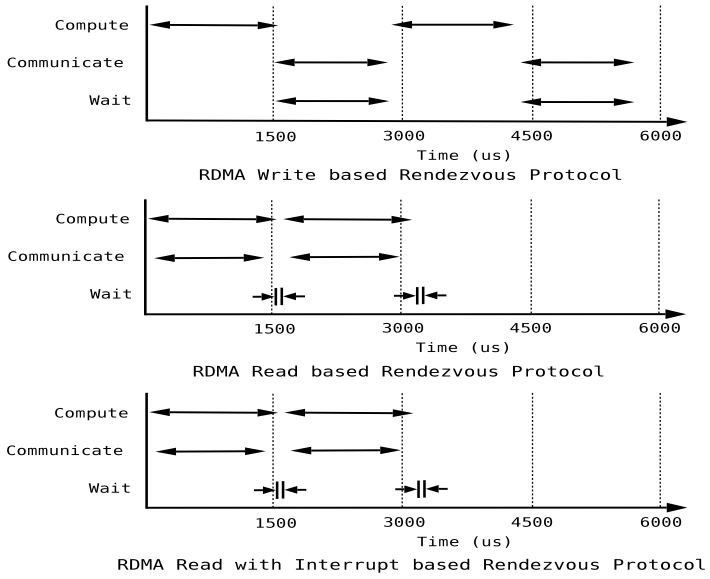


Figure 2.6: Computation and Communication Overlap (Sender) with Time Stamps

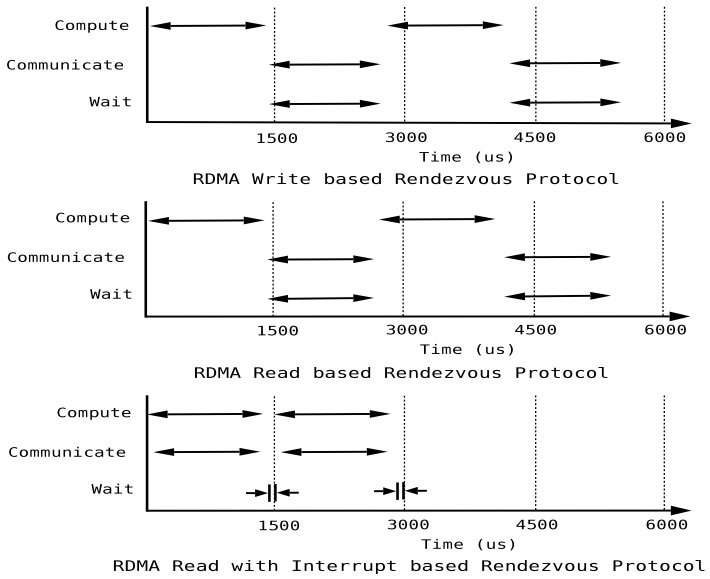


Figure 2.7: Computation and Communication Overlap (Receiver) with Time Stamps

immediately issues the `DATA` message. As noted before, for low values of computation time (W), the communication time is dominant, resulting in a low overlap ratio.

The experimental platform is dual SMPs. In the case of RDMA Read with Interrupt scheme, it may happen that the interrupt handler thread is scheduled on the "idle" processor, thus inflating the benefits of RDMA Read with Interrupt. In order to eliminate such an effect, we perform this experiment on a uni-processor kernel on the same machines. Our experiments reveal that with RDMA Read with Interrupt, we get 99.5% overlap, whereas with RDMA Read and RDMA Write we observe only 62.2% and 59% overlap, respectively, for a 1MB message size with 1800 μs computation time. These results are almost identical with the dual SMP results. This is because the interrupt handler thread consumes very little CPU time and is very short lived. It needs to be "awake" only for a few micro seconds to perform tag matching and post necessary network transactions only if it is required.

Communication Progress: In this execution, we take consecutive time stamps from the micro-benchmark execution. These time stamps are recorded just before the application enters the computation phase, in the `MPI_Wait` and from inside the MPI library when the actual communication takes place.

Figure 2.6 shows the progress snapshot during the sender overlap test. We observe from this figure, that in the RDMA Write based Rendezvous Protocol, the computation and communication are completely serialized. It offers no overlap at all. Whereas, in the RDMA Read based schemes, the communication happens during the application is computing. The RDMA Read based schemes can progress 50% faster when transferring messages of 1MB and computing for 1500 μs.

Similarly, Figure 2.7 shows the progress during the receiver overlap test. We observe from this figure, that in the RDMA Write and the RDMA Read based protocol, the computation

and communication are completely serialized. They hardly offer any overlap. Whereas, in the RDMA Read with Interrupt scheme, the communication happens during the application is computing. The RDMA Read with Interrupt schemes can progress around 50% faster when transferring messages of 1MB and computing for 1500 μs.

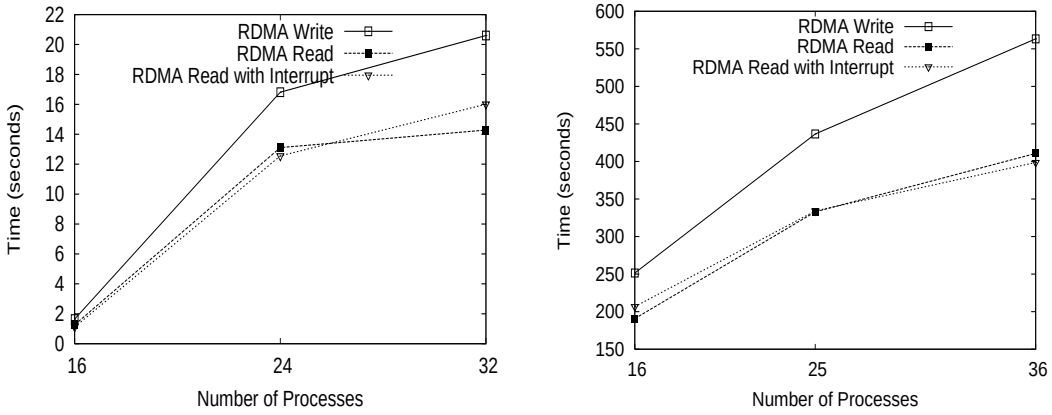
2.4.2 Application level Evaluation

In this section, we evaluate the impact of our RDMA Read and RDMA Read with Interrupt schemes on application wait times. For our evaluation, we choose two well known applications - HPL and NAS-SP (Scalar Pentadiagonal Benchmark). High Performance Linpack (HPL) is a well known benchmark for distributed memory computers [2]. It is used to rank the top 500 computers [45] twice every year. NAS-SP [6] is a CFD simulation which solves linear equations for the Navier-Stokes equation. We used the Class C benchmark for our evaluation.

To find out the communication time for these applications, we use a light-weight MPI profiling library [20], `mpiP`. This profiling tool reports the top aggregate MPI calls and the time spent in each one of them. We collect the aggregate time spent in the `MPI_Wait()` function call. This time is spent by the application just busy waiting for the pending sends and receives to be completed. Since this time is just wasted by the application waiting for the network to complete the operations, this represents time which can possibly be overlapped with computation. Figure 2.8(a) and 2.8(b) show the `MPI_Wait` times for HPL and NAS-SP (Class C) with increasing number of processes, respectively.

We observe that the wait time of HPL is reduced by around 30% for 32 processes by the RDMA Read and RDMA Read with Interrupt designs. Similarly, for the NAS-SP, we can see around 28% improvement for 36 processes. This is mainly because the RDMA Write

based Rendezvous implementation waits till the `MPI_Wait()` to issue the `DATA` message, and hence cannot achieve good overlap. In addition, we observe from the figure that the benefits provided by the new design are scaling with the number of processes. Hence, our new design is capable of taking better advantage of network when there is possibility of overlap. In these results we see that the RDMA Read and RDMA Read with Interrupt perform equally well. This might be due to the fact that these applications do not require computation/communication overlap on the receiver side.



(a) MPI Wait Time for HPL

(b) MPI Wait Time for NAS SP

Figure 2.8: Application Level Evaluation for Rendezvous Protocol Designs

2.5 Summary

In this chapter, we have presented new designs which exploit the RDMA Read and the capability of generating selective interrupts to implement a high-performance Rendezvous Protocol. We have evaluated in detail the performance improvement offered by the new design in several different areas of high performance computing. We have observed that the

new designs can achieve nearly complete computation and communication overlap. Additionally, our schemes yield a 50% better communication progress rate when computation is overlapped with communication. Further, our application evaluation with Linpack (HPL) and NAS-SP (Class C) reveals that `MPI_Wait` time is reduced by around 30% and 28% respectively for a 36 node InfiniBand cluster. We observe that the gains obtained in the `MPI_Wait` time increase as the system size increases. This indicates that our designs have a strong positive impact on scalability of parallel applications.

CHAPTER 3

IMPROVING PERFORMANCE OF ALL-TO-ALL COMMUNICATIONS

MPI defines collectives which are blocking in nature, including `MPI_Alltoall`. Since applications will be waiting on the collective communication call to finish before proceeding with their computation, the latency of the collective operation is a very important metric for collective performance. In addition, the performance should scale well with the number of processes. In this Chapter, we describe work which aims to reduce overheads associated with collective operations and improve their performance scalability.

We approach the problem in two different methods. First, we remove extra software overheads incurred when MPI collective operations are based on top of MPI point-to-point operations (e.g. `MPI_Send`, `MPI_Recv`). Rather, we base our design directly on network-level primitives. Secondly, we exploit novel features offered by the InfiniBand network-interface, such as native support for non-contiguous communication. Using both these methods, we can reduce the number of memory copy operations required by each collective operation.

The rest of the Chapter is organized as follows. In Section 3.1, we provide necessary background information about this work. In Section 3.2, we describe current approaches and their limitations. This is followed by Section 3.3, where we give details of our proposed

solution. In Section 3.4, we evaluate our proposed design and finally in Section 3.5, we summarize our results from this Chapter.

3.1 Background

In this section we provide an overall background for our work in optimizing the performance and scalability of the `MPI_Alltoall` operation.

3.1.1 Overview of MPI All-to-All Operation and Existing Algorithms

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

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

There are several different algorithms to efficiently implement `MPI_Alltoall`. We describe two of the most popular ones: Hypercube based combining algorithm and the Direct Virtual Ring algorithm.

Hypercube Combining Algorithm

A hypercube is a multidimensional mesh of nodes with exactly two nodes in each dimension. A d-dimensional hypercube consists of $p = 2^d$ nodes. The All-to-All personalized

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

$$T_{hypercube} = (t_s + 1/2t_w mp)logp \tag{3.1}$$

Where,

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

Direct Virtual Ring

The direct algorithm is a straightforward way to exchange messages among all the processes. It assumes that all the processes have direct links connecting them. The processes are arranged in a virtual ring. Each process then sends its message to its neighbor. To avoid all the processes from sending to a single destination, the destinations are scattered among all of them, by using a modulus operation. Process $rank$ sends its message for process $(rank + i)\%p$, $\forall i, (0 \leq i \leq p)$. So, at every step, each process sends m bytes of data and it does it for $(p - 1)$ steps. Thus, the total time for an All-to-All exchange is:

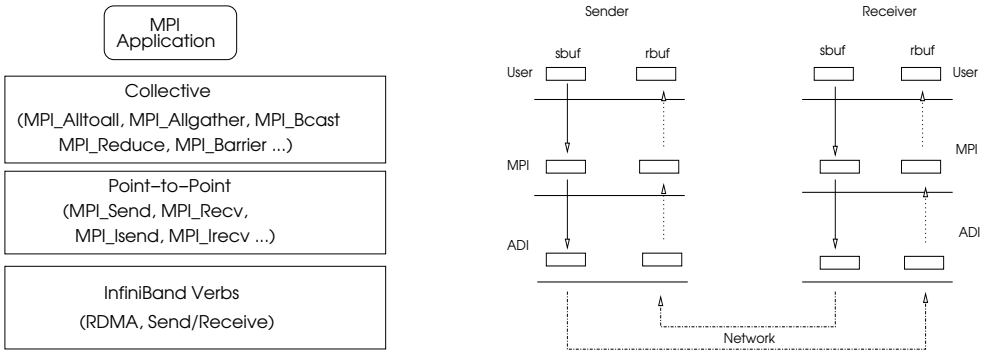
$$T_{direct-ring} = (p-1)t_s + t_w m(p-1) \tag{3.2}$$

3.2 Current Approaches and Limitations

Collective communications are a very important part of the MPI specification. These operations allow multiple MPI processes to participate in group communication. The algorithms used for collective communication in MVAPICH are based on the MPICH collective algorithms [44]. The collective operations are built on top of MPI point-to-point operations. ie. they follow the layered design structure shown in Figure 3.1(a). This layered design allows for the collective algorithms to execute on top of point-to-point operations (which are already based on RDMA). However, this layered design also means that the collective messages also incur the same overheads associated with point-to-point messages. Figure 3.1(b) illustrates this problem when a buffer is copied multiple times internally to perform an efficient algorithm which may require intermediate aggregation (such as hypercube or tree-based). In addition, the collective messages (if they are large enough) may be transferred over the Rendezvous protocol. In such circumstances, each collective message needs to undergo the Rendezvous handshake which may add unnecessary delays.

3.3 Proposed design for RDMA based All-to-All

In this section, we describe our work in optimizing the latency and scalability of the MPI All-to-all personalized communication operation. Broadly, we will take the approach shown in Figure 3.2. We aim to by-pass the layered structure of the current implementation in favor of a more direct design which leverages the InfiniBand features.



(a) Layered Design of Collective Operations

(b) Copy Overhead due to Layers

Figure 3.1: Layered Design of Collective Operations and Associated Overheads

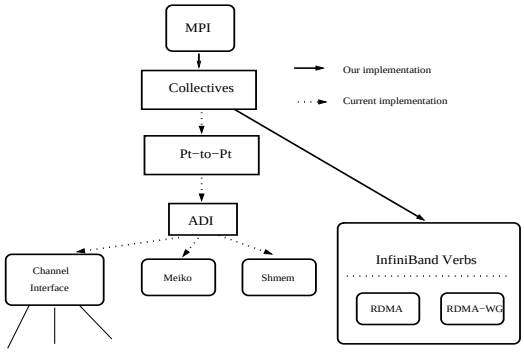


Figure 3.2: Proposed implementation path for Collectives

3.3.1 Design issues for RDMA Collectives

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

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

Message Arrival Detection: The RDMA Write operation in InfiniBand is totally transparent to the receiver process. The only way the receiver process can make out whether data has really arrived in the buffers is by polling the contents of a specific pre-defined memory location. For achieving this, there needs to be a persistent association of buffers at the sender and receiver end. Achieving a persistent association is relatively simple when pre-registered buffers are used. All the processes can implicitly decide on start and end buffer locations. Prior to the All-to-All communication the processes can reset the last bytes of the persistent buffers. Marking the completion of a RDMA write for direct application buffers, which are registered on-the-fly, is impossible using the earlier technique. Here, there is no unique value of the memory location that the MPI implementation can poll on. The application may choose to send any data value of its choice. Usually, in such a case, completion of a RDMA write operation can only be determined by using an explicit ACK. Hence, specific buffers need to be provided for collecting ACKs.

3.3.2 Design for Small Messages: HRWG

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

Now we look closely at the startup time cost t_s. There are various costs associated with message startup based on the implementation mechanism. If we decide to have a zero copy implementation, then the address exchange phase will dominate the message startup time. Also, we would have to pay on-the-fly registration cost. This can be comparably costlier than copying the data over to a pre-registered buffer. Hence, we choose a copy-based approach, implementing the Hypercube algorithm for small message sizes. However, our copy-based mechanism has only 2 data copies bypassing the MPI-level buffer in Figure 3.1(b) instead of the 4 copies in the point-to-point based implementation.

Buffer Management: We implement a buffer management scheme for the copy-based approach. In order to detect the arrival of an incoming message, we use memory polling. In our implementation, the collective communication buffer is created during the communicator initialization time. All processes in the communicator need to exchange addresses and memory handles for remote buffers. The collective communication buffer can then be divided into several parts. This can be done implicitly by all processes. There are two divisions of the buffer for supporting back-to-back collective calls. Also, we set up persistent associations with our $\log p$ neighbors of the hypercube. Figure 3.3 shows how these buffers are arranged.

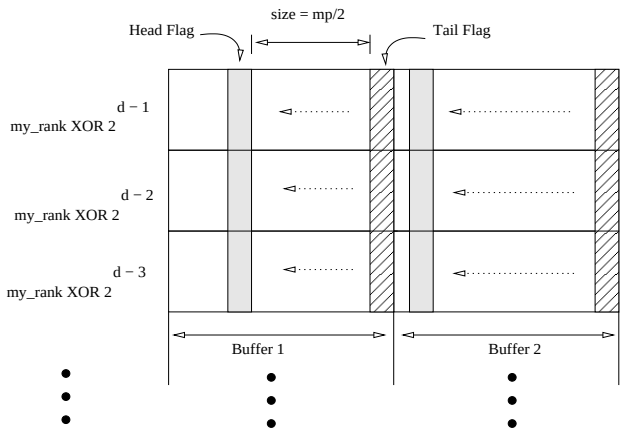


Figure 3.3: Buffer arrangement for Hypercube Algorithm

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

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

For implementing the Hypercube algorithm, we have to keep track of the buffers in transit and forward them correctly to the next dimension across which communication will take place. We can keep track of the buffers to send, by simply observing a pattern of communication within the hypercube. At every step of communication in the hypercube, a

process sends $p/2$ messages. The number of contiguous buffers from the received buffer-pool per peer is, $2^{(d-(d-i))}$, $\forall i$ in 0 to $(d-1)$. The buffer to be forwarded can be easily found out by maintaining a set of pointers at the middle of the receive buffers. The number of *middle* pointers is equal to the number of contiguous buffers to be forwarded from the persistent buffer associated with that dimension. After each iteration, the middle pointers can be moved either backwards or forwards depending on whether the rank of the destination process is greater than the rank of the sending process and vice-versa. The amount of data to be sent per *middle* pointer is recursively halved. Figure 3.4 gives a detailed view of how the *middle* pointers are managed.

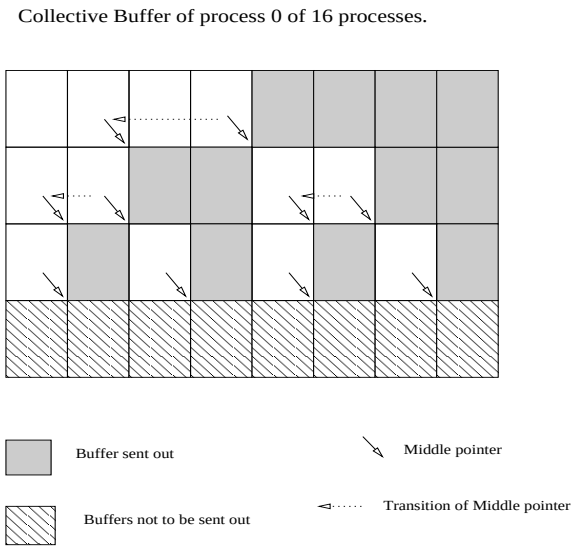


Figure 3.4: Managing Buffer pointers

Transfer Mechanisms: For transferring the buffers over the network, two different methods can be adopted. We can either transmit the buffer as one single RDMA or we can send the buffers individually using multiple RDMA writes. By using RDMA with gather

list, we can reduce the message startup time. Hence, we choose the RDMA Write with Gather for our implementation

3.3.3 Design for Large messages: DE

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

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

- Register the user buffer
- Exchange addresses of the user buffer
- Mark the completion with an explicit ACK

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

$$T_{direct-ring} = (p-1)t_{c-reg} + (p-1)mt_{w-reg} + (p-1)t_{rndz} + t_w m(p-1) \tag{3.3}$$

Where,

$$
\begin{aligned}
t_{c-reg} &= \text{Constant registration cost} \\
t_{w-reg} &= \text{Time to register one page} \\
t_{rndz} &= \text{Rendezvous exchange cost}
\end{aligned}
$$

In addition to multiple memory registrations, the *rendezvous* protocol presents a bottleneck at each of the $(p-1)$ steps of the Direct algorithm. Each message transfer has to be preceded by interaction of both sender and receiver. This takes away the advantage of RDMA, in that it is no longer truly one sided. The entire `MPI_Alltoall` is then bottlenecked. Instead, we adopted a new *Direct Eager* mechanism for implementing the All-to-All operation. In this new scheme, every process sends its receive buffer address to its next nearest neighbor, then the next one and so on in a ring-like manner. That is, process *rank* sends its address to $(rank+i)\%p$. $\forall i(0 \leq i \leq p)$. This phase is totally network parallelized as the addresses and memory handles are RDMA-ed to pre-registered buffers. Then, process *rank* waits for address from process $(rank+p-1-i)\%p$. $\forall i(0 \leq i \leq p)$. We note that the time spent waiting for the first address is almost negligible since the process i sends address to j first and j sends data to i first. Hence, we name our scheme as *Direct Eager*. Figure 3.5 shows one step the algorithm. This is repeated for $\forall i(0 \leq i \leq p)$ steps.

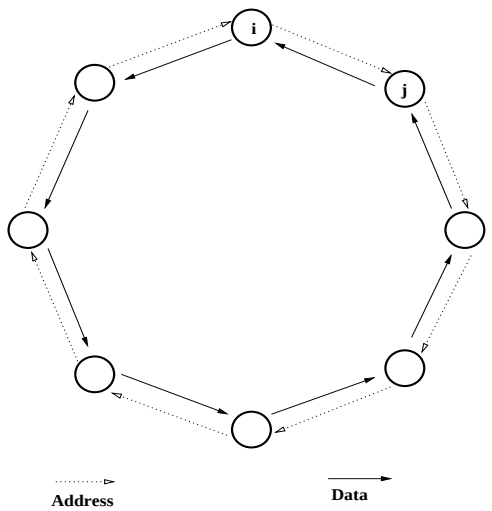


Figure 3.5: Direct Eager Mechanism

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

$$T_{direct-ring} = t_{c-reg} + (p-1)mt_{w-reg} + t_w m(p-1) \tag{3.4}$$

3.4 Performance Evaluation

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

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

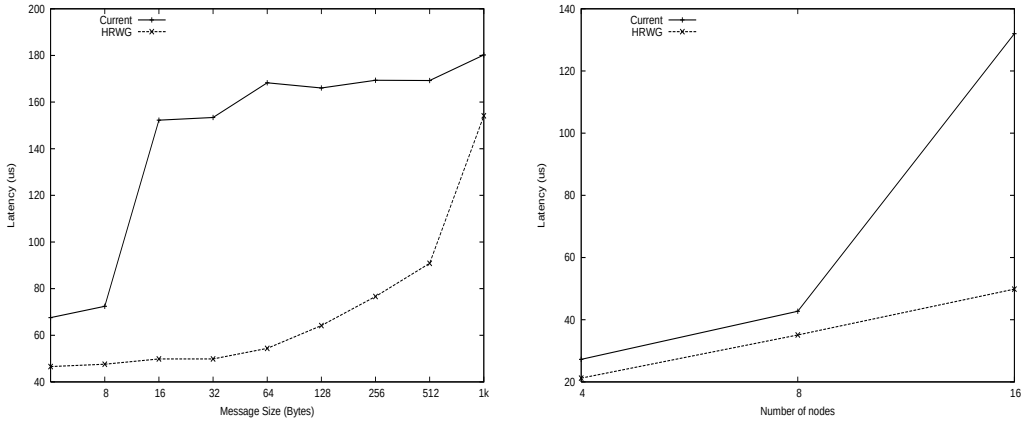
All the machines had Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected using the Mellanox InfiniScale 24 port switch MTS 2400. The Linux kernel version used was 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. It is to be noted that performance numbers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

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

3.4.1 Evaluation for Small Messages

We implemented the Hypercube algorithm (HRWG) for performing the All-to-All communication for small messages. Figure 3.6(a) compares the performance of the current

implementation and the proposed scheme. We observe that the proposed scheme (HRWG) has a 3.07 factor of improvement over the current implementation. The current implementation is limited to doing a point-to-point based communication due to the rise in copy-cost as a result of increase in the number of processes. Figure 3.6(b) shows the scalability of our implementation. We note that with increasing system size, it is indeed better to have a combining algorithm than a naive point-to-point based implementation.



(a) Small Messages on 16 Nodes (b) Small Messages (32 Bytes) Scalability

Figure 3.6: Small Message Performance Benefits for All-to-all Personalized Communication

3.4.2 Evaluation for Larger Messages

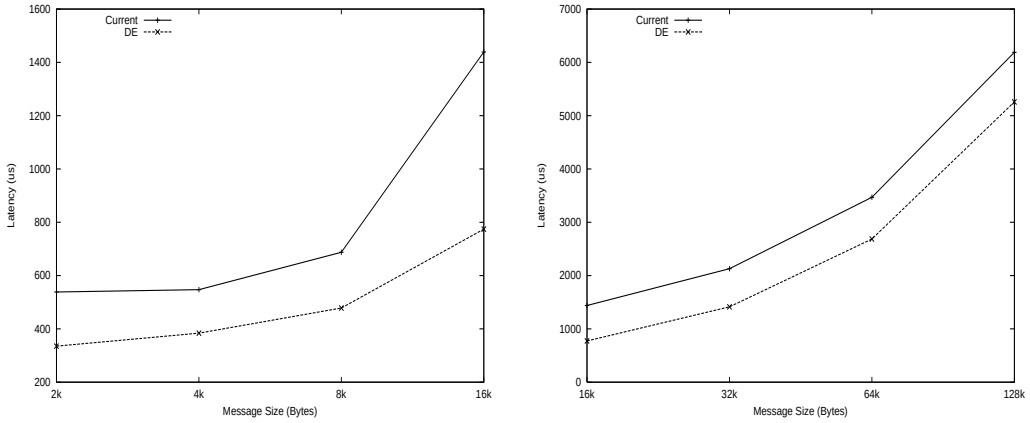
For medium and large messages we implement the Direct Virtual Ring based algorithm with the Direct Eager mechanism. For larger messages the current implementation falls back on a Pair-by-Pair Exchange algorithm implemented on `MPI_Sendrecv`. We observe that this algorithm is designed mainly with older generation networks where sending large messages indiscriminately in the fabric would lead to congestion. The current generation

InfiniBand switches are entirely non-blocking and provide cross-bar connectivity. Thus, it is no longer essential for us to fall back on Pair-by-Pair Exchange algorithm. The Direct Eager mechanism performs better than the current implementation. Figures 3.7(a) and 3.7(b) show the performance of our implementation compared to the current one. We note that DE performs better for medium sized messages. This is mainly because the total cost for *rendezvous* is comparable to that of the message transfer latency. Figures 3.7(c) and 3.7(d) show the scalability of our designs. We note that the difference between the current implementation and the proposed designs is in-fact growing as the number of processes increases.

3.4.3 Performance Extrapolation for Large Messages

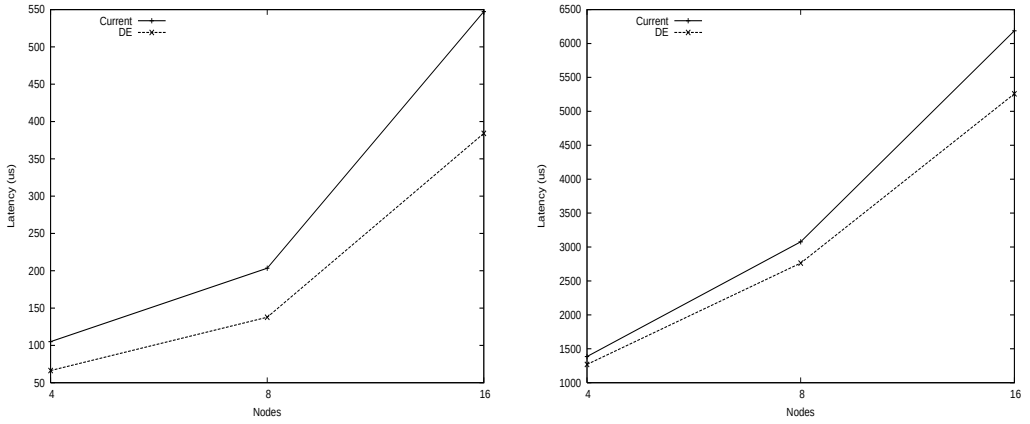
In this section, we try to extrapolate the performance of our *Direct Eager* mechanism to find out how much performance improvement we can expect over larger scale clusters.

We note from Equations (3) and (4), the *Direct Eager* mechanism avoids adding a cost of rendezvous that is linear according to the number of processes. We expect that this will show performance improvement when the All-to-All communication happens over a large cluster. In order to evaluate the benefits clearly, we assume 100% buffer re-use. That is, we try to eliminate the effects of the MVAPICH cache from the cost model. Including the cache will not lead to degradation of our implementation, since we use lesser (actually only one) cache entry per All-to-All communication as compared to $(p - 1)$ for current implementation. Using our extrapolation, we determine that a performance benefit of 77% can be obtained for an All-to-All communication of 4k message size among 1k nodes. The Figure 3.8 shows this extrapolation graph obtained from equations (3) and (4).



(a) Medium Messages on 16 Nodes

(b) Large Messages on 16 Nodes



(c) Large Messages (4k) Scalability

(d) Large Messages (128k) Scalability

Figure 3.7: Medium and Large Message Performance Benefits for All-to-all Personalized Communication

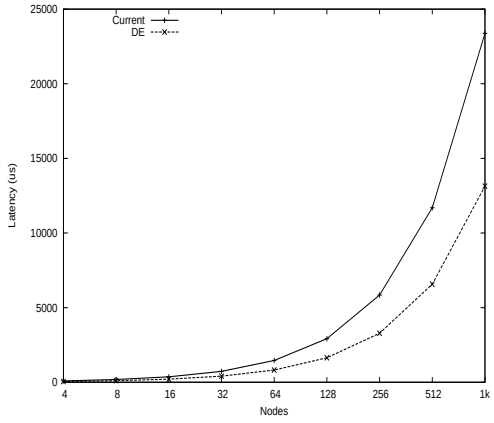


Figure 3.8: Performance for 4k message among 1k processes

3.5 Summary

In this Chapter, we presented new designs to take advantage of the advanced features offered by InfiniBand in order to achieve scalable and efficient implementation of the `MPI_Alltoall` collective. We proposed that the implementation of collectives be done directly on the InfiniBand Verbs Interface rather than using MPI level point-to-point functions. We evaluated in detail why MPI send receive calls are a hindrance to achieving good performance from collective operations. We detailed our design challenges and proposed two different schemes for small and large messages, HRWG and DE respectively. Our performance evaluation on a 16 node cluster shows that we can get an improvement of upto a factor of 3.07 for 32 byte messages. We studied the analytical models of our implementation, and our investigation shows that for a 1k node cluster, we can get a performance improvement of upto 64% for 4k messages.

CHAPTER 4

IMPROVING PERFORMANCE OF ALL-TO-ALL BROADCAST

The All-to-all broadcast (`MPI_Allgather`) is an important collective operation used in many applications such as matrix multiplication, lower and upper triangle factorization, solving differential equations, and basic linear algebra operations. InfiniBand provides powerful features such as Remote DMA (RDMA) which enables a process to directly access memory on a remote node. To exploit the benefits of this feature, we design collective operations directly on top of RDMA. In this Chapter we describe our design of the All-to-all Broadcast operation over RDMA which allows us to eliminate messaging overheads like extra message copies, protocol handshake and extra buffer registrations. Our designs utilize the basic choice of algorithms [44] and extend that for a high performance design over InfiniBand.

The proposed designs improve the latency of `MPI_Allgather` on 32 processes by 30% for 32 KB message size. Additionally, our RDMA design can improve the performance of `MPI_Allgather` by a factor of 4.75 on 32 processes for 32 KB message size, under no buffer reuse conditions. Further, our design can improve the performance of a parallel matrix multiplication algorithm by 37% on eight processes, while multiplying a 256x256 matrix.

The rest of the Chapter is organized as follows. In Section 4.1, we provide necessary background details for this work. In Section 4.2, we discuss the motivation for this work. In

Section 4.3, we present our design which aim to meet the current challenges. In Section 4.4, we evaluate our design and present associated results. Finally, in Section 4.5, we summarize the results from this Chapter.

4.1 Background

In this section we present the necessary background details required for our work in optimizing the scalability and performance of `MPI_Allgather` operation.

4.1.1 Overview of All-to-All Broadcast and Existing Algorithms

`MPI_Allgather` is an All-to-all broadcast collective operation defined by the MPI standard [31]. It is used to gather contiguous data from every process in a communicator and distribute the data from the jth process to the jth receive buffer of each process. `MPI_Allgather` is a blocking operation (i.e. control does not return to the application until the receive buffers are ready with data from all processes).

Several algorithms can be used to implement `MPI_Allgather`. Depending on system parameters and message size, some algorithms may outperform the others. Currently, MPICH [15] 1.2.6 uses the Recursive Doubling algorithm for power-of-two process numbers and up to medium message sizes. For non-power of two processes, it uses the Bruck's algorithm [8] for small messages. Finally, the Ring algorithm is used for large messages [44]. In this section, we provide a brief overview of the Recursive Doubling and Ring algorithms. We will use these algorithms in our RDMA based design.

Recursive Doubling: In this algorithm, pairs of processes exchange their buffer contents. But in every iteration, the contents collected during all previous iterations are also included in the exchange. Thus, the collected information *recursively doubles*. Naturally, the number of steps needed for this algorithm to complete is $log(p)$, where p is the number of processes.

The communication pattern is very dense, and involves one half of the processes exchanging messages with the other half. On a cluster which does not have constant bisection bandwidth, this pattern will cause contention. The total communication time of this algorithm is:

$$T_{rd} = t_s * log(p) + (p-1) * m * t_w \tag{4.1}$$

Where, t_s = Message transmission startup time, t_w = Time to transfer one byte, m = Message size in bytes and p = Number of processes.

Ring Algorithm: In this algorithm, the processes exchange messages in a ring-like manner. At each step, a process passes on a message to its neighbor in the ring. The number of steps needed to complete the operation is $(p-1)$ where p is the number of processes. At each step, the size of the message sent to the neighbor is same as the `MPI_Allgather` message size, m. The total communication time of this algorithm is:

$$T_{ring} = (p-1) * (t_s + m * t_w) \tag{4.2}$$

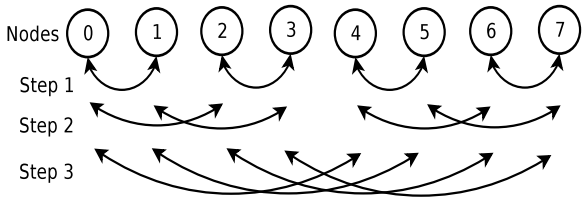


Figure 4.1: Recursive Doubling Algorithm for `MPI_Allgather`

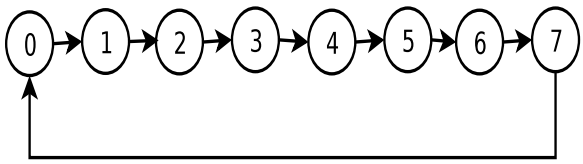


Figure 4.2: Ring Algorithm for `MPI_Allgather`

4.2 Can RDMA benefit Collective Operations?

Using RDMA, a process can directly access the memory locations of some other process, with no active participation of the remote process. While it is intuitive that this approach can speed up point-to-point communication, it is not clear how *collective* communications can benefit from it. In this section, we present the answer to this question and present the motivation of using RDMA for collective operations.

4.2.1 Bypass intermediate software layers

Most MPI implementations [15] implement MPI collective operations on top of MPI point-to-point operations. The MPI point-to-point implementation in turn is based on another layer called the ADI (Abstract Device Interface). This layer provides abstraction and can be ported to several different interconnects. The communication calls pass through several software layers before the actual communication takes place adding unnecessary overhead. On the other hand, if collectives are directly implemented on top of the InfiniBand RDMA interface, all these intermediate software layers can be bypassed.

4.2.2 Reduce number of copies

High-performance MPI implementations, MVAPICH [34], MPICH-GM [33] and MPICH-QsNet [38] often implement an eager protocol for transferring short and medium-sized messages. In this eager protocol, the message to be sent is copied into internal MPI buffers and is directly sent to an internal MPI buffer of the receiver. This causes two copies for each message transfer. For a collective operation, there are either $2*log(p)$ or $2*(p-1)$ sends and receives (every send has a matching receive). It is clear that as the number of processes in a collective grows, there are increasingly more and more message copies. Instead, with RDMA based design, messages can be directly transferred without undergoing several copies.

4.2.3 Reduce Rendezvous handshaking overhead

For transferring large messages, high-performance MPI implementations often implement the Rendezvous Protocol. In this protocol, the sender sends a `RNDZ_START` message. Upon its receipt, the receiver replies with `RNDZ_REPLY` containing the memory address of the destination buffer. Finally, the sending process sends the `DATA` message directly to the destination memory buffer and issues a `FIN` completion message. By using this protocol, zero-copy message transfer can be achieved.

This protocol imposes bottlenecks for MPI collectives based on point-to-point design. The processes participating in the collective need to continuously exchange addresses. However, these address exchanges are redundant. Once the base address of the collective communication buffer is known, the source process can compute the destination memory address for each iteration. This computation can be done locally by the sending process by calculating the array index for the particular algorithm and iteration number. Thus, for each iteration, RDMA can be directly used without any need for address exchange.

4.2.4 Reduce Cost of Multiple Registrations

InfiniBand, like most other RDMA capable interconnects, requires that all communication buffers be registered with the InfiniBand HCA. This "registration" actually involves locking of pages into physical memory and updating HCA memory access tables. After registration, the application receives a "memory handle" with keys which can be used by a remote process to directly access the memory. Thus, for performing each send or receive, the memory area needs to be registered.

Collective operations implemented on top of point-to-point calls would need to issue several MPI sends or receives to different processes (with different array offsets). This will cause multiple registration calls. For current generation InfiniBand software/hardware stacks, each registration has high setup overhead of around 90 μs. Thus, point-to-point implementation of collectives requires multiple registration calls with significant overhead. However, the RDMA based design would need only *one* registration call. The entire buffer passed to the collective call can be registered in one go. Thus, this will eliminate unnecessary registration calls.

4.3 Proposed Design for All-to-All Broadcast

In this section we describe our efforts for improving the latency and scalability of the MPI All-to-all broadcast operation (`MPI_Allgather`) based on RDMA operations. However, the algorithms are different in this work.

4.3.1 RDMA-based Design for Recursive Doubling

We propose a RDMA based design for Recursive Doubling (RD) algorithm. In RD, the size of the message exchanged by pairs of nodes doubles each iteration along with the

distance between the nodes. If m is the message size contributed by each process, the amount of data exchanged between two processes increases from m in the first iteration to $\frac{mp}{2}$ in the $\log(p)^{th}$ iteration. As we have said in previous sections, the optimal method to transfer short messages is copy based and for longer messages, we need to use zero copy. However, since in the RD algorithm, the actual message size in each iteration changes, we also have to dynamically switch between copy based and zero copy protocols to achieve an optimal design.

Hence, we switch between the two design alternatives at an iteration k ($1 \leq k \leq \log(p)$) such that the message size being exchanged, $2^{k-1}m$, crosses a fixed threshold M_T. The threshold M_T is determined empirically. Hence, message exchanges in the first k from $k+1$ through $\log(p)$ use a zero copy approach.

For performing the copy based approach, we need to maintain a pre-registered buffer. We call it "Collective Buffer". The design issues relating to maintaining this buffer and buffering schemes are described as follows:

Collective Buffer: This buffer is registered at communicator initialization time. Processes exchange addresses of their collective buffers also during that time. Some pre-defined space in the collective buffer is reserved to store the peer addresses and completion flags required for zero-copy data transfers. Data sent in any iteration comprises data received in all previous iterations along with the process' own message.

Buffering Scheme: In RD, data is always sent from and received to contiguous locations in either the collective buffer or the user's receive buffer. Since the amount of data written to a collective buffer cannot exceed M_T, the collective buffer never needs to be more than $2M_T$ which is 8 KB (ignoring space for peer addresses and completion flags) for a single Allgather call.

4.3.2 RDMA Ring for large messages

We implement the Ring algorithm for `MPI_Allgather` over RDMA only for large messages and large clusters. As observed in [44], large clusters may have better near-neighbor bandwidth. Under such scenarios, it is beneficial for `MPI_Allgather` to mainly communicate between neighbors. The Ring algorithm is ideal for such cases. Since we implement this algorithm for only large messages, we use a complete zero copy approach here. The design in this case is much simpler. The benefit of the RDMA-based scheme comes from the fact that we have a single buffer registration and a single address exchange performed by each node instead of p registrations, and $(p-1)$ address exchanges in the point-to-point based design. We use this Ring algorithm for messages larger than 1 MB and process numbers greater than 32.

4.4 Performance Evaluation

In this section, we evaluate the performance of our RDMA based design for All-to-all Broadcast. This operation is also called `MPI_Allgather`. We use three cluster configurations for our tests:

1. *Cluster A:* 32 Dual Intel Xeon 2.66 GHz nodes with 512 KB L2 cache and 2 GB of main memory. The nodes are connected to Mellanox MT23108 HCA using PCI-X 133 MHz I/O bus. The nodes are connected to Mellanox 144-port switch (MTS 14400).

2. *Cluster B:* 16 Dual Intel Xeon 3.6 GHz nodes (EM64T) with 1MB L2 cache and 4 GB of main memory. The nodes are connected to Mellanox MHES18-XT HCA using PCI-Express (x8) I/O bus.

3. *Cluster C:* 8 Dual Intel Xeon 3.0 GHz nodes with 512 KB L2 cache and 2 GB of main memory. The nodes are connected to the same InfiniBand network as Cluster A.

We have integrated our RDMA based design in the MVAPICH [34] stack. We refer to the new design as "MVAPICH-RDMA". The current implementation of `MPI_Allgather` over point-to-point is referred to as "MVAPICH-P2P". Our experiments are classified into three types. First, we demonstrate the latency of our new RDMA design. Secondly, we investigate performance of the new design under low buffer re-use conditions. Finally, we evaluate the impact of our design on a Matrix Multiplication application kernel which uses All-to-all broadcast.

4.4.1 Latency benchmark for `MPI_Allgather`

In this experiment, we measure the basic latency of our `MPI_Allgather` implementation. All the processes are synchronized with a barrier and then `MPI_Allgather` is repeated 1000 times, using the same communication buffer. The results are shown in Figures 4.3 and 4.4 for Cluster A and in Figures 4.5 for Cluster B. The results from both Clusters A and B follow the same trends. The results are explained as follows:

Small Messages: The RDMA based design can avoid the various copy and layering overheads in different layers of the MPI point-to-point implementation. The results indicate that latency can be reduced by 17%, 13% and 15% for 16 processes on Cluster A (Fig 4.3(a)), 32 processes on Cluster A (Fig 4.4(a)) and 16 processes on Cluster B (Fig 4.5(a)) for 4 byte message size, respectively.

Medium Messages: For medium sized messages, the point-to-point based design required rendezvous address exchange for transferring messages at every step of the algorithm. However, for the RDMA based `MPI_Allgather`, no such exchange is required. We note that the

number of steps increases as the number of processes, and so does the cumulative cost of address exchange. Our RDMA based design is able to successfully avoid this increasing cost.

The results indicate that latency can be reduced by 23%, 30% and 37% for 16 processes on Cluster A (Fig 4.3(b)), 32 processes on Cluster A (Fig 4.4(b)) and 16 processes on Cluster B (Fig: 4.5(b)) for 32 KB message size, respectively.

Large Messages: Large messages are also transferred using the same zero copy technique used for medium sized messages. Hence, the same address exchange cost can be saved (as described in the previous case). However, since the message sizes are large, the address exchange forms a lesser portion of the overall cost of `MPI_Allgather`. The results for large messages indicate that latency can be reduced by 7%, 6% and 21% for 16 processes on Cluster A (Fig 4.3(c)), 32 processes on Cluster A (Fig: 4.4(c)) and 16 processes on Cluster B (Fig 4.5(c)) for 256 KB message size, respectively.

Scalability: We plot the `MPI_Allgather` latency numbers with varying process counts, for a fixed message size to see the impact of RDMA design on scalability. Figure 4.6(a) shows the results for 32 KB message size. We observe that as the number of processes increase, the gap between the point-to-point implementation and RDMA design increases. This is due to the fact that the RDMA design eliminates the need for address exchange (which increases as the number of processes).

4.4.2 `MPI_Allgather` latency with no buffer reuse

In the above experiment, we measured the latency of `MPI_Allgather` when utilizing the same communication buffers for a large number of iterations. The cost of registration was thus amortized over all the iterations by the registration cache maintained by MVAPICH. However, it is not necessary that all MPI applications will always reuse their buffers. In the

case where applications use `MPI_Allgather` with different buffers, the point-to-point based design will be forced to register the buffers separately, thus incurring high cost. The cost of just memory registration is shown in Figure 4.6(b). We observe that memory registration is in fact quite costly.

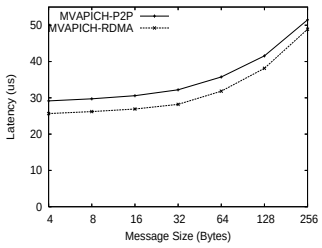
In the following experiment, we conduct the same latency test (as mentioned in previous section), but the buffers used for each iteration are different. Figures 4.7(a) and 4.7(b) show the results for Clusters A and B, respectively. The RDMA based `MPI_Allgather` performs 4.75 and 3 times better for Cluster A and B for 32 KB message size, respectively.

4.4.3 Matrix Multiplication Application Kernel

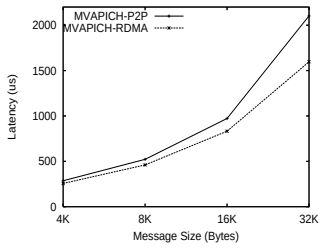
In the previous sections, we have seen how the RDMA design impacts the basic latency of `MPI_Allgather`. In order to evaluate the impact of this performance boost on end-MPI applications, we build a distributed-memory Matrix Multiplication routine over the optimized BLAS provided by the Intel Math Kernel Library [19]. We use a simple row-block decomposition for the data. In each iteration, the matrix multiplication is repeated with a fresh set of buffers. This application kernel is run on Cluster C using 8 processes. We observe that using our RDMA design, the application kernel is able to perform 37% better for an array size of 256x256, as shown in Figure 4.7(c).

4.5 Summary

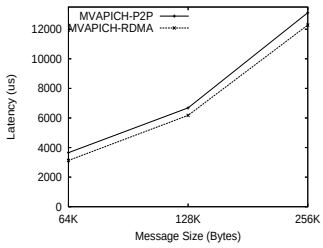
In this Chapter, we proposed a RDMA based design for the All-to-all Broadcast collective operation. Our design reduces software overhead, copy costs, protocol handshake – all required by the implementation of collectives over MPI point-to-point. Performance evaluation of our designs reveals that the latency of `MPI_Allgather` can be reduced by 30% for 32 processes and a message size of 32 KB. Additionally, the latency can be improved by a



(a) Small Messages

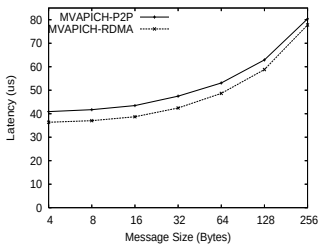


(b) Medium Messages

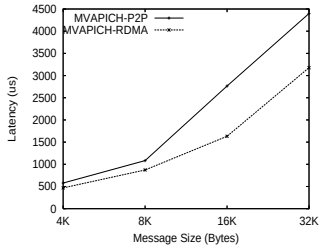


(c) Large Messages

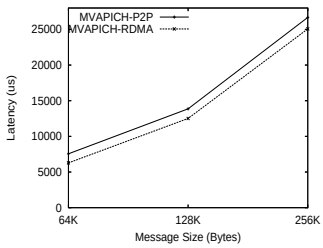
Figure 4.3: MPI_Allgather Performance on 16 Processes (Cluster A)



(a) Small Messages

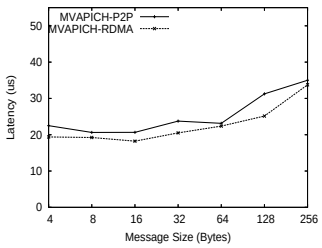


(b) Medium Messages

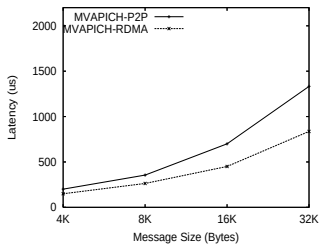


(c) Large Messages

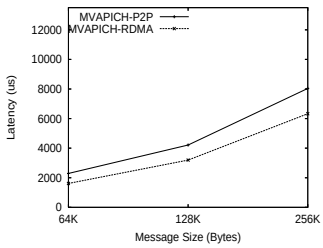
Figure 4.4: MPI_Allgather Performance on 32 Processes (Cluster A)



(a) Small Messages

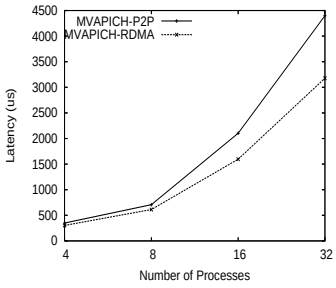


(b) Medium Messages

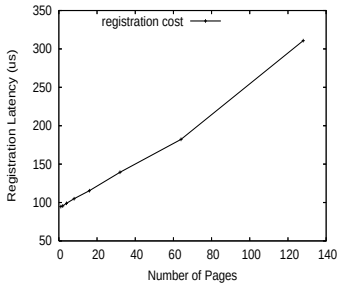


(c) Large Messages

Figure 4.5: MPI_Allgather Performance on 16 Processes (Cluster B)

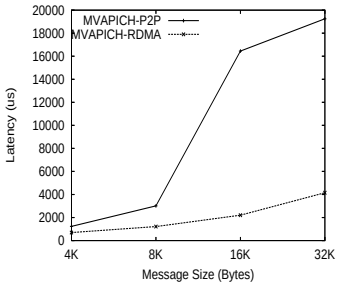


(a) Scalability of RDMA Design for 32 KB message size

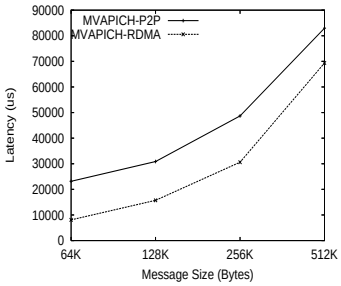


(b) Cost of Registration

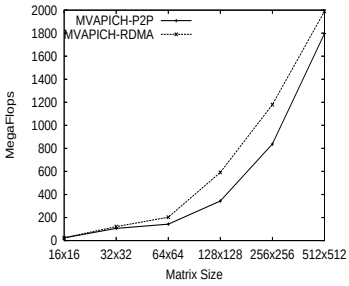
Figure 4.6: Scalability and Registration Cost on Cluster A



(a) No Buffer Reuse (Cluster A)



(b) No Buffer Reuse (Cluster B)



(c) Matrix Multiplication

Figure 4.7: Impact of Buffer Registration and Performance of Matrix Multiplication

factor of 4.75 under no buffer reuse conditions for the same process count and message size. Further, our design can speed up a parallel matrix multiplication algorithm by 37% on 8 processes, while multiplying a 256x256 matrix. The presented designs are expected to yield benefit for very large InfiniBand clusters.

CHAPTER 5

SCALABLE COMMUNICATION BUFFER MANAGEMENT TECHNIQUES

MVAPICH uses a reliable connection oriented model provided by InfiniBand. This model provides superior performance on current generation InfiniBand stacks than the unreliable connection less model as well as providing reliable transport. However, one of the restrictions of using a connection oriented model is that messages can be received only in buffers which are already available to the Host Channel Adapter (HCA) or Network Interface Card (NIC). In order to achieve this, MVAPICH allocates and dedicates buffers for each connection (the number of connections increases as the number of processes). Although the amount of buffers allocated per connection can be tuned and MVAPICH has scaled quite well for contemporary clusters (up to 1000 nodes and beyond), the challenges imposed by the scale of next generation very large clusters (up to 10,000 nodes and beyond) is quite hard to meet with the current buffer management model.

The latest InfiniBand standard (Release 1.2) [17] has provided a new feature called *Shared Receive Queues* (SRQ) which aims at solving this scalability issue at the HCA level. This new feature removes the requirement that message buffers be dedicated for each connection. Using this feature, a process which intends to receive from multiple processes can in fact

provide receive buffers in a single queue. The HCA uses these buffers in an FCFS manner for incoming messages from all processes.

In this Chapter, we carry out detailed analysis of the design alternatives and propose a high-performance MPI design using SRQ. We propose a novel flow control mechanism using a "low watermark" based approach. In addition, we design a mechanism which can help users fine tune our designs on their specific platforms. Further, we come up with an analytical model which can predict memory usage by the MPI library on clusters of tens-of-thousands of nodes. Verification of our analytical model reveals that our model is accurate within 1% error margin. Based on this model, our proposed designs will take only $1/10^{th}$ the memory requirement as compared to the default MVAPICH distribution on a cluster sized at 16,000 nodes. Performance evaluation of our design on our 8-node PCI-Express cluster shows that our new design was able to provide the same performance as the existing design utilizing only a fraction of the memory required by the existing design. In comparison to tuned existing designs, our design showed a 20% and 5% improvement in execution time of NAS Benchmarks (Class A) LU and SP, respectively. The High Performance Linpack [2] was able to execute a much larger problem size using our new design, whereas the existing design ran out of memory.

The rest of the chapter is organized as follows. In Section 5.1, we provide an overview of Shared Receive Queues in InfiniBand. In Section 5.2 we discuss the current approaches and their limitations, particularly when scaling to very large clusters. Then in Section 5.3, we describe the potential benefits of using shared receive queues, followed by design alternatives and a proposed design in Section 5.4. In Section 5.5, we evaluate our proposed design and summarize this chapter in Section 5.6.

5.1 Overview of Shared Receive Queues

InfiniBand provides several types of transport services: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD) and Unreliable Datagram (RD). RC and UC are connection-oriented and require one QP to be connected to exactly one other QP. On the other hand, RD and UD are connection less and one QP can be used to communicate with many remote QPs. To the best of our knowledge, Reliable Datagram (RD) transport has not been implemented by any InfiniBand vendor yet.

On top of these transport services, IBA provides software services. However, all software services are not defined for all transport types. Figure 5.1 depicts which software service is defined for which transport, as of IBA specification release 1.2. As shown in the figure, the send/receive operations are defined for all classes of transport. For connection-oriented transport, a new type of software service called Shared Receive Queue (SRQ) has been introduced. This allows the association of many QPs to one receive queue even for connection oriented transport. Thus, any remote process which is connected by a QP can send a message which is received in buffers specified in the SRQ.

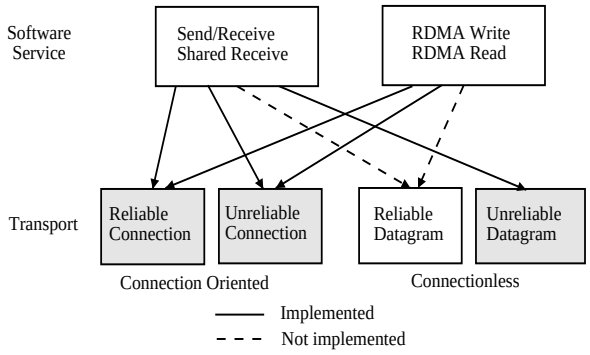


Figure 5.1: IBA Transport and Software Services

5.2 Current Approaches and Limitations

MVAPICH is based on the connection-oriented reliable transport of InfiniBand utilizing both RDMA and Send/Receive channels. In order to communicate using these channels, it has to allocate and dedicate buffers to each remote process. This means that the memory consumption grows linearly with the number of processes. Although the number of buffers per process can be tuned (at runtime), and MVAPICH has scaled well for contemporary InfiniBand clusters, the next-generation InfiniBand clusters are in the order of tens-of-thousands of nodes. In order for MVAPICH to scale well for these clusters, the linear growth of memory requirement with number of processes has to be removed.

Adaptive buffer management is a mechanism by which the MPI can control the amount of buffers available for each connection during runtime based on message patterns. However, there are several problems with this mechanism when implemented on top of the Send/Receive and RDMA channels:

- **Send/Receive Channel:** This channel allows us to choose how many buffers are posted on it dynamically. However, buffers once posted on a receive queue cannot be recalled. Hence, posted buffers on idle connections lead to wasted memory. This problem exacerbates memory consumption issues in large scale applications that run for a very long time. In addition, if MVAPICH is very aggressively tuned to run with low number of buffers per Send/Receive channel, this will lead to performance degradation. This is because the Send/Receive channel is based on window-based flow control mechanism [25]. Reducing the window in order to reduce memory consumption hampers the message passing performance.

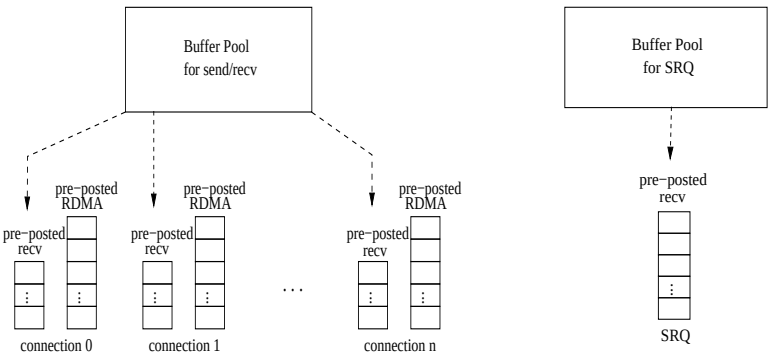
- **RDMA Channel:** This channel allows very low-latency message passing. However, the allocation of buffers for every connection is very rigid. The cyclic window of buffers needs to be contiguous memory. If not, then another round of address and memory key exchange (extra overhead) is required. Recalling of RDMA buffers is possible from any connection, but there is an additional overhead of informing remote nodes about the reduced memory they have with the receiving process. This process can lead to some race conditions which have to be eliminated using further expensive atomic operations, thus, leading to high overheads.

Thus, in order to improve the buffer usage scalability of MPI while preserving high-performance we need to explore a different communication channel.

5.3 Benefits of using Shared Receive Queues

Since we aim to remove the dependence of number of communication buffers with the number of MPI processes, we need to look at connection less models. As described in earlier Sections, InfiniBand provides two kinds of connection less transport. One is Reliable Datagram (RD) and the other is Unreliable Datagram (UD). Unfortunately, Reliable Datagram is not implemented in any InfiniBand stack (to the best of our knowledge), so that rules out this option. UD can provide the scalable features, but the MPI design would now have to provide reliability. This will add to the overall cost of message transfers, and may result in loss of high-performance. In addition, UD does not support RDMA features, which are needed for zero-copy message transfer, thus further degrading performance.

Shared Receive Queues (SRQ) provides a model to efficiently share receive buffers across connections whilst maintaining the good performance and reliability of a connection oriented transport. Thus, the SRQ is a good candidate for achieving scalable buffer management.



(a) Existing MVAPICH Buffer Organization

(b) Proposed SRQ Based Buffer Organization

Figure 5.2: Comparison of Buffer Management Models

Figure 5.2 shows the difference between the buffer organization schemes for MVAPICH and the new proposed design based on SRQ.

5.4 MPI Design Alternatives using Shared Receive Queues

In this section, we present our research towards a highly resource scalable design of MPI over InfiniBand. The basic idea is to use the Shared Receive Queue feature and to design flow control mechanisms around it.

The SRQ mechanism achieves good buffer scalability by exposing the same set of receive WQEs to all remote processes on a first come first serve (FCFS) basis. However, in this mechanism, the sending process lacks a critical piece of information: number of available receive buffers at the receiver. In the absence of this information, the sender can overrun the available buffers in the SRQ. To achieve optimal message passing performance, it is critical that this situation is avoided. In the following sections, we propose our novel design which

enables the benefits provided by SRQ, while avoiding senders from over-running receive buffers.

5.4.1 Proposed SRQ Refilling Mechanism

A high-performance MPI design often requires the MPI progress engine to be polling to achieve the lowest possible point-to-point latency. MVAPICH is thus based on a polling progress engine. Ideally, we would like to maintain the polling nature of the MPI for the SRQ based design. However, in this polling based design, MPI can only discover incoming messages from the network when explicit MPI calls are made. This increases the time intervals in which MPI can check the state of the SRQ. Moreover, if the MPI application is busy performing computation or involved in I/O, there can be prolonged periods in which the state of SRQ is not observed by the MPI. In the meantime, the SRQ might have become full. In order to efficiently utilize SRQ feature, we must avoid this situation. Broadly, three design alternatives can be utilized: Explicit acknowledgement from receiver, Interrupt based progress and Selective interrupt based progress.

- **Explicit Acknowledgement from Receiver:** In this approach, the sending processes can be instructed to refrain from sending messages to a particular receiver unless they receive an explicit `OK_TO_SEND` message after every k messages. Arrival of the `OK_TO_SEND` message means that the receiver has reserved k buffers in a dedicated manner for this sender and allows the sender to send k more messages. Where k is a threshold of messages that can be tuned or selected at runtime. This scheme can avoid the scenario in which the sender completely fills up the receiver queue with messages. This scheme is illustrated in Figure 5.3. However, this scheme suffers from a couple of critical deficiencies:

1. **Waste of Receive Buffer:** Since in this design alternative we reserve k buffers for a specific sender if the sender does not have more messages to send the memory resource for the reserved buffers can be wasted. To prevent this problem, if we reduce the value of k, we cannot achieve high bandwidth because the sender should wait the `OK_TO_SEND` message for every few messages.

2. **Early throttling of senders:** Even though not all senders may be transmitting at the same time, a particular sender may send k messages and then be throttled until the receiver sends the `OK_TO_SEND` message. If the receiver is busy because of a computation, the sender blocks until the receiver operates the progress engine and sends the `OK_TO_SEND` message.

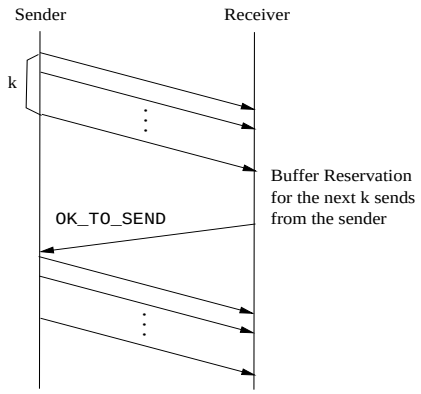


Figure 5.3: Explicit ACK mechanism

- **Interrupt Based Progress:** As mentioned earlier in this section, if the MPI application is busy performing computation or I/O, it cannot observe the state of the SRQ. In this design approach, the progress engine of the MPI is modified so as to explicitly request an interrupt before returning execution control to the MPI application. If there is an arrival of a new message, the interrupt handler thread becomes active and

processes the message along with refilling the SRQ. Figure 5.4 illustrates this design alternative.

This approach can effectively avoid the situation where the SRQ is left without any receive WQEs. However, this approach also has a limitation. There is now an interrupt on arrival of any new message when the application is busy computing. This can cause increased overhead and lead to non-optimal performance. In addition, we note that the arrival of the next message as such is not a critical event. There may be several WQEs still available in the SRQ. Hence, most of the interrupts caused by this mechanism will be unnecessary.

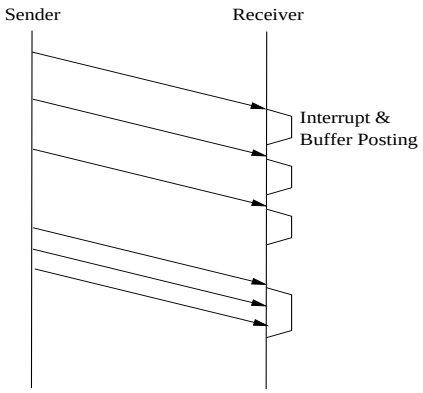


Figure 5.4: Interrupt Based Progress

- **Selective Interrupt Based Progress:** In this design alternative, we try to minimize the number of interrupts to the bare minimum. InfiniBand provides an asynchronous event associated with a SRQ called `SRQ_LIMIT_REACHED`. This asynchronous event is fired when a low "watermark" threshold (preset by the application) is reached. This event allows the application to act accordingly. In our case, we can utilize this event to trigger a thread to post more WQEs in the SRQ. Figure 5.5 demonstrates the sequence

of operations. In step 1, the remote processes send messages to the receiver. In step 2, the arrival of a new message causes the SRQ WQE count to drop below the limit (as shown by the grayed out region of the SRQ). In step 3, the thread designated to handle this asynchronous event (called LIMIT thread from now on) becomes active. In step 4, the LIMIT thread posts more WQEs to the SRQ. It should be noted that as soon as a SRQ WQE is consumed it is directly moved to the completion queue (CQ) by the HCA driver.

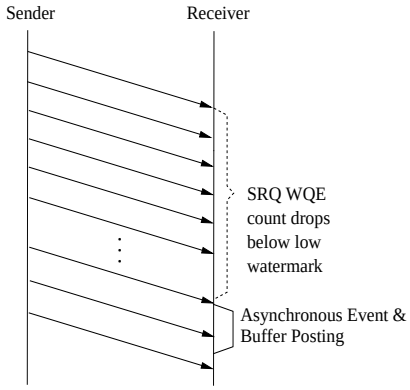


Figure 5.5: SRQ Limit Event Based Design

This design alternative meets our design criteria and causes minimum interference with the MPI application. Hence, we choose this design alternative for our SRQ based MPI design.

5.4.2 Proposed Design of SRQ Limit Threshold

As mentioned earlier in this section, we utilize the `SRQ_LIMIT_REACHED` asynchronous event provided by InfiniBand. This event is fired when a preset limit is reached on the SRQ. In order to achieve an optimal design, we need to make sure that the event is: a) not fired too often and b) has enough time to post buffers so that SRQ is not left empty.

In order to calculate a reasonable low watermark limit, we need to find out the rate at which the HCA can fill up receive buffers. We can find out this information in a dynamic manner by querying the HCA. In addition to that, we need to find out the time taken by the LIMIT thread to become active. For finding out this value, we design an experiment, as illustrated in Figure 5.6. In this experiment, we measure the round-trip time using SRQ (marked as t_1). The subsequent message triggers the SRQ LIMIT thread which replies back with a special message. We mark this time as t_2. The LIMIT thread wakeup latency is given by: $(t_2 - t_1)$. On our platform, this is around 12μs.

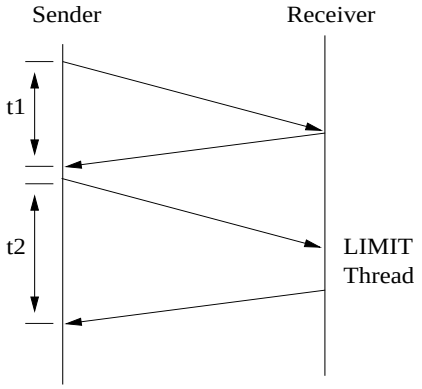


Figure 5.6: LIMIT Thread Wakeup Latency

Thus, we can calculate the minimum low watermark limit as:

$$Watermark = \frac{BW * 10^3}{MinPacketSize} * t_{wakeup} \tag{5.1}$$

Where, BW is the maximum bandwidth supported by the HCA is Gb/s, $MinPacketSize$ is the minimum packet size of MPI messages in bits and t_{wakeup} is the time taken by the LIMIT thread to wake up in microseconds. For our experimental platform, the $Watermark$ value is 300. In addition to the MPI library, another utility will be distributed which can

automatically calculate the value of the $Watermark$ value on the MPI library user's platform. The user can then simply plug in this value in the MPI application's environment, from where it will be picked up by the MPI library.

5.4.3 Analytical Model for Memory Usage Estimation

In order to fully understand the impact of the memory usage model of our proposed SRQ based design, we construct an analytical model of the memory consumption by MPI internal buffers.

There are several components of the memory consumed during startup. The major components are memory consumed by the Buffer Pool, RDMA channel, Send/Receive channel and the memory consumed by the InfiniBand RC connections themselves.

The size of the Buffer Pool is given by the product of the number of buffers in the pool and the size of each buffer.

$$M_{bp} = N_{pool} * S_{buf} \tag{5.2}$$

Where, M_{bp} is the amount of memory consumed by the Buffer Pool, N_{pool} is the number of buffers in the pool and S_{buf} is the size of each buffer.

The memory consumed by MVAPICH-SR (tuned version of MVAPICH using only Send/Receive channel) is composed of three parts, the memory consumed by the Buffer Pool and the memory consumed by each connection and the Send/Receive buffers.

$$M_{sr} = M_{bp} + (M_{rc} + N_{sr} * S_{buf}) * N_{conn} \tag{5.3}$$

Where, M_{sr} is the amount of memory consumed by MVAPICH-SR, M_{rc} is the memory needed for each InfiniBand connection by HCA driver, N_{sr} is the number of Send/Receive buffers for each connection and N_{conn} is the total number of connections.

MVAPICH-RDMA (default version of MVAPICH using both RDMA and Send/Receive channels) consumes all the memory as MVAPICH-SR and in addition, allocates RDMA buffers for each connection. The RDMA channel also needs to keep dedicated send buffers per connection [26]. Hence, the amount of dedicated buffers per connection doubles.

$$M_{rdma} = M_{sr} + 2 * N_{rdma} * S_{buf} * N_{conn} \tag{5.4}$$

Where, M_{rdma} is the amount of memory consumed by MVAPICH-RDMA and N_{rdma} is the number of RDMA buffers per connection.

Finally, the MVAPICH-SRQ (our proposed SRQ based design) only needs to allocate the Buffer Pool and a fixed number of buffers for posting to the SRQ.

$$M_{srq} = M_{bp} + M_{rc} * N_{conn} + N_{srq} * S_{buf} \tag{5.5}$$

Where, M_{srq} is the memory consumed by MVAPICH-SRQ and N_{srq} is the number of SRQ buffers.

Analyzing Equations 5.3, 5.4 and 5.5, we observe that the memory requirement by MVAPICH-SRQ is much lesser if the number of connections is very large. We plug in practical values to the above parameters and analyze the reduction in memory usage while using a SRQ based design. Our analysis reveals that for a cluster with 16,000 nodes, M_{rdma} is around 14 GigaBytes, whereas M_{srq} is only 1.8 GigaBytes.

5.5 Performance Results

In this section we evaluate the memory usage and performance of our MPI with SRQ design over InfiniBand. We first introduce the experimental environment, and then compare our design with MVAPICH in terms of memory usage and application performance. We also show the importance of flow control in using SRQ.

The default configuration of MVAPICH is to use a set of pre-registered RDMA buffers for small and control messages. In our performance graphs we call this configuration "MVAPICH-RDMA". MVAPICH can also be configured to use "Send/Receive" buffers for small and control messages. We also compared with this configuration, and it is called "MVAPICH-SR" in the graphs. We have incorporated our design into MVAPICH, and it is called "MVAPICH-SRQ".

5.5.1 Experimental Environment

Our test bed cluster consists 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. The Linux kernel used here is version 2.6.13.1.

5.5.2 Startup Memory Utilization

In this section we analyze the startup memory utilization of our proposed designs as compared to MVAPICH-RDMA and MVAPICH-SR. In our experiment, the MPI program starts up and goes to sleep after `MPI_Init`. Then we use the UNIX utility *pmap* to record the total memory usage of any one process. The same process is repeated for MVAPICH-RDMA, MVAPICH-SR and MVAPICH-SRQ. The results are shown in Figure 5.7.

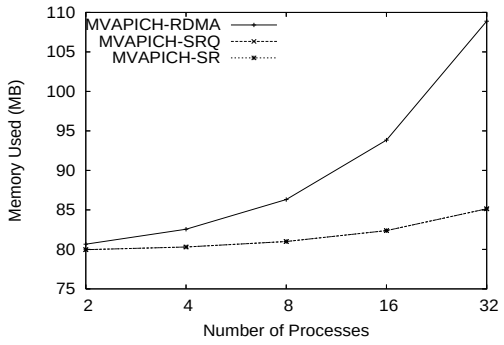


Figure 5.7: Memory Utilization Experiment

Observing Figure 5.7, we can see that MVAPICH-RDMA scheme consumes the most memory. Since the RDMA buffers are dedicated to each and every connection, the memory requirement grows linearly as number of processes. On the other hand, the MVAPICH-SR, as described earlier in this section is a highly-tuned version of MVAPICH using Send/Receive channel. This uses the same amount of memory as MVAPICH-SRQ. This is because the number of actual processes running is not enough for the per connection buffer posting to empty the *Buffer Pool*. If the number of processes is increased to a few hundred, then MVAPICH-SR will consume more memory than MVAPICH-SRQ. MVAPICH-SRQ just requires the same Buffer Pool and a *fixed* number of buffers which are posted on the SRQ. This number does not grow with the number of processes.

In Section 5.4.3, we have developed an analytical model for predicting the memory consumption by MVAPICH-RDMA, MVAPICH-SR and MVAPICH-SRQ on very large scale systems. In this section, we will first validate our analytical model and then use this to extrapolate memory consumption numbers on much larger scale systems.

On our experimental platform and MVAPICH configuration, the values of these parameters are: $N_{rdma} = 32$, $N_{sr} = 10$, $N_{pool} = 5000$, $S_{buf} = 12$KB, $M_{rc} = 88$KB. In addition, we have measured a constant overhead of 20MB which is contributed by various other libraries

(including lower-level InfiniBand libraries) required by MVAPICH. It is to be noted that in the experiment, N_{sr} is simply taken from the Buffer Pool, so this factor does not show up. In Figure 5.8 we show the error margin of our analytical model with the measured data. We observe that our analytical model is indeed quite accurate.

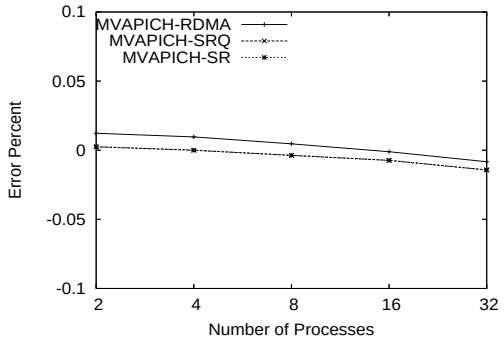


Figure 5.8: Error Margin of Analytical Model

Now we use this model to predict the memory consumption on much larger scale clusters. By increasing the number of connections and using the above mentioned parameter values, we extrapolate the memory consumption for each of the three schemes. The results are shown in Figure 5.9.

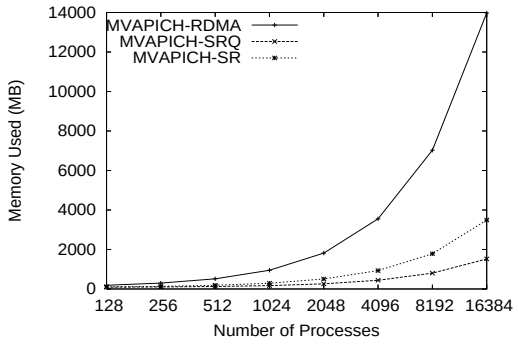


Figure 5.9: Estimation of memory consumption on very large clusters

5.5.3 Flow Control

In this section we present the importance of having flow control in using SRQ. We designed a micro-benchmark to illustrate it. The benchmark includes two nodes. The receiver first posts non-blocking receives (`MPI_Irecv`), and then starts computing. While the receiver is busy computing, the sender sends a "burst size" number of messages to the receiver. After the receiver finishes computing, it calls `MPI_Waitall` to finally get all the messages. We record the time the receiver spends in `MPI_Waitall` as an indication of how well the receiver can handle the incoming messages while it is computing.

Figure 5.10 shows the experimental results. We used the selective interrupt based approach for flow control as described in previous sections. We can easily see from the graph that MVAPICH-SRQ without flow control can handle messages as well as MVAPICH-SRQ with flow control up to burst size around 250. After that, the line of MVAPICH-SRQ without flow control goes up steeply, which means the performance becomes very bad. As we discussed in previous sections, without flow control the receiver can only update (refill) the SRQ when it calls the progress engine. In this benchmark, since the receiver is busy computing, it has no means to detect the SRQ is full, so the incoming messages get silently dropped. Only after computation, the receiver can resume to receive messages, but it has already lost computation/communication overlap and the network traffic becomes messy because of the sender retries. MVAPICH-SRQ with flow control, however, can handle a large "burst size" number of messages, and it doesn't add much overhead. In later sections MVAPICH-SRQ refers to MVAPICH-SRQ with selective interrupt based flow control.

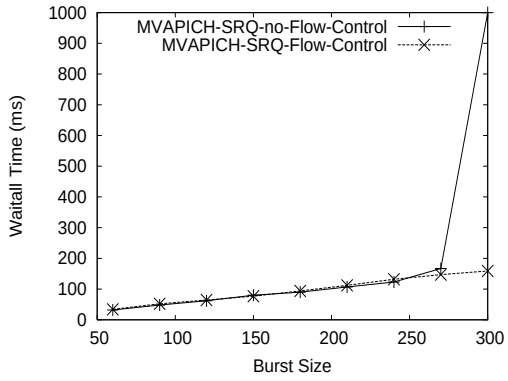


Figure 5.10: MPI_Waitall Time comparison

5.5.4 NAS Benchmarks

In this section we present the performance of MVAPICH-SRQ by using NAS Parallel Benchmarks [6], Class A. We conducted experiments on 16 processes. Figures 5.11 and 5.12 show the total execution time of MVAPICH-RDMA, MVAPICH-SR, and MVAPICH-SRQ.

From these two graphs we can see that for all benchmarks MVAPICH-SRQ performs almost exactly the same as MVAPICH-RDMA, which means using MVAPICH-SRQ we can dramatically reduce memory usage while not sacrificing performance at all. Looking at MVAPICH-SR, however, we can see that for LU, it performs 20% worse than MVAPICH-SRQ. This is because LU uses a lot of small messages, and in MVAPICH-SR, the sender will be blocked if it doesn't have enough credits from the receiver. This is not a problem in MVAPICH-SRQ, because the sender can always send without any limitations. Similarly we can see a 5% performance difference between MVAPICH-SR and MVAPICH-SRQ for SP.

Comparing the performance of MVAPICH-SRQ and MVAPICH-SR, we find that although MVAPICH-SR can also reduce memory usage compared with MVAPICH-RDMA, it leads to performance degradation, so MVAPICH-SRQ is a better solution.

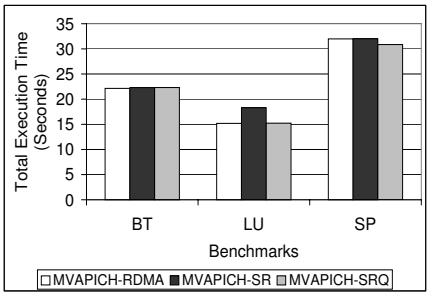


Figure 5.11: NAS Benchmarks Class A Total Execution Time (BT, LU, SP)

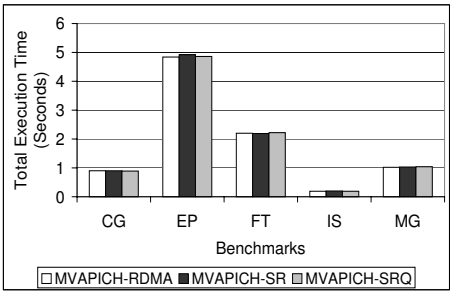


Figure 5.12: NAS Benchmarks Class A Total Execution Time (CG, EP, FT, IS, MG)

5.5.5 High Performance Linpack

In this section we carry out experiments using the standard High Performance Linpack (HPL) benchmark [2]. HPL stresses various components of a system including memory usage. For every system there is a limit for problem size based on the total amount of physical memory. The benchmark cannot run if the problem size goes beyond the limit. Figure 5.13 shows the performances of MVAPICH-RDMA, MVAPICH-SR, and MVAPICH-SRQ, in terms of GFlops.

From this graph we can see that MVAPICH-SR and MVAPICH-SRQ perform comparably with MVAPICH-RDMA for problem size from 10000 to 15000. For some problem sizes, such as 11000, 12000, and 13000, MVAPICH-SR and MVAPICH-SRQ perform even 10% better than MVAPICH-RDMA. This is because MVAPICH-RDMA needs to poll RDMA buffers of each connection when it makes communication progress. This polling wastes CPU cycles and pollutes cache content.

It is to be noted that for problem size 16000, the result for MVAPICH-RDMA is missing. This is because the memory usage of MVAPICH-RDMA itself is so large that the benchmark doesn't have enough memory to run. In other words, the problem size limit for MVAPICH-RDMA is around 15000. MVAPICH-SR and MVAPICH-SRQ, however, continue to give good performance as the problem size increases. Our system size is not large enough to show that MVAPICH-SRQ scales better than MVAPICH-SR. On a much larger cluster we will also be able to show that MVAPICH-SR has a smaller problem size limit than MVAPICH-SRQ.

5.6 Summary

In this chapter, we have proposed a novel Shared Receive Queue based Scalable MPI design. Our designs have been incorporated into MVAPICH which is a widely used MPI

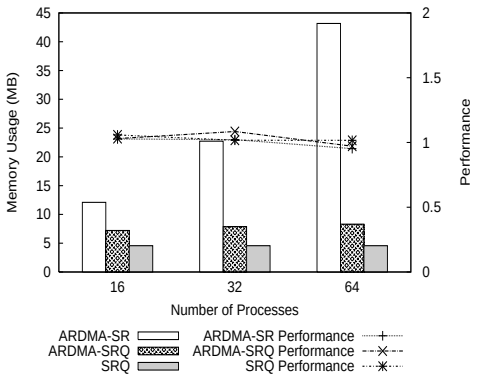


Figure 5.13: High Performance Linpack

library over InfiniBand. Our design uses low-watermark interrupts to achieve efficient flow control and utilizes the memory available to the fullest extent, thus dramatically improving the system scalability. In addition, we also proposed an analytical model to predict the memory requirement by the MPI library on very large clusters (to the tune of tens-of-thousands of nodes).

Verification of our analytical model reveals that our model is accurate within 1%. Based on this model, our proposed designs will take 1/10th the memory requirement as compared to the default MVAPICH distribution on a cluster sized at 16,000 nodes. Performance evaluation of our design shows that our new design was able to provide the same performance as the existing design utilizing only a fraction of the memory required by the existing design. In comparison to tuned existing designs our design showed a 20% and 5% improvement in execution time of NAS Benchmarks (Class A) LU and SP, respectively. The High Performance Linpack [2] was able to execute a much larger problem size using our new design, whereas the existing design ran out of memory.

CHAPTER 6

IN-DEPTH SCALABILITY ANALYSIS OF MPI DESIGN

MVAPICH provides various designs to perform message passing [26, 42]. Depending upon the requirement of the end MPI application and available InfiniBand hardware, different designs may be chosen by the user. In addition, all these designs are runtime tunable with various parameters. Most of these parameters are "hints" to the MPI library of the user's intentions. These parameters directly affect the performance, memory usage and other characteristics of the MPI library. Using these parameters, the MPI library allocates internal buffers that are used for communication. In addition, depending on the requirements of the application, more memory may be allocated during its actual execution. These communication buffers represent the majority of the memory consumption of the MPI library. Allocating more buffers may allow the library to offer better communication performance. On the other hand, lack of buffers may lead to runtime allocation and management of required memory (which is costly) and hence degradation of end application performance. Thus, the following two questions are of great significance to MPI library designers, cluster system vendors, and the end users:

1. Does aggressively reducing communication buffer memory lead to degradation of end application performance?

2. How much memory can we expect the MPI library to consume during execution of a typical application, while still providing the best available performance?

To the best of our knowledge, there has been no contemporary study that comprehensively answers these questions. In this Chapter, we provide answers to the above two questions by analyzing the internal MPI operations during execution of well known MPI applications and benchmarks such as NAS Parallel Benchmarks [6], SuperLU [46], NAMD [37], and HPL [2]. Our analysis reveals that for the NAS Benchmarks (Class B), NAMD, and HPL on 64 processes, the latest designs of MVAPICH require less than 5MB of internal memory on average per process and yet deliver the best available performance. For SuperLU, the memory usage increases to 10MB for the evaluated data sets, but still maintains optimal performance and a 5 times reduction in memory usage over older MVAPICH designs.

The rest of this chapter is organized as follows. We begin by providing an overview of the MPI design 6.1. Then we present the performance evaluation parameters and scope 6.2, followed by the performance results in Section 6.3. Finally, we summarize this chapter in Section 6.4.

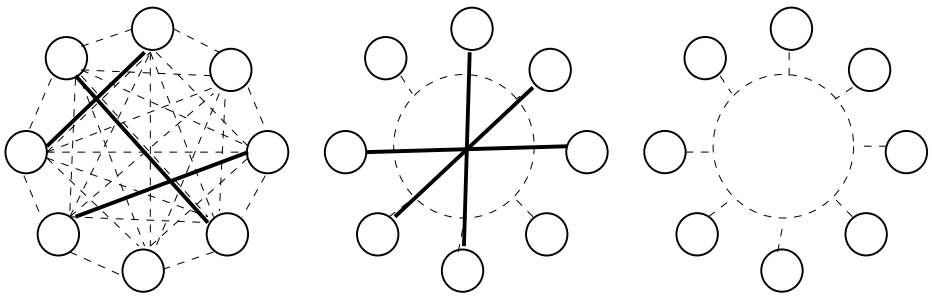
6.1 Overview of MPI Design

MVAPICH [34] is a popular implementation of MPI over InfiniBand. It uses several InfiniBand services like Send/Receive, RDMA-Write, RDMA-Read, and Shared Receive Queues to provide high-performance and scalability to end MPI applications. There are two major protocols used by MVAPICH. The first is the *Eager Protocol*, which is used to transfer small messages. The second protocol used is the *Rendezvous Protocol*, which is used for large messages. In order to avoid buffering large messages inside the MPI library, the Rendezvous protocol negotiates the availability of receive buffer by using control messages. After

the negotiation phase, the messages are sent directly to receiver user memory with the use of RDMA. These control messages used by the Rendezvous protocol are small in size and are sent over the Eager protocol. For more information on the design alternatives of the Rendezvous protocol, please refer to [43]. Thus, the Eager protocol can be used for MPI application generated small messages as well as Rendezvous control messages.

The Eager protocol requires the presence of "pre-allocated" communication buffers on both sender and receiver sides, in order to avoid any runtime costs and achieve low latency. The Rendezvous protocol does not require any additional buffer space other than the control messages sent over the Eager protocol. Hence, only the Eager protocol consumes communication memory in a MPI process. In this Chapter we focus on the requirement and usage of MPI internal buffers; hence, we will describe the Eager protocol in detail.

MVAPICH provides several implementations for the Eager protocol based on different designs and utilizing different InfiniBand features. In addition, these eager protocols can be used and combined to form hybrid protocols with dynamic thresholds. There are three basic protocols: a) based on per-connection Send/Receive model, b) based on RDMA-Write and c) based on Shared Receive queue. Combining two protocols at a time, there can be a total of six protocols, out of which we describe and evaluate three in this Chapter. We leave out three combinations: Send/Receive + Shared Receive Queue, since the use of shared resources implies attaching a Queue Pair to a shared queue instead of its per connection receive queue; RDMA-Write only protocol, since it is inherently unscalable due to the lack of flexibility to move communication buffers across connections, and; Send/Receive only protocol, since it is impossible to recall posted buffers to a particular connection, thus leading to inferior scalability. The remaining three protocol combinations are described below:



(a) Adaptive RDMA with Send/Receive Channel (with 6 processes using RDMA)

(b) Adaptive RDMA with SRQ Channel (with 6 processes using RDMA)

(c) SRQ Channel

Figure 6.1: Various Eager Protocol Designs in MVAPICH

6.1.1 Adaptive RDMA with Send/Receive Channel

The RDMA feature of InfiniBand offers very low latency due to the absence of receiver side software involvement, which is desirable for small messages. The RDMA channel [26] in MVAPICH provides a design by which the RDMA feature can be fully exploited to deliver low latency. The use of RDMA requires that communication buffers be made available for each remote process that may send messages. In order to avoid a memory-scalability problem when there are thousands of remote processes, this channel has an "adaptive" nature (hence the name Adaptive RDMA). RDMA channels are not created until after a threshold of messages (runtime tunable) have been exchanged over the Send/Receive channel. At the time of communication initialization, only a limited number (typically only two or three) of buffers are allocated per remote process. These initial buffers are posted on the InfiniBand Send/Receive channel. Accordingly, all processes initially communicate using the InfiniBand Send/Receive channel semantics. MVAPICH maintains an internal counter of the number of messages exchanged by each pair of processes, and if this count increases beyond some

threshold (runtime tunable), buffers are allocated and made available to the remote process over RDMA.

For the sake of brevity, this design will be referred to as ARDMA-SR for the rest of the Chapter and the connection between a pair of process that uses RDMA for Eager protocol will be called a RDMA Connection. Figure 6.1(a) illustrates this channel with the dotted lines showing the limited number of buffers for the Send/Receive channel. The bold lines indicate that six of the most frequently communicating processes actually communicate over RDMA. This channel provides reasonably good memory scalability along with the low latency offered by RDMA.

6.1.2 Adaptive RDMA with SRQ Channel

The Shared Receive Queue (SRQ) is a hardware feature provided by InfiniBand that allows upper-level software to post receive buffers to only one receive queue. Incoming messages from all remote processes in the MPI application can then consume buffers from this queue in a first-come-first-serve (FCFS) basis. This feature allows very efficient sharing of receive buffers across many InfiniBand connections. Thus, reducing the memory requirement by an order of magnitude for MPI applications that execute on very large process counts (up to tens of thousands).

One drawback of the SRQ is that the processes sending messages do not have an accurate picture of the receiver buffer availability. As such, if senders keep injecting packets into the network that do not have any destination buffer available, the performance of the application is degraded. In order to alleviate this situation, we have designed a novel, receiver-driven flow-control mechanism [42]. The receiving MPI process sets a "low-watermark" for the SRQ. When the number of available buffers in that queue drops below this threshold, an

interrupt is generated by the HCA, which is caught by the MPI library. If there are more receiver buffers allocated already, then they are posted to the HCA to keep the SRQ full; however, if no buffers are available, new ones are allocated and posted to the SRQ to fill it.

During communication initialization, all processes have full SRQs and communicate using these buffers. When a certain number of runtime tunable buffers have been consumed from the SRQ, RDMA buffers are made available for that remote process. Hence, similar to the design described in the previous section, this design also achieves scalable memory usage along with low latency of RDMA. Again, for the sake of brevity, the design will be referred to as ARDMA-SRQ for the rest of the Chapter.

6.1.3 SRQ Channel

This channel exclusively utilizes the SRQ feature of InfiniBand. It employs the same receiver-driven flow-control mechanism as described in the previous section. The only difference in this channel from the previous one is that no RDMA buffers are allocated, even for frequently communicating pairs of processes. Even though RDMA channels can achieve lower-latency message passing, they consume more memory. This channel, which is exclusively based on SRQ, may have slightly increased point-to-point latency (only by around 1μs), but can provide very scalable message passing. Figure 6.1(c) illustrates this channel. For the rest of the Chapter, this design will be simply referred to as SRQ.

6.2 Performance Evaluation Parameters

Much of the data required for our analysis are not obtainable through any other publicly available tools. This is mainly because we aim to analyze information that is specific and *internal* to MVAPICH. In addition to this, our analysis requires the size and volume information of the messages actually sent by the MPI library. Most MPI profiling tools

can provide information only about messages that were generated by the MPI application. As mentioned in previous Sections, large message transfer may in fact involve several small message transfers as required by the Rendezvous protocol. The information about these messages is lost if we simply use MPI-level profilers.

In order to obtain an accurate view of the various events occurring inside MVAPICH, we design an extremely low overhead profiling mechanism *internal* to MVAPICH. Our profiling implementation records information inside internal data structures of MVAPICH during the application execution. All the information is then collected at the root process by `MPI_Reduce` during `MPI_Finalize`. Since the profiler need only update a few memory locations during the execution, there is almost no perceivable impact on the performance; e.g. the 0-byte MPI message latency is unaffected, proving our hypothesis that our profiling introduces almost negligible overhead. Our profiling mechanism records important information such as:

1. Allocation of communication buffers
2. Message size and data volume profiles
3. Number of processes communicating over RDMA Eager Protocol
4. Number of "low-watermark" events experienced by the SRQ

In addition to our internal profiling of MVAPICH, we used `mpiP` [20], which is a lightweight, scalable MPI profiling tool. This tool provides us with information about which MPI calls were issued by the application. Combining this information (generated by `mpiP`) with our internal profiling of MVAPICH, provides an in-depth look into several aspects of the MVAPICH designs for the Eager protocol.

	IS	MG	CG	FT	LU	BT	SP	NAMD	HPL
Avg. RDMA Connections	6.14	9.0	3.09	0.98	3.92	3.89	1.17	53.15	6.26
Avg. Low-Watermark events	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.03	0.0
Unexpected Messages (%)	2.7	10.2	13.5	11.9	38.1	0.3	0.7	48.2	13.6
Total Messages	$1.9e5$	$3.1e5$	$2.7e6$	$3.6e5$	$5.8e6$	$1.6e6$	$4.7e6$	$3.7e6$	$7.8e5$
MPI Time (%)	47.25	9.16	33.87	37.85	14.23	10.17	11.88	23.54	24.68

Table 6.1: Profiling Results on 64 processes of NAS (Class B), NAMD (apoa1) and HPL

6.3 Performance Results

In this section, we present our analysis of the performance and the memory utilization of the MPI library while executing several well-known MPI applications and benchmarks. The Eager protocol designs evaluated are the Adaptive RDMA with Send/Receive (called ARDMA-SR), Adaptive RDMA with SRQ (called ARDMA-SRQ), and the SRQ channel (called SRQ).

Our experimental platform is a 64 node dual Opteron 2.4GHz (Processor 250) cluster. Each node is equipped with 8GB of main memory and PCI-Express interface. The nodes have MT25204 Mellanox HCAs with firmware version 1.0.1 and the OpenFabrics software stack. The Linux kernel version used is 2.6.15.

Table 6.1 shows the results of our profiling various applications on 64 processes. SuperLU profiling results are presented separately in Table 6.2. The percentage MPI time is reported by `mpiP` and the rest of the parameters are given by the MVAPICH internal profiling. This table will be referred to later as part of our analysis of the results of each individual benchmark.

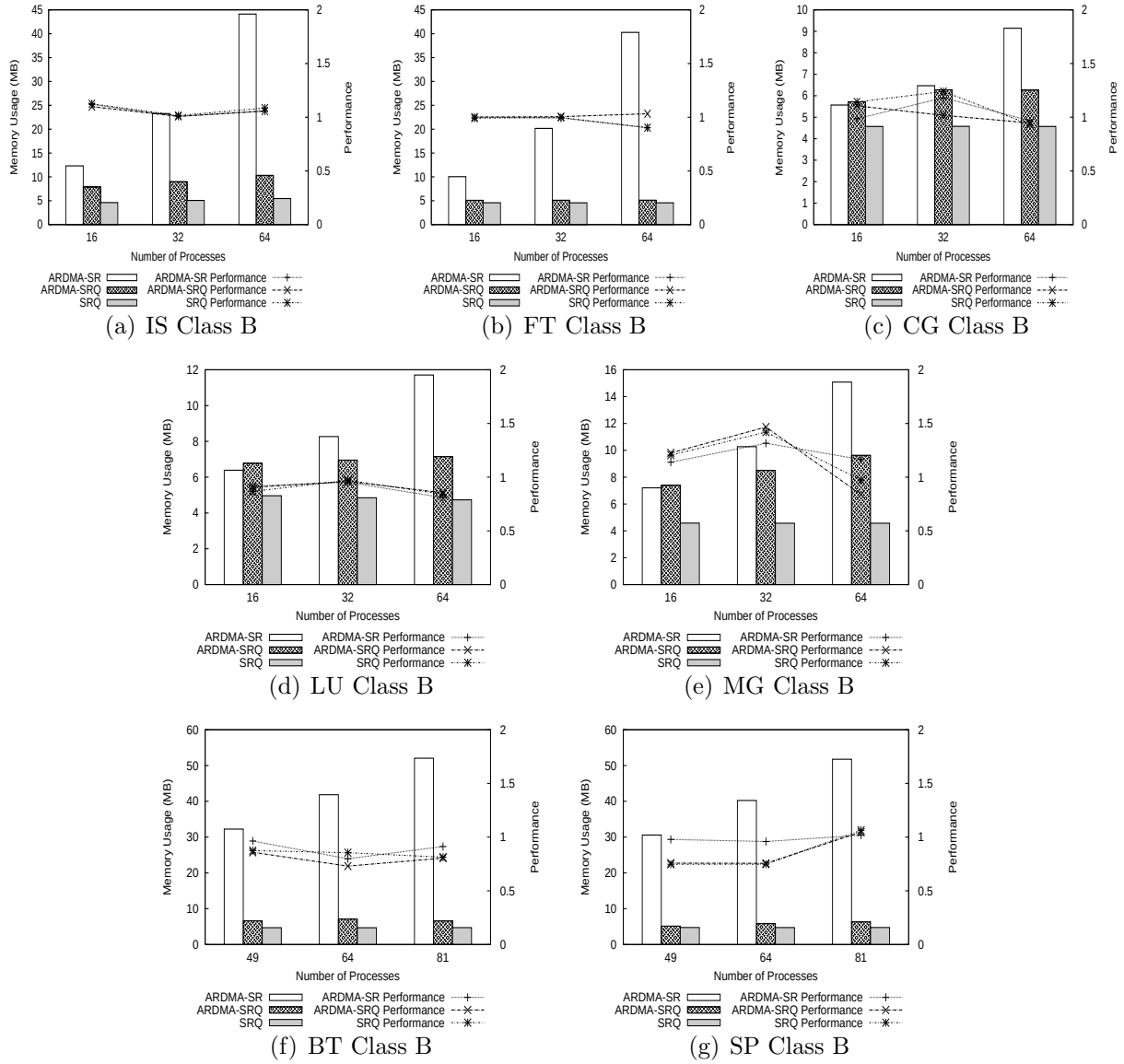
6.3.1 NAS Benchmarks

The NAS Parallel Benchmarks [6] are a set of programs that are designed to be typical of several MPI applications, and thus, help in evaluating the performance of parallel machines. For the purposes of our evaluation, we include all the NAS Benchmarks except the Embarrassingly Parallel (EP) benchmark. We excluded this benchmark from our evaluation, since it has very little MPI communication and as such is of lesser significance when analyzing the operations inside the MPI library.

Figure 6.2 shows the performance of the NAS Benchmarks (Class B) using all three designs of the Eager protocol. The number of processes is varied from 16 to 64 for IS, FT, CG, LU, and MG. The SP and BT benchmarks are run on 49 to 81 processes since they require the total number of processes to be a square. Each graph has two y-axes. The left y-axis shows the communication memory used by MVAPICH while executing that particular benchmark, whereas the right y-axis shows the relative performance achieved by that benchmark execution. All the performance ratios have been normalized with respect to the best possible benchmark number obtained by the default configuration of MVAPICH version 0.9.7. A ratio > 1 indicates better performance than the default configuration of MVAPICH 0.9.7, while a ratio < 1 indicates worse performance.

The results indicate that the SRQ channel is able to provide almost the same level of performance as the other two schemes: ARDMA-SR and ARDMA-SRQ. While the SRQ channel provides almost the same performance, it does so with markedly less communication memory. In fact, in all the Figures 6.2(a) through 6.2(g), the SRQ channel consumes less than 5MB of communication buffers.

Memory utilization numbers for benchmarks IS, FT, BT, and SP are shown in Figures 6.2(a), 6.2(b), 6.2(f), and 6.2(g), respectively. These show an order of magnitude

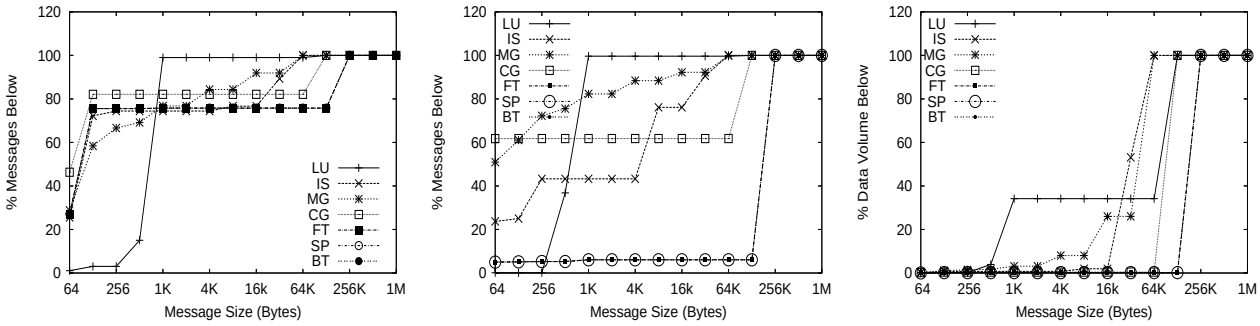


(a) IS Class B (b) FT Class B (c) CG Class B

(d) LU Class B (e) MG Class B

(f) BT Class B (g) SP Class B

Figure 6.2: Performance of NAS Benchmarks



(a) Percentage messages below a certain message size

(b) Percentage Unexpected messages below a certain message size

(c) Percentage of Data Volume below a certain message size

Figure 6.3: Network-Level Message and Volume Profile of NAS Benchmarks

improvement (around 10 times for 64 and 81 process executions) in memory usage when ARDMA-SR is compared with ARDMA-SRQ or SRQ. However, Table 6.1 shows that the average number of RDMA connections (Section 6.1.1) is in fact not that high. To answer this apparent contradiction, we examine the message and volume profile graphs in Figures 6.3(a) and 6.3(c). By looking at these graphs, we can make out that these benchmarks do the major part of their communications using very large messages. As explained in previous Sections, every large message transfer is associated with several smaller messages. These smaller messages are never sent over RDMA, rather exclusively use the Send/Receive channel. In order to transfer these small messages, an increasing number of communication buffers are allocated for the Send/Receive channel. Once the number of messages over the Send/Receive channel exceeds a certain amount, a much larger communication buffer set (64 in number) is required to be allocated *per* remote process for the Send/Receive channel. This consumes the most memory and exposes an inherent scalability issue even while using an adaptive protocol. The other NAS Benchmarks LU, MG, and CG show an improvement in memory usage as well as seen in Figures 6.2(d), 6.2(e), and 6.2(c). The SRQ channel

consumes around half the memory required by ARDMA-SR. The difference in memory usage between ARDMA-SRQ and SRQ can be explained by the number of processes using RDMA. For example, in the LU benchmark (for 64 processes), there are on an average 3.92 RDMA connections. According to default MVAPICH 0.9.7 parameters, each RDMA connection utilizes around 500KB of memory, so analytically, the difference in memory usage between ARDMA-SRQ and SRQ should be $(500 * 3.92)/1024$ MB = 1.9 MB. From Figure 6.2(d), we can observe that the memory usage difference is indeed around 2MB for 64 processes.

SuperLU is a general purpose library for the solution of large systems of linear equations on high performance machines [46]. SuperLU is offered in three different versions: sequential, multi-threaded (for shared memory machines), and an MPI version to be used on distributed memory machines. We used the MPI version, called SuperLU_DIST [23] that contains a set of subroutines to solve a sparse linear system $A * X = B$. Currently, the program SuperLU_DIST parallelizes the LU factorization and triangular solution routines, which are the most time consuming.

The communication characteristics of SuperLU have been studied previously by Shalf, et al [40]. It has a variety of MPI calls which are predominantly `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`, `MPI_Bcast`, and `MPI_Alltoall`. There are various data sets available for SuperLU_DIST. In our experiments, we have used `garon2.rua` and `rim.rua` from [12].

As seen in Figure 6.5(a), 94.99% of messages are less than 2KB for the `garon2` data set and 94.33% for the `rim` data set. While most messages are of small size, Figure 6.5(c) shows a few large messages that comprise most of the data volume.

Figures 6.4(a) and 6.4(b) show the performance and memory usage observed from our internal library profiling. As in the case of the NAS Benchmarks, the results indicate the ability of the SRQ channel to provide near-identical performance with significantly lower

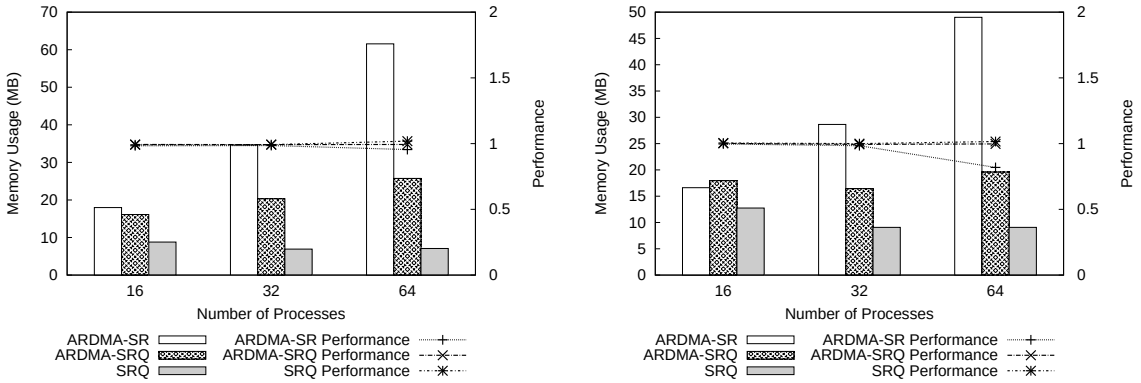
	garon2			rim		
Processes	16	32	64	16	32	64
Avg. RDMA Connections	12.44	25.75	40.25	7.25	12.06	14.25
Avg. Low-Watermark events	1.56	0.06	0.12	1.56	0.66	0.64
Unexpected Messages (%)	33.5	22.0	31.6	29.4	24.2	30.0
Total Messages	2.9e5	4.8e5	7.5e5	3.8e5	7.4e5	1.1e6

Table 6.2: Profiling Results for SuperLU

allocation of communication buffers. In the case of the `garon2` data set, usage remains roughly constant between the range of 7 to 9MB. Interestingly, with both data sets the memory usage for the SRQ design per process is higher for 16 processes than 32 or 64. From Table 6.2, we observe that using 16 processes, the average number of "low-watermark" events (when SRQ buffers are low) is approximately 1.5, while 32 and 64 processes have significantly lower values.

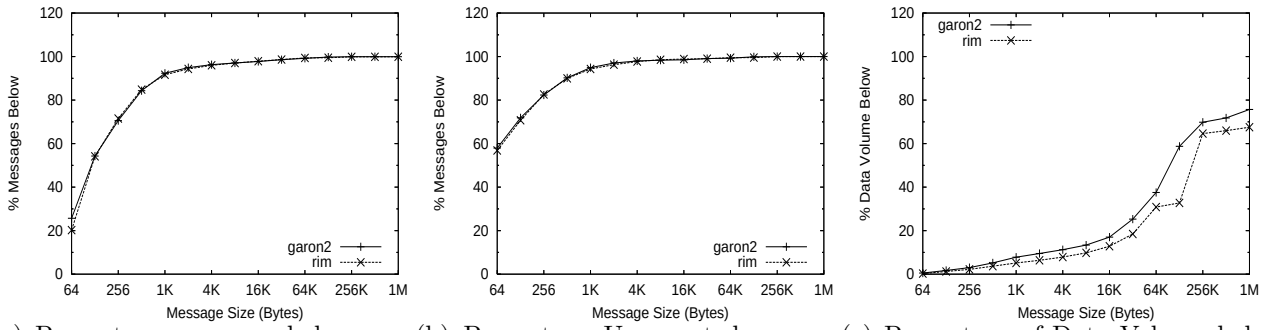
This result suggests a communication pattern with significant bursts of unexpected messages and additionally that these bursts occur less frequently with a larger number of processes. These significant traffic bursts wake a thread to post additional buffers to the shared received queue, increasing the overall memory usage.

The benefits of the SRQ Eager protocol design are most prominent at a process group size of 64. We observe from Figures 6.4(a) and 6.4(b) that the communication buffer memory usage for `garon2` is nearly an order of magnitude less than the ARDMA design, yet maintains the same level of performance. The SRQ results for the `rim` data set yield similar results, with a 9 and 4 times improvement over the ARDMA-SR and ARDMA-SRQ designs, respectively, with near-identical performance. Most importantly, our evaluation shows a near-constant memory usage per process, regardless of the process group size.



(a) garon2 (b) rim

Figure 6.4: Memory Usage and Performance of SuperLU



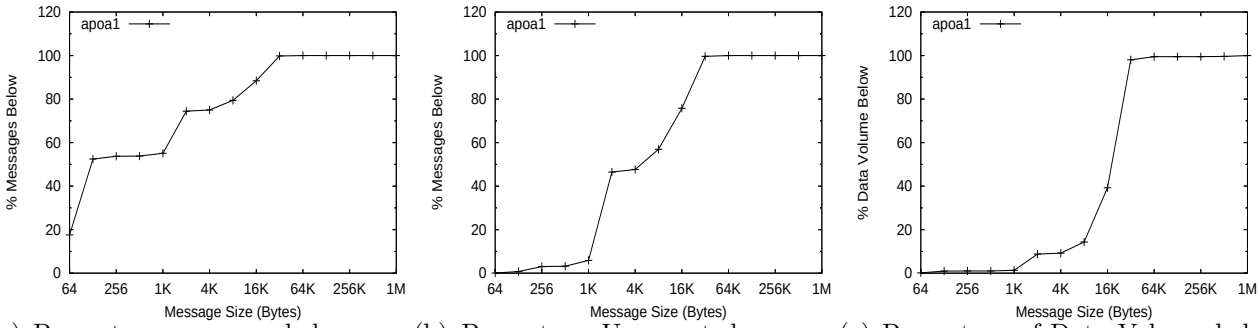
(a) Percentage messages below a certain message size (b) Percentage Unexpected messages below a certain message size (c) Percentage of Data Volume below a certain message size

Figure 6.5: Network-Level Message and Volume Profile of SuperLU Datasets

6.3.2 NAMD

NAMD is a fully featured, production molecular dynamics program for high performance simulation of large biomolecular systems [37]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. Of the various data sets available with NAMD, we use the one called `apoa1`, which models a bloodstream lipoprotein particle.

The communication characteristics, as reported by `mpiP`, show the calls are primarily to `MPI_Isend`, `MPI_Send`, `MPI_Recv`, and `MPI_Barrier`. Our profile of the messages sent by the MPI library show 50% are under 128 bytes and the remaining 50% are between 128 and 32K bytes.



(a) Percentage messages below a certain message size

(b) Percentage Unexpected messages below a certain message size

(c) Percentage of Data Volume below a certain message size

Figure 6.6: Network-Level Message and Volume Profile of NAMD Datasets

In Figure 6.7 we observe the same trends in performance and memory usage as in previous applications. For a process group size of 16 the SRQ design uses on average 6.1MB of memory and drops to 5.5MB and 5.2MB for the 32 and 64 process groups. As in SuperLU, we see the

ability of the SRQ Eager protocol design to consume less memory with larger process groups due to a more balanced application communication pattern between all nodes. However, even with patterns with short bursts of unexpected traffic, such as the 16 process run, we observe a 50% improvement in memory usage over both of the ARDMA designs.

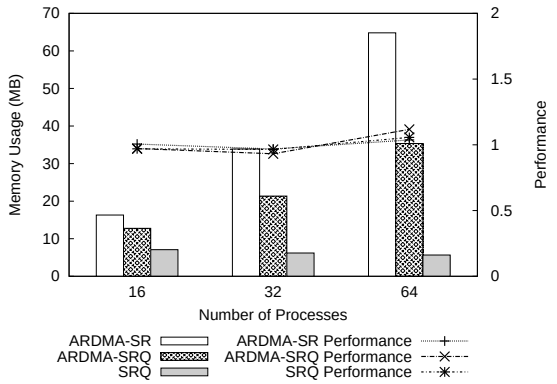


Figure 6.7: Performance of NAMD (apoa1)

In contrast, the communication buffer usage in the ARDMA-SR design scales linearly with the number of processes. Table 6.1 shows one of the reasons for this scale. The number of RDMA connections also scales linearly with the number of processes due to a balanced communication pattern. This pattern triggers the creation of an RDMA channel after communicating a set number of messages, as discussed in Section 6.1.1. For 64 processes, our evaluation shows an average of 53.15 RDMA connections. The ARDMA-SRQ design also shows a significant increase over the SRQ design in memory usage due to RDMA channels. The difference in memory usage between the SRQ and ARDMA-SRQ designs is 28MB, which matches our previous model of the RDMA channel overhead: (RDMA Connections $\times$ 500KB) = 53.15 Connections $\times$500KB = 26.6MB.

6.3.3 High Performance Linpack (HPL)

High Performance Linpack (HPL) is benchmark based on solving systems of linear equations [2]. It is used as the primary measure for ranking a bi-annual Top 500 list [45] of the world's fastest supercomputers.

The communication pattern, as recorded by `mpiP`, shows the calls are primarily to `MPI_Recv`, `MPI_Send`, and `MPI_Irecv`. Figure 6.8 shows the performance and communication buffer memory usage observed for 16, 32, and 64 process runs of HPL. We once again see a relatively constant rate of performance for all of the Eager design schemes. The SRQ channel, however, is able to use a constant communication buffer size of less than 5MB for all evaluated process sizes. Figure 6.9 shows the results of our profiling of the messages sent by the MPI library. We observe that while 50% of the messages are under 128 bytes, most of these are control messages for the larger application-level messages.

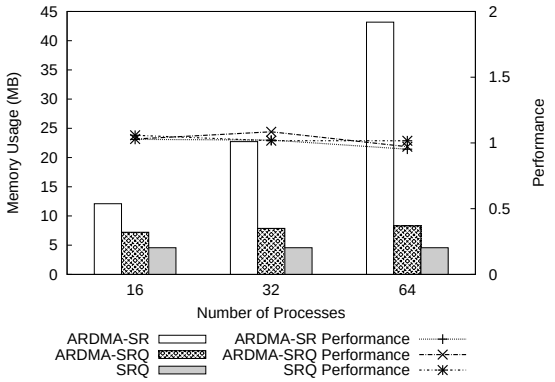


Figure 6.8: Performance of HPL

Referring to Table 6.1 we can see that for 64 processes, on average, only 6.26 RDMA connections are established. This result explains the approximately 3.5MB difference between the ARDMA-SRQ and SRQ designs; our model relating to RDMA channel memory requirements from other sections holds here as well. There is also a marked increase in the memory usage between the ARDMA-SRQ and ARDMA-SR designs of nearly 35MB for 64 processes. Although Figure 6.9 shows that many messages sent are of medium size, there are also a significant number of larger messages. As discussed earlier, even large messages require smaller control messages to be sent over the Send/Receive channel. When many of these smaller messages are transferred, an increasing number of communication buffers must be allocated on a per connection basis in the ARDMA-SR design, raising the memory usage of the MPI library.

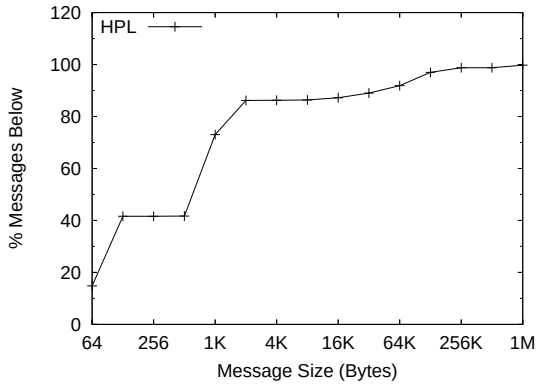


Figure 6.9: Message Size Distribution for HPL

6.3.4 Scalability Analysis

In this section, we combine some of the results obtained by the evaluation of the various applications and benchmarks in order to observe the scalability of the SRQ channel.

We observe from Tables 6.1 and 6.2, that only NAMD and SLU applications have Low-Watermark events. These events are caused when the SRQ channel is running low on available receive buffers. After each Low-Watermark event occurs, previously unused receive buffers can be made available to the network, or more receive buffers may be allocated if required. This is expected, since both SLU and NAMD have a predominantly small messages which end up utilizing communication buffers. Figure 6.10 shows the number of Low-Watermark events for both these applications as the number of processes increases.

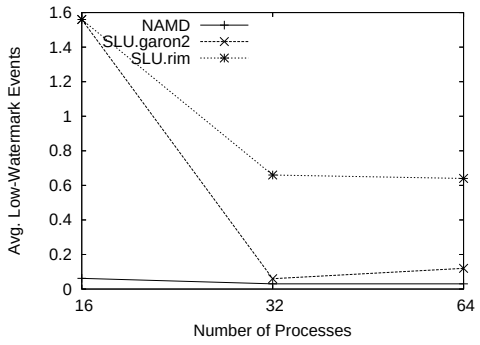


Figure 6.10: Avg. Low-Watermark Events

The results indicate an interesting trend – that the average number of interrupts actually decreases as the system size increases. This implies, that given these application characteristics, as the system size increases, it is expected that no more dynamically allocated communication memory is required. This trend also explains why addition of more buffers, as in the case of ARDMA-SR and ARDMA-SRQ does not lead to any "extra" improvement in application performance. This is because the amount of communication memory allocated at startup, is almost sufficient for the entire application run and the SRQ channel is able

to effectively utilize them. Thus, the SRQ channel is expected to achieve a high degree of memory-scalability while providing excellent performance on even larger system sizes.

6.4 Summary

As InfiniBand gains popularity and is included in increasingly larger clusters, having a scalable MPI library is imperative. Through our evaluation of the NAS Parallel Benchmarks, SuperLU, NAMD, and HPL, we have explored the impact of reduction of communication memory on the performance. We have shown that all of the schemes in MVAPICH are able to attain near-identical performance on a variety of applications. Our evaluation showed that the latest SRQ design of MVAPICH is able to use a constant amount of internal memory per process with optimal performance, regardless of the number of processes, an order of magnitude lesser than other Eager protocol designs of MVAPICH. In our experiments, only 5-10MB of communication memory was required by the SRQ design to attain the best recorded performance level achievable with MVAPICH.

CHAPTER 7

OPTIMIZING MPI APPLICATIONS: A CASE STUDY WITH TWO HPCC BENCHMARKS

The field of parallel computing is rapidly evolving with many different types of parallel architectures emerging. Evaluation of these architectures is a very big challenge for the entire High-Performance Computing (HPC) community. Without a thorough evaluation, the community cannot decide which architectural changes are worthwhile and which are not desirable. In order to answer this challenge, DARPA [5] has constituted a HPC Challenge program with the competition based on several benchmarks. These benchmarks are available from the Innovative Computing Labs at University of Tennessee in the form of the HPCC Benchmark Suite [18].

As we have seen in the previous Chapters, the MPI design parameters can have a significant impact on the performance characteristics of end applications. Moreover, some of the MPI optimizations leverage certain network strengths. However, in order to maximize application performance, leveraging modern network features, the design of the communication sections of MPI applications should be revisited. In this Chapter, we will demonstrate our optimizations to the communication section of two well known HPCC Benchmarks, namely, High-Performance Linpack (HPL) and RandomAccess. HPL is designed to expose

the maximum computation power of a distributed memory computer, whereas RandomAccess is designed to test the maximal memory and network latency. Since the benchmarks are well studied and very popular in the HPC community, modifying them to achieve better performance gives a strong use case to application developers who can then identify similar communication patterns in their applications and modify them accordingly to improve performance. Evaluation of our modifications reveal that performance of HPL can be improved by around 10% and performance of RandomAccess can be improved by 10x on our cluster with 512 processes.

The rest of the Chapter is organized as follows. In Section 7.1, we provide an overview of the HPCC benchmarks studied in this Chapter. In Section 7.2, we describe the relevant MPI optimizations from the previous chapters and their implications to end MPI application design. Then, in Section 7.3, we discuss in detail our modifications to the two benchmarks. In Section 7.4 we present the results of our performance evaluation and finally in Section 7.5 we conclude this Chapter.

7.1 Overview of HPCC Benchmarks

In this section, we provide an overview of the two HPCC benchmarks, HPL and RandomAccess. In particular, we are interested in the communication characteristics of these benchmarks.

7.1.1 Overview of HPL Benchmark

The HPL Benchmark [2] aims to measure the "peak" computational power of a distributed memory computer. The benchmark solves a linear system $Ax = B$, by performing LU decomposition. The matrices are distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to achieve good load balancing. In this work, we

will focus on the communication pattern exhibited by HPL. At every iteration, a "panel" is factorized by every process in a column and then broadcasted along the row to other panels. This broadcast is not done using `MPI_Bcast`. In `MPI_Bcast`, all the processes (with exception of the root) need to wait for incoming message before proceeding. HPL tries to optimize the broadcast, by doing it in a non-blocking fashion. In every step, a process along the row sends a message. The next process in the row proceeds with computation as normal and occasionally calls `HPL_bcast`. `HPL_bcast` looks for incoming messages and if the message has arrived forwards it along the row to the next process. An example of this process is seen in Figure 7.1. There are several variation of this algorithm where the root may send two messages, divide the message, and so on. However, all the algorithms have the same basic structure, non-blocking poll for incoming messages and forward message on to next process in the row. In all the variations of the algorithm the process sending the message uses a blocking `MPI_Send` call, whereas the receiving processes use `MPI_Iprobe` to look for incoming messages. When the message has arrived, they call `MPI_Recv` to receive the message. Depending on whether they are the last process or not, they forward it again using `MPI_Send`. Thus, HPL implements overlap of computation and communication on the receiver side, but not on the sender side (where it uses blocking MPI calls).

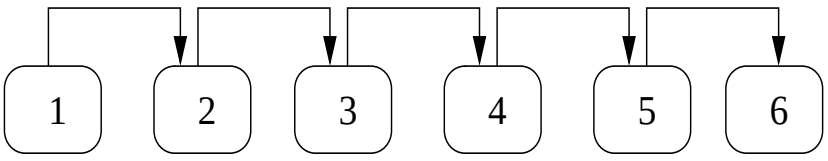


Figure 7.1: HPL Broadcast Algorithm

7.1.2 Overview of RandomAccess Benchmark

The RandomAccess benchmark [18] is an attempt to categorize system performance based on the rate at which random memory locations can be updated. The benchmark is motivated from the point of view of improving overall system performance, where the pattern of memory accesses is unpredictable. This characteristic is believed to be representative of end application performance and development time. It is also an attempt to capture the growing gap between CPU performance, which has been exponentially increasing as per Moore's Law [14], and the slower rate of growth in memory speeds; commonly this phenomenon is known as the "Memory Wall".

The RandomAccess benchmark requires only one parameter, n, such that, $n \leq \log_2(TotalMem/2)$. Where, $TotalMem$ is the total memory available in the system in bytes. The benchmark, then allocates a table T of size 2^n, which is distributed uniformly across the entire system. The benchmark then generates a random sequence of 64-bit integers. These numbers are generated using a primitive polynomial $x^{63} + x^2 + x + 1$ over GF(2). GF(2) is simply the binary digits, 0 and 1. For each random number $a[i]$, the lower $(N-1)$ bits, where N is the total number of processes, is used to index into T, say $T[j]$. Then the value at $T[j]$ is updated to $T[j] = T[j]$ XOR $a[i]$. In the case $T[j]$ belongs to a remote process R, the random number $a[i]$ is sent to R and then R applies the update. Thus, random update to a random memory location is performed.

The benchmark rules permit buffering up to 1024 updates. This "look-ahead" is permitted to enable some amount of pipelining. Each update is inserted into a "bucket" before being sent out. Once the limit of 1024 updates is reached, the bucket containing the largest number of updates is sent out, followed by next largest, and so on. When a process receives a packet, it applies all the updates present in the packet. The entire series of updates is

generated and applied by all the processes, and finally a *GUPS* (Giga Updates Per Second) metric is calculated. This is the final score which is reported by the benchmark.

Since the RandomAccess benchmark sends messages to almost all the remote processes and the messages are very small, the performance of this benchmark might be highly dependent upon the implementation of the *Eager Protocol* and communication buffer management in MPI libraries. In particular, the more short messages a network and MPI library can handle at a given point, the better the performance on this benchmark.

7.2 Overview of MPI Library Optimizations

In this section, we discuss the relevant MPI optimizations for this work. Based on these optimizations, the HPCC benchmarks may be modified to take the maximum advantage.

7.2.1 Optimizations for Computation/Communication Overlap

As we discussed in Chapter 2, we have redesigned the MPI Rendezvous Protocol to take better advantage of the InfiniBand RDMA Read capability. Using this new protocol, we can achieve nearly complete computation and communication overlap. But in order to leverage this capability, the MPI application needs to make use non-blocking MPI communication calls, namely `MPI_Isend` and `MPI_Irecv`. If the MPI application uses blocking calls, then by the very semantics there is no potential for computation and communication overlap.

7.2.2 Optimizations to Communication Buffer Management

In Chapter 5 we discussed our new design of communication buffer management technique using InfiniBand Shared Receive Queues. Using this mechanism, there is no need for dedicating per process buffers to store in-coming messages. Instead, a shared queue is provided which is directly used by the hardware to place messages in FIFO order. In addition to

the change in the buffering mechanism, the *flow control* method also has been redesigned. In the new method, the flow control is invoked in a *reactive* manner. There is no pre-set limit of how many messages a process can send to its remote partners. Rather, based on how many messages are arriving, the receiver may decide to allocate more buffers. If the receiver has no more buffers, the InfiniBand Hardware will back off gracefully and not flood the receiver with more messages. This *reactive* method contrasted with the *proactive* method that is usually employed by the per process based buffer management techniques (i.e. those using InfiniBand Send/Receive channel, not Shared Receive Queue). In the proactive method, if there are no dedicated buffers at the remote end, then even small messages fall back to the Rendezvous Protocol, which makes progress only when the receiving application posts a receive. This ruins computation and communication overlap and also inserts several more steps during the communication process.

7.3 Modifications to HPCC Benchmarks

In this Section, we describe the modifications to the HPCC benchmarks. These modifications are made specifically to leverage the MPI library optimizations mentioned in the previous Section.

7.3.1 Modifications to HPL

The HPL benchmark typically uses large messages in the `HPL_bcast` communication function. In order to hide network latency, the benchmark is pipelined. "Panels" are factorized and they need to be broadcasted along the row of the P-by-Q grid. The benchmark starts the broadcast and continues to factorize the *next* panel. By calling `HPL_bcast` in between computations, progress is achieved on the discovery and forwarding of messages. However, as mentioned in Section 7.1.1, the benchmark uses `MPI_Send`, which is a blocking call. Thus,

even though, the underlying Rendezvous protocol is based on Read semantics, the sender is effectively blocked waiting for the receiver to arrive and accept the message.

Our modifications are to the `HPL_bcast` function which replaces the blocking `MPI_Send` call with `MPI_Isend`. With our modifications, every time the sending process calls `HPL_bcast`, a corresponding `MPI_Test` is issued which checks to see if the message has been sent or not. If the message send is complete, it returns `HPL_SUCCESS`, otherwise it returns `HPL_KEEP_TESTING`. As long as the result is `HPL_KEEP_TESTING`, the benchmark returns to computation and calls `HPL_bcast` after periodic intervals. Using this method, we can effectively leverage the RDMA Read semantics. The sending process can continue computing, as the network-interface on the remote side takes the responsibility of transferring the message. Thus, using this modification to HPL, we aim to improve the computation and communication overlap ratio.

7.3.2 Modifications to RandomAccess

The RandomAccess benchmark generates 64-bit updates for a table that is distributed across all the processes. Based on where the updates are applicable, processes send messages with the updates and receiving processes apply them. The updates are generated randomly and are expected to be distributed all over the table. The benchmark allows up to 1024 updates to be buffered. As the system size increases, however, the 1024 updates are scattered more and more throughout the cluster. This leads to making this benchmark being very sensitive regarding small message latency and concurrency offered by the network.

In the default implementation of the benchmark, there are very conservative assumptions regarding the concurrency offered by the MPI library and the network. The benchmark executes the following in a loop. Look for one incoming message, generate an update and

buffer, if all 1024 updates are generated – send one message. Profiling this benchmark reveals that a huge amount of time around 60% is spent just looking for incoming messages.

Our modifications to the benchmark allow for it to generate the full 1024 updates, disperse the updates and then start looking for messages. In this manner, we intend to make full use of the concurrency of the network. We have use `MPI_Isend` to send the messages, used multiple `MPI_Irecv` operations to look for messages incoming from any source and with any tag. As long as the underlying network and library can support high rate of outgoing/incoming messages, our version of the benchmark should achieve much higher GUPS rates.

7.4 Performance Results

In this section we present the performance results with our modifications to the HPL and RandomAccess benchmarks. The system used for the performance evaluation is described as follows.

Experimental Platform

Our cluster consists of 64 nodes connected with Mellanox InfiniHost III DDR network-interfaces. A 144-port Qlogic DDR switch is used to connect all the nodes. Each node is a dual-quad-core with Intel Clovertown processors running at 2.0GHz. The nodes have Open-Fabrics version 1.2 InfiniBand access layers installed. For all the evaluations, MVAPICH-0.9.9 is used as the MPI library. This library has all the optimizations (described in Section 7.2). These optimizations can be turned on/off during runtime.

7.4.1 Performance Results for HPL

We perform two different types of experiments with HPL. In the first experiment, we increase the number of processes while adjusting the HPL problem size (N) to a value such

that it will occupy close to 80% of the memory available. In the second experiment, we keep the number of processes fixed at 512 and vary the problem size. For the purposes of this experiment, all process distribution is made in a cyclic manner.

Figure 7.2 shows the results of the first experiment. The "Default" legend indicates the GFlops obtained by running the default version of HPL with the default runtime options of MVAPICH. The "Optimized" legend indicates the modified version of HPL which is run with MVAPICH runtime option `VIADEV_RNDV_PROTOCOL=RGET`. This option turns on RDMA Read support in MVAPICH. We observe from the results that the modifications have a strong impact on overall performance as the number of processes increase. Especially, at 512 processes, we can see an improvement of around 10%.

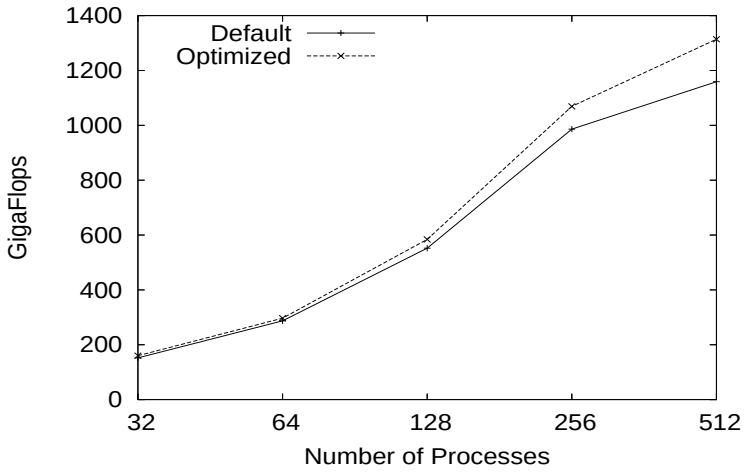


Figure 7.2: HPL Results with increasing number of processes

Figure 7.3 shows the results of the second experiment, conducted on 512 processes while increasing the problem size. The legend is as explained above. From these results, we observe

that the benefit offered by our benchmark modifications and MPI library optimization is nearly constant at around 10% as the problem size increases.

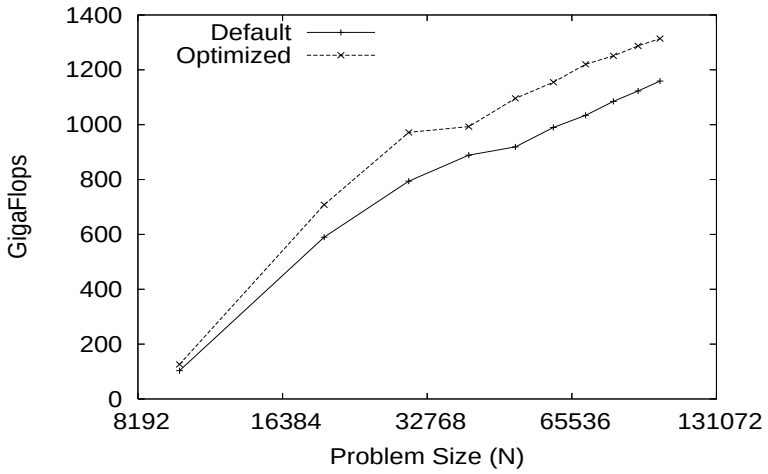


Figure 7.3: HPL Results on 512 processes with increasing problem size

7.4.2 Performance Results for RandomAccess

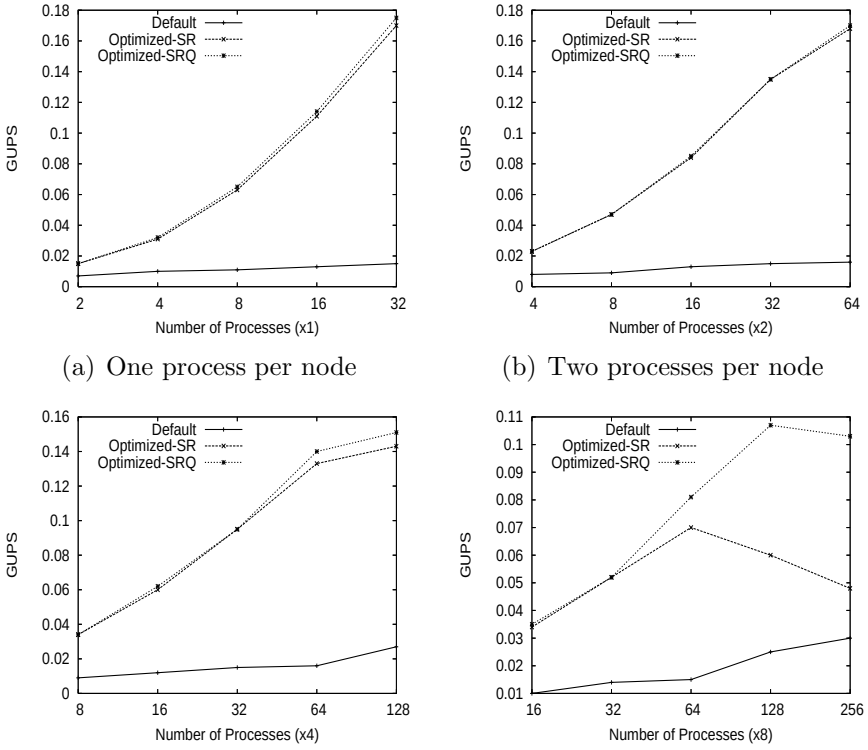
In this section, we provide the results of the RandomAccess optimizations. The experiments are strong scaling in nature. We choose 2^{26} as the default global table size for various number of processes. We have three legends in the following results. The "Default" legend indicates the standard RandomAccess benchmark run with default MVAPICH parameters. The "Optimized-SR" legend indicates the optimized version of the benchmark, run with the InfiniBand send/receive mode. Finally, the "Optimized-SRQ" indicates the optimized version of the benchmark, run with Shared Receive Queues.

Figure 7.4 shows the results obtained. We distribute the results by the number of processes used on each node. From Figures 7.4(a) to 7.4(d), we can observe that the GUPS

rate is significantly enhanced by our optimized version of the benchmark, up to 10x. This is a testament to the fact that modern networks can support very high concurrency and many outstanding small messages. More importantly, we observe that with the Shared Receive Queue mode, the performance is sustained at a higher level even when the number of processes increases up to 256, as per Figure 7.4(d). This is because, with Send/Receive, the number of credits, as per the point-to-point *proactive* flow control diminishes, all the small messages are in-turn sent over the Rendezvous Protocol. The version using Shared Receive Queues, however does not face this problem because of the *reactive* flow control methods. Communication buffers are used only as per the messages coming in and no reservation is made per remote process. Thus, even though the "Optimized-SR" provides much better performance than the "Default" run of the code, it does not perform as well as the "Optimized-SRQ".

7.5 Summary

In this Chapter, we demonstrated that by revisiting the design of end MPI applications, we can gain significant performance improvement. The communication patterns of these applications and benchmarks need to be studied and modified to take the most advantage out of modern networks and their capabilities. As we had seen in the previous Chapters, the MPI design parameters can have a significant impact on the performance characteristics of end applications. With the coupling of the application modifications with optimized MPI library design, we can improve overall performance significantly. In this Chapter, we modified two well known benchmarks, namely, High-Performance Linpack (HPL) and RandomAccess. Our modifications coupled with optimized runtime parameters could boost the performance



(a) One process per node

(b) Two processes per node

(c) Four processes per node

(d) Eight processes per node

Figure 7.4: Performance of RandomAccess Benchmark

of HPL by around 10% and the performance of RandomAccess by around 10x on our cluster using 512 processes.

CHAPTER 8

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

The work described in this dissertation has been incorporated into our MVAPICH software package and is distributed in an open-source manner. The duration of this work has spanned several release versions of this package, from version 0.9.2 to 0.9.9 (current). Some additional enhancements may also be included in upcoming version 1.0.

MVAPICH supports many software interfaces, OpenFabrics [35], uDAPL [11], and VAPI [28]. Most of the work described in this dissertation is geared towards the OpenFabrics interface. MVAPICH also supports 10GigE networks through iWARP support which is integrated in the OpenFabrics software package. In addition, any network which implements the network independent uDAPL interface may make use of MVAPICH. Further, MVAPICH supports a wide variety of target architectures, like IA32, EM64T, X86_64 and IA64.

Since its release in 2002, more than 520 computing sites and organizations have downloaded this software. In addition, nearly every InfiniBand vendor and the Open Source OpenFabrics stack includes this software in their packages. Our software has been used on some of the most powerful computers, as ranked by Top500 [45]. Examples from the June 2007 rankings include $15th$, 5848-core Dell PowerEdge (Intel EM64T) cluster at Texas Advanced Computing Center/Univ. of Texas (TACC), $19th$, 9216-core Appro Quad Opteron

dual Core at Lawrence Livermore National Laboratory and $71st$, 2200-processors Apple Xserve 2.3 GHz cluster at Virginia Tech.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The research in this dissertation has demonstrated the feasibility of scaling MPI applications successfully to very large InfiniBand clusters with the aid of employing scalable techniques inside the MPI library. We have described how we can take advantage of InfiniBand features such as Shared Receive Queues, RDMA with Gather/Scatter capabilities and Selective Interrupts in order to design a high-performance MPI library. Our work has involved designing new MPI protocols, Collective communication mechanisms, Communication buffer management techniques, Flow control, and lightweight communication profiling layers.

9.1 Summary of Research Contributions

The research in this dissertation aims towards designing highly scalable and high-performance MPI over InfiniBand. Several of the best designs from the related publications are already a part of the MVAPICH software package. MVAPICH is very widely used, including the largest InfiniBand clusters to-date: Sandia Thunderbird at 8960 processors [39], 9216-core LLNL Peloton, 5848-core TACC Lonestar cluster and 2200 processor Apple X-serve cluster at Virginia Tech. The work done in this dissertation will enable applications to execute at even larger scales and achieve the best performance.

We note that several of the ideas developed in this dissertation are in-fact are applicable to other high-performance middleware such as parallel file systems [3], other parallel programming models [4, 1]. Thus, we foresee that the contribution of this proposal will be significant for the HPC community. Following is a more detailed summary of the research presented in this dissertation.

9.1.1 Improving Computation/Communication Overlap

In Chapter 2, we have presented new designs which exploit the RDMA Read and the capability of generating selective interrupts to implement a high-performance Rendezvous Protocol. We evaluated in detail the performance improvement offered by the new design in several different areas of high performance computing. We observed that the new designs can achieve nearly complete computation and communication overlap. The results indicate that our designs have a strong positive impact on scalability of parallel applications.

9.1.2 Improving Performance of Collective Operations

In Chapters 3 and 4, we presented new designs to take advantage of the advanced features offered by InfiniBand in order to achieve scalable and efficient implementation of the `MPI_Alltoall` and `MPI_Allgather` collectives. We proposed that the implementation of collectives be done directly on the InfiniBand Verbs Interface rather than using MPI level point-to-point functions. We evaluated in detail why collective operations may not be optimally designed over simple MPI Send/Receive calls. Our experimental results and analytical models enable us to conclude that our new designs are more scalable and efficient than current approaches.

9.1.3 Scalable Communication Buffer Management Techniques

In Chapter 5, we proposed a novel Shared Receive Queue based Scalable MPI design. Our designs have been incorporated into MVAPICH which is a widely used MPI library over InfiniBand. Our design uses low-watermark interrupts to achieve efficient flow control and utilizes the memory available to the fullest extent, thus dramatically improving the system scalability. In addition, we also proposed an analytical model to predict the memory requirement by the MPI library on very large clusters (to the tune of tens-of-thousands of nodes).

9.1.4 In-Depth Scalability Analysis of MPI Design

As InfiniBand gains popularity and is included in increasingly larger clusters, having a scalable MPI library is imperative. Through our evaluation of the NAS Parallel Benchmarks, SuperLU, NAMD, and HPL in Chapter 6, we have explored the impact of reduction of communication memory on the performance. We have shown that all of the schemes in MVAPICH are able to attain near-identical performance on a variety of applications. Our evaluation showed that the latest SRQ design of MVAPICH is able to use a constant amount of internal memory per process with optimal performance, regardless of the number of processes, an order of magnitude lesser than other Eager protocol designs of MVAPICH. In our experiments, only 5-10MB of communication memory was required by the SRQ design to attain the best recorded performance level achievable with MVAPICH.

9.1.5 Optimizing end MPI Applications/Benchmarks

In Chapter 7, we demonstrated that by revisiting the design of end MPI applications, we can gain significant performance improvement. The communication patterns of these applications and benchmarks need to be studied and modified to take the most advantage

out of modern networks and their capabilities. The MPI design parameters can have a significant impact on the performance characteristics of end applications. With the coupling of the application modifications with optimized MPI library design, we can improve overall performance significantly.

9.2 Future Research Directions

The high-performance and rich features offered by InfiniBand make it a very attractive interconnect for large scale system design. In this dissertation, we have shown the methods one can employ to design a highly scalable MPI communication library over InfiniBand. However, there are several interesting research topics that are still left to be explored.

Next-Generation Programming Models – One of the challenges to PetaScale computing is programmer productivity. It has been a widely held view that although MPI allows development of very scalable and high-performance applications, it requires a huge amount of effort from the part of the programmer. In this regard, DARPA HPCS [10] has initiated a challenge to come up with next generation programming models which allow much more easier programming from application developers. Examples of these are X10 [16], Fortress [41] and Chapel [9]. While boosting programmer productivity, close attention should be paid to performance. Techniques developed in this dissertation could find applications in the runtime environments of these new languages. In addition, newer techniques may be designed according to the specific features offered by these languages.

Reliable Datagram – The Reliable connection mode has so far scaled to tens-of-thousands of nodes. However, the technique is expected to hit a bottleneck when ultra-scale clusters are being designed. The very nature of reserving resources (by establishing connections) before actual communication takes place, is not extremely scalable. This has

spurred research in using the Unreliable Datagram in MPI libraries [21, 22]. The results are very encouraging, and indicate that using datagrams, not only can MPI libraries scale to hundreds-of-thousands of nodes, but also the performance penalty can be reduced. While this is encouraging, using unreliable mode of communication places an onerous task on the programmers of communication libraries. It is a huge challenge to add reliability to all communication libraries. Also, relying on reliability at the host layer can add several more issues, like making progress on dropped packets. InfiniBand has a feature called "Reliable Datagram" which allows the network-interface to provide a datagram interface, but also take care of reliability. Unfortunately, no InfiniBand vendor has implemented this interface, citing a not-so-optimal specification. There is a huge scope of doing research in this area and addressing the short comings of InfiniBand Reliable Datagram and enabling next generation clusters to scale to hundreds and thousands of nodes.

Leveraging upcoming QoS features of InfiniBand – The next-generation InfiniBand has several new features pertaining to Quality of Service. ConnectX [29] Architecture, a new offering from Mellanox Technologies offers several new features. It is now possible to identify communication channels as, low latency, high bandwidth, best effort, etc. These new features when coupled with MPI library support and integration with job schedulers will provide excellent manageability capabilities to very large scale clusters.

BIBLIOGRAPHY

[1] Aggregate Remote Memory Copy Interface. http://www.emsl.pnl.gov/docs/parsoft/armci/.

[2] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[3] Parallel Virtual File System. http://www.pvfs.org.

[4] Unified Parallel C. http://upc.lbl.gov.

[5] The Defense Advanced Research Projects Agency. The Defense Advanced Research Projects Agency. http://www.darpa.mil/.

[6] 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.

[7] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[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] Cray, Inc. Chapel Programming Language. http://chapel.cs.washington.edu/.

[10] DARPA. High Productivity Computer Systems. http://www.highproductivity.org/.

[11] DAT Collaborative. Direct Access Transport Layer. http://www.datcollaborative.org/udapl.html.

[12] T. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[14] Gordon Moore. Moore's Law. http://www.intel.com/technology/mooreslaw/index.htm.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[16] IBM. The X10 Programming Language. http://www.research.ibm.com/x10/.

[17] InfiniBand Trade Association. InfiniBand Trade Association. http://www.infinibandta.com.

[18] University of Tennessee Innovative Computing Laboratory. HPC Challenge Benchmark Suite. http://icl.cs.utk.edu/hpcc/index.html.

[19] Intel Corporation. The Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm.

[20] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/.

[21] M. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters. In *Int'l ACM Conference on Supercomputing*, June 2007.

[22] M. Koop, S. Sur, and D. K. Panda. Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram. In *IEEE Int'l Conference on Cluster Computing (Cluster)*, 2007.

[23] X. Li and J. Demmel. SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110 – 140, 2003.

[24] J. Liu. *Designing High Performance and Scalable MPI over InfiniBand*. PhD dissertation, The Ohio State University, Department of Computer Science and Engineering, September 2004.

[25] J. Liu and D. K. Panda. Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand. In *Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS*, 2004.

[26] J. Liu, J. Wu, , and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, 32(3), June 2004.

[27] MCS, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[28] Mellanox. Verbs Application Programming Interface (VAPI). http://www.mellanox.com.

[29] Mellanox Technologies. ConnectX Architecture. http://www.mellanox.com/products/connectx_architecture.php.

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

[31] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[32] Myricom. Myrinet. http://www.myri.com/.

[33] Myricom Inc. Portable MPI Model Implementation over GM, March 2004.

[34] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand. http://nowlab.cse.ohio-state.edu/projects/mpi-iba.

[35] OpenFabrics Alliance. OpenFabrics. http://www.openfabrics.org/, April 2006.

[36] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[37] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *Supercomputing*, 2002.

[38] Quadrics. MPICH-QsNet. http://www.quadrics.com.

[39] Sandia National Laboratories. Thunderbird Linux Cluster. http://www.cs.sandia.gov/platforms/Thunderbird.html.

[40] J. Shalf, S. Kamil, L. Oliker, and David Skinner. Analyzing UltraScale Application Communication Requirements for a Reconfigurable Hybrid Interconnect. In *Supercomputing*, 2005.

[41] Sun Microsystems. Fortress Programming Language. http://fortress.sunsource.net/.

[42] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[43] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2006.

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

[45] The Top 500 Project. The Top 500. http://www.top500.org/.

[46] Xiaoye Sherry Li, James Demmel, John R. Gilbert. SuperLU. http://crd.lbl.gov/~xiaoye/SuperLU/.